Attack Tree Generation by Policy Invalidation

Marieta Georgieva Ivanova[1], Christian W. Probst[1](✉), René Rydhof Hansen[2], and Florian Kammüller[3]

[1] Technical University of Denmark, Lyngby, Denmark
{mgiv,cwpr}@dtu.dk
[2] Aalborg University, Aalborg, Denmark
rrh@cs.aau.dk
[3] Middlesex University, London, UK
f.kammueller@mdx.ac.uk

Abstract. Attacks on systems and organisations increasingly exploit human actors, for example through social engineering, complicating their formal treatment and automatic identification. Formalisation of human behaviour is difficult at best, and attacks on socio-technical systems are still mostly identified through brainstorming of experts. In this work we formalize attack tree generation *including* human factors; based on recent advances in system models we develop a technique to identify possible attacks analytically, including technical and human factors. Our systematic attack generation is based on invalidating policies in the system model by identifying possible sequences of actions that lead to an attack. The generated attacks are precise enough to illustrate the threat, and they are general enough to hide the details of individual steps.

1 Introduction

Many attacks against organisations and how to prevent them are well understood. Traditional and well-established risk assessment methods often identify these potential threats, but due to a technical focus, often abstract away the internal structure of an organisation and ignore human factors. However, an increasing number of attacks do involve attack steps such as social engineering.

Attack trees [1] are a loosely defined, yet (or therefore) widely used approach for documenting possible attacks in risk assessment; they can describe attack goals and different ways of achieving these goals by means of the individual steps in an attack. The goal of the defender is then to inhibit one or more of the attack steps, thereby prohibiting the overall attack, or at least making it more difficult or expensive. While attack trees for purely technical attacks may be constructed by automated means [2], this is currently not possible for attacks exploiting the human factors. Actually, only few, if any, approaches to systematic risk assessment take such "human factor"-based attacks into consideration.

Our work closes this gap by developing models and analytic processes that support risk assessment in complex organisations *including* human factors and physical infrastructure. Our approach simplifies the identification of possible

R.N. Akram and S. Jajodia (Eds.): WISTP 2015, LNCS 9311, pp. 249–259, 2015.
DOI: 10.1007/978-3-319-24018-3_16

attacks and provides qualified assessment and ranking of attacks based on the expected impact. Based on earlier work [3,4] we describe a systematic approach for the generation of attack trees for attacks that may include elements of human behaviour. These attack trees can be used as input to a traditional risk assessment process and thereby extend and support the brainstorming results. System models such as ExASyM [5] and Portunes [6] have been used to model and analyse organisations for possible attacks [7]. The models contain both physical infrastructure and information on actors, access rights, and policies; consequently, analysis of such models can include social engineering in the identified attacks. The generated attack trees are complete with respect to the model, that is, our method identifies all attacks that are possible in the model. This is achieved by basing the attack tree generation on invalidation of policies; policies in our model describe both access control to locations and data, as well as system-wide policies such as admissible actions and actor behaviour.

The rest of this paper is structured as follows. After introducing our socio-technical system model and a running example in the next section, we discuss policies in Section 3. These policies are at the core of the attack generation, which is described in Section 4. After evaluating our approach and discussing related work in Section 5 and Section 6, we conclude the paper with an outlook on future developments.

2 Modelling Socio-technical Systems

Our model represents the infrastructure of organisations as nodes in a directed graph [5], representing rooms, access control points, and similar locations. A location may belong to several *domains*, *e.g.*, it can be part of the building and the network. Actors are represented by nodes and are associated with behaviour. Assets model any data relevant in the modelled organisation, and can be annotated with a value and a metric, *e.g.*, the likelihood of being lost. Nodes also represent assets that can be attached to locations or actors; assets attached to actors move around with that actor. Actors perform actions on locations, including physical locations or actors. Actions are restricted by policies that represent both access control and the behaviour as expected by an organisation from its employees. Policies consist of required credentials and enabled actions, representing what an actor needs to provide in order to enable the actions in a policy, and what actions are enabled if an actor provides the required credentials, respectively.

In contrast to Klaim [8], we attach processes to special nodes that move around with the process. This makes the modelling of actors and items carried by actors more intuitive and natural. The metrics mentioned above can represent any quantitative knowledge about components, for example, likelihood, time, price, impact, or probability distributions. The latter could describe behaviour of actors or timing distributions.

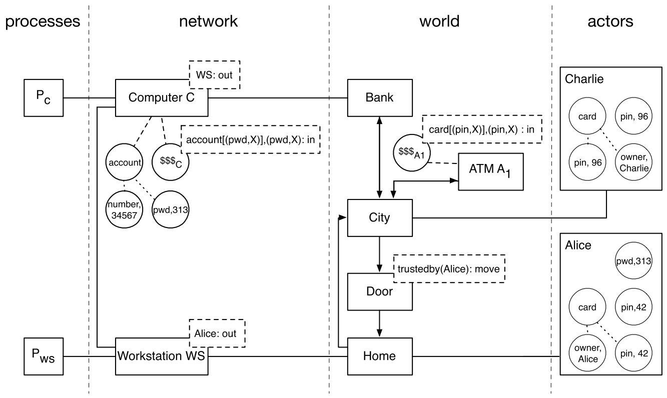

Fig. 1. Graphical representation of the running example. The small rectangles represent locations, the big rectangles represent actors and contain the assets known or owned by the actor, the round nodes represent assets, and the small squares represent process nodes. Solid lines represent the physical connections between locations, while dashed lines indicate containment of information and assets. The dashed rectangles in the upper right part of some nodes represent the policies assigned to these nodes.

2.1 Running Example

We use a running example based on actor Alice, who receives some kind of service, *e.g.*, care-taking, provided by actor Charlie. Charlie's employer has a company policy that forbids him to accept money from Alice. Figure 1 shows the example scenario, consisting of Alice's home, a bank with a bank computer, and an ATM. Alice has a card with a pin code to obtain money, and a password to initiate online transfers from her workstation. The policies in the model require, *e.g.*, a card and a matching pin to obtain money from the ATM.

Actor nodes can also represent processes running on the corresponding locations. The processes at the workstation and the bank computer represent the required functionality for transferring money; they initiate transfers from Alice's home (P_{WS}), and check credentials for transfers (P_C).

3 Policy Language

Our model supports *local policies* for annotating elements with access control polices, and *global policies* for annotating the model with organisational policies. *Local policies* consist of a set of required credentials and a set of *actions* that are enabled by the required credentials: $\mathit{LocalPolicies} \subseteq \mathit{ReqCred} \times \mathit{Actions}$ with $\mathit{Actions} \subseteq \{\mathbf{in}, \mathbf{out}, \mathbf{move}, \mathbf{eval}\}$. The actions come from the set of actions supported by the modelling formalism acKlaim [5]. To ease presentation, we treat credentials as terms from the term algebra over a suitable signature, yielding a flexible and expressive, yet simple, formalisation. The signature is chosen based on a concrete system model, and contains enough structure to represent the model's important features. In our running example, we would expect the signature to at least contain such elements as cards, pin codes, locations, accounts, and actor ids. As signatures depend on the model, we only assume the existence of relevant signatures Σ for assets and predicates, and define required credentials as $\mathit{ReqCred} = \mathcal{P}\left(T\left(\Sigma\right)\right)$ where $T(\Sigma)$ is the term algebra generated by Σ.

Checking whether an actor provides the required credentials of a policy is based on the set of concrete credentials $\mathit{ProvCred} = \mathcal{P}\left(T\left(\Sigma\right)\right)$ that an actor has. Using first order unification as defined by Robinson [9] we determine if a set $c \subseteq \mathit{ProvCred}$ of credentials is valid with respect to a given set $r \subseteq \mathit{ReqCred}$ of required credentials: if c and r can be successfully unified, then the credentials c are sufficient to satisfy the required credentials r of a given policy.

Our policy language supports variables for generic policies; these are left out for space reasons. The system model also supports predicates in credentials. Predicates are used to establish facts about actors; in the example a predicate *isEmployee* could express that the actor is an employee of the service provider, and *isCustomer* could express that the actor is a customer of the company. Predicates are specified in the model, and become part of the knowledge base used in unification, and consequently the term algebra.

Global policies express organisational policies in the system model, describing a state or actions that are disallowed in the system and are to be enforced system-wide. We assume two basic kinds of organisational policies: action-based global policies forbid actors to perform certain actions, and location-based global policies forbid data to reach certain locations. Action-based global policies are specified like local policies with required credentials and a set of actions, and contain a component identifying the attacker: $\mathit{GlobalActionPolicies} = (\mathit{Actors} \cup \mathit{Vars}) \times \mathit{Credentials} \times \mathit{Actions}$. Of course, the set of actions here specifies the prohibited actions. Location-based global policies are considerably simpler, since they only specify an asset and a location $\mathit{GlobalLocationPolicies} = \mathit{Asset} \times \mathit{Location}$.

In the rest of this paper we only consider action-based global policies, which generalise location-based global policies: for data to reach a location it either must be co-located with an actor, who must have input the data, or it must have been output at that location, which in turn again requires that an actor has input the data. Location-based policies can therefore be translated to an action-based global policy that forbids inputing the data in question.

In the example from Figure 1, the global action-based policy could specified to be $\mathit{not}(\{X, \mathit{isEmployee}(X), \mathit{card}[(\mathit{owner}, Y)], \mathit{isCustomer}(Y)\}, \{\mathbf{in}\})$, stating that an actor X is not allowed to use a card as credential when performing an **in** action, if the predicate *isEmployee* is true for X and the card is owned by an actor Y, for whom the predicate *isCustomer* holds. In the example, the only possible binding for X is Charlie, and the only possible binding for Y is Alice, and the **in** action would represent obtaining money at an ATM.

4 Policy Invalidation and Attack Tree Generation

We are now ready to present the main contribution of our work, the generation of attack trees by invalidating policies. We choose attack trees as a succinct way of representing attacks; they are defined by

Definition 1. *$AT := (N_i \dot\cup N_l, n, E, L)$ is an attack tree with inner nodes $N_i := N_\wedge \dot\cup N_\vee$ and leaf nodes N_l, a root node $n \in N_i$, directed edges $E \subseteq N_i \times N_i \dot\cup N_l$, and a labelling function $L := N \to \Sigma^\star$. Nodes in AT are conjunctions ($N_\wedge$) or disjunctions ($N_\vee$) of sub-attacks, or basic actions (N_l). Let $\mathcal{N}^{label}$ be the attack tree that only contains one node n that is mapped by L to label. For $AT_1 = (N_1, n_1, E_1, L_1)$ and $AT_2 = (N_2, n_2, E_2, L_2)$, $kind \in \{\vee, \wedge\}$, label being a string, and $n \in N_{kind}$, we define the addition of attack trees as $AT_1 \oplus^{label}_{kind} AT_2 := (N_1 \cup N_2 \cup \{n\}, E_1 \cup E_2 \cup \{(n, n_1), (n, n_2)\}, n, L_1 \cup L_2 \cup \{(n, label\})$.*

We assume an implicit, left to right order for children of conjunctive nodes. For example, an attacker first needs to move to a location before being able to perform an action.

On a high level, our approach for invalidating a policy consists of four basic steps:

1. Choose the policy to invalidate, and identify the possible actors who could do so; these are the potential attackers.
2. Identify a set of locations where the prohibited actions can be performed. Since there might be several possible actions, this results in a set of pairs of location and action.
3. Recursively generate attacks for performing these actions. This will also identify required assets to perform any of these actions, and obtain them.
4. Finally, move to the location identified in the second step and perform the action.

It should be noted that all rules specified below either block if no valid result can be computed, or return an empty attack tree, for example, if no credentials are required. The rules take as input an infrastructure component $\mathcal{I}$, which represents the socio-technical security model described in Section 2, and an actor component $\mathcal{A}$, which stores identities, locations, and assets collected and reached by an actor during an attack. Also note that we extend rules from working on singular elements to sets by unifying the results of rule applications.

Identify Attackers. To start attack generation from a global policy (see Figure 2), we compute the unification of the global policy and the set of all actors, identify the set of attackers by means of function *getAttacker*, which replaces a variable with the identified bindings, or returns an explicitly specified attacker:

$$getAttacker_{\mathcal{I}}(a, \sigma) := \begin{cases} \{a\} \text{ if } a \in \mathsf{N}_a \\ \sigma(a) \text{ if } a \in Vars \end{cases}$$

$$\frac{\begin{array}{c}\sigma = \mathit{unify}_{\mathcal{I}}(\mathit{Actors}, \mathit{credentials}) \\ \mathit{attackers} = \mathit{getAttacker}_{\mathcal{I}}(\mathit{actor}, \sigma) \qquad \mathit{goals} = \mathit{applicableAt}_{\mathcal{I}}(\mathit{credentials}, \mathit{enabled}, \sigma) \\ \mathcal{I}, \mathit{attackers}, \mathit{goals} \vdash_{\mathit{goal}} \mathit{trees} \qquad \mathcal{T} := \oplus_{\vee}^{\text{“perform any actions”}} \mathit{trees}\end{array}}{\mathcal{I}, \mathit{not}(\mathit{actor}, \mathit{credentials}, \mathit{enabled}) \vdash_{P} \mathcal{T}}$$

Fig. 2. Attack generation starts from the global action-based policy *not*(*actor*, *credentials*, *enabled*). Attack trees are generated for all possible policy violations. As every attack tree represents a violation of the policy, the resulting attack trees are combined by an *or* node.

$$\frac{\mathcal{I}, \mathcal{A}, \mathit{goto}(\mathit{location}) \wedge \mathit{perform}(\mathit{action}) \vdash_{GP} \mathcal{T}}{\mathcal{I}, \mathcal{A}, (\mathit{location}, \mathit{action}) \vdash_{\mathit{goal}} \mathcal{T}}$$

$$\frac{\mathcal{I}, \mathcal{A}, \mathit{goto}(l) \vdash_{\mathit{goto}} \mathcal{T}_{\mathit{goto}}, \mathcal{A}' \qquad \mathcal{I}, \mathcal{A}', \mathit{perform}(a) \vdash_{\mathit{perform}} \mathcal{T}_{\mathit{action}}, \mathcal{A}''}{\mathcal{I}, \mathcal{A}, \mathit{goto}(l) \wedge \mathit{perform}(a) \vdash_{GP} \mathcal{T}_{\mathit{goto}} \oplus_{\wedge}^{\text{“goto } l \text{ and perform } a\text{”}} \mathcal{T}_{\mathit{action}}, \mathcal{A}''}$$

Fig. 3. For each identified goal (consisting of a location and an action) an attacker moves to the location and performs the action. The rules result in an attack tree and a new state of the attacker, which includes the obtained keys and reached locations.

Identify Target Locations. We then compute all locations at which one of the actions in *enabled* could be applied using the credentials specified in the policy. The function *applicableAt* identifies all these locations in the system model and returns goals as pairs of actions and locations.

Attack Generation. The rules in Figure 3 connect the identified goals with the generation of attack trees. For each goal we generate two attack trees: moving to the location and performing the action. While moving to the location new credentials may be required; as a result, the actor acquires new knowledge, which is stored in the actor component $\mathcal{A}$. The rules in Figure 4 and Figure 5 generate attack trees for moving around, performing actions, and obtaining credentials, resulting in attack trees for every single action of the attacker. The resulting trees are combined in the overall attack tree. The function *missingCredentials* uses the unification described above to match policies with the assets available in the model. This implies that all assets that can fullfil a policy are identified; the attack generation then generates one attack for each of these assets, and combines them with a disjunctive node.

For space reasons we do not discuss the interaction between actors and processes, and for the global policy chosen in this example, this is not necessary either. Another global action-based policy could forbid in general to obtain money that has been “owned” by a customer before. In this case, the processes defined in Section 2.1 for the work station and the bank computer would become important, as they allow to transfer money from Alice’s to Charlie’s account. When invalidating this global policy one has to consider asset flow.

$$\frac{\mathit{paths} = \mathit{getAllPaths}_{\mathcal{I}}(\mathcal{A}, l) \qquad \mathcal{I}, \mathcal{A}, \mathit{paths} \vdash_{\mathit{path}} \mathit{trees}, \mathcal{A}' \qquad \mathcal{T} := \oplus_{\vee}^{\text{“find path to } l\text{”}} \mathit{trees}}{\mathcal{I}, \mathcal{A}, \mathit{goto}(l) \vdash_{\mathit{goto}} \mathcal{T}, \mathcal{A}'}$$

$$\frac{\mathit{missing} = \mathit{missingCredentials}_{\mathcal{I}}(\mathcal{A}, \mathit{path}) \qquad \mathcal{I}, \mathcal{A}, \mathit{missing} \vdash_{\mathit{credential}} \mathit{trees}, \mathcal{A}' \qquad \mathcal{T} := \oplus_{\wedge}^{\text{“get credentials”}} \mathit{trees}}{\mathcal{I}, \mathcal{A}, \mathit{path} \vdash_{\mathit{path}} \mathcal{T} \oplus_{\wedge}^{\text{“get credentials and pass path”}} \mathcal{N}^{\text{pass path}}, \mathcal{A}'}$$

Fig. 4. Going to a location and performing an action results in two attack trees. The function *getAllPaths* returns all paths from the current locations of the actor to the goal location l, and the resulting attack trees are alternatives for reaching this location.

$$\frac{i \notin \mathit{identities} \implies \mathcal{T} = \mathcal{N}^{\text{obtain identity } i}}{\mathcal{I}, (\mathit{identities}, \mathit{locations}, \mathit{assets}), \mathit{identity}\ i \vdash_{\mathit{credential}} \mathcal{T}, (\mathit{identities} \cup \{i\}, \mathit{locations}, \mathit{assets})}$$

$$\frac{\begin{array}{c}\mathcal{A} = (\mathit{identities}, \mathit{locations}, \mathit{assets}) \wedge a \notin \mathit{assets} \implies \\ \mathit{goals} = \mathit{availableAt}_{\mathcal{I}}(a) \qquad \mathcal{I}, \mathcal{A}, \mathit{goals} \vdash_{\mathit{goal}} \mathit{trees}, \mathcal{A}' \qquad \mathcal{T} := \oplus_{\vee}^{\text{“get a”}} \mathit{trees}\end{array}}{\mathcal{I}, \mathcal{A}, \mathit{asset}\ a \vdash_{\mathit{credential}} \mathcal{T}, \mathcal{A}'}$$

$$\frac{\mathcal{I}, \mathcal{A}, \mathit{predicate}\ p(\mathit{arguments}) \vdash_{\mathit{predicate}} \mathit{trees}, \mathcal{A}' \qquad \mathcal{T} := \oplus_{\vee}^{\text{“fullfil predicate } p\text{”}} \mathit{trees}}{\mathcal{I}, \mathcal{A}, \mathit{predicate}\ p(\mathit{arguments}) \vdash_{\mathit{credential}} \mathcal{T}, \mathcal{A}'}$$

Fig. 5. Depending on the missing credential, different attacks are generated. If the actor lacks an identity, an attack node representing an abstract social engineering attack is generated, for example, social engineering or impersonating. If the missing credential is an asset, the function *availableAt* returns a set of pairs of locations from which this asset is available, and the according **in** actions. If the missing credential is a predicate, a combination of credentials fulfilling the predicate must be obtained.

Post-Processing Attack Trees. The generated attack trees do not contain annotations or metrics about the success likelihood of actions such as social engineering, or the potential impact of actions. Also the likelihood of a given attacker to succeed or fail is not considered. Computing qualitative and quantitative measures on attack trees is beyond the scope of this work. The generated attack trees also often contain duplicated sub-trees, due to similar scenarios being encountered in several locations, for example, the social engineering of the same actor, or the requirement for the same credentials. This is not an inherent limitation, but may clutter attack trees. Similar to [2], a post-processing of attack trees can simplify the result.

5 Evaluation

We now describe briefly the attack generation based on the results of a prototype implementation. The attack tree shown in Figure 6 is generated from

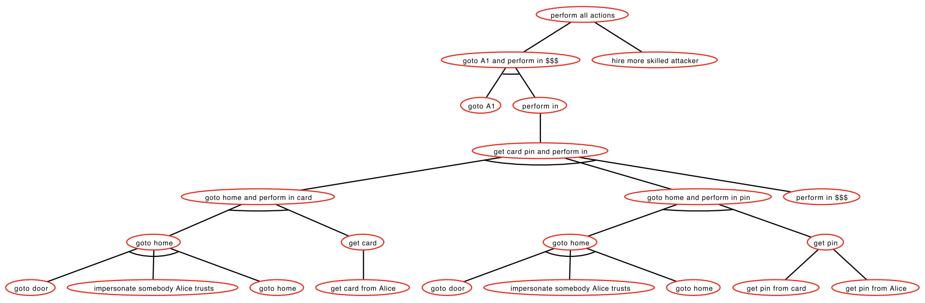
perform all actions
goto A1 and perform in $$$
hire more skilled attacker
goto A1
perform in
get card pin and perform in
goto home and perform in card
goto home and perform in pin
perform in $$$
goto home
get card
goto home
get pin
goto door
impersonate somebody Alice trusts
goto home
get card from Alice
goto door
impersonate somebody Alice trusts
goto home
get pin from card
get pin from Alice

Fig. 6. Attack tree generated by the prototype implementation for the example shown in Figure 1.

the example scenario. As mentioned in the previous section, we assume the global policy that an employee is not allowed to use a customer's card to obtain money: $not(\{X, isEmployee(X), card[(owner, Y)], isCustomer(Y)\}, \{\mathbf{in}\})$. Using the rule from Figure 2, we compute the substitution $\sigma = [X \mapsto Charlie, Y \mapsto Alice]$ for variables X, Y: Charlie has the role employee, and Alice has the role customer. In the next step, the attacker is identified to be X, and based on the system specification from Figure 1, the only location with a policy restricting the **in** action is the money location at the ATM A1. The location and action pair $\{(A1, \mathbf{in})$ is therefore the only goal, and next $\vdash_P$ from Figure 4 generates the attack tree for moving to $A1$ and performing the **in** action.

Going to the location does not require additional credentials, but performing the **in** action does. The *missingCredentials* function returns the card and the pin, which combined with the requirement from the goal policy, that the owner of the card must be Alice, implies that the attacker needs Alice's card and pin. The second rule $\vdash_{credential}$ in Figure 5 identifies where they are: Alice has the card and the pin, and the pin code is also stored in the card. Our approach generates an attack tree for going to the location "home", and in doing so the attacker must fullfil the policy "trustedBy(Alice)", meaning that he must impersonate somebody trusted by Alice. Then the attacker can either "input" the card and the pin, or only the card and try to extract the pin code from the card.

The stealing and the extraction of the pin code are not represented in the model since they are context and technology dependent. In a given scenario, they can be instantiated with the matching "real" actions. After the assets have been obtained, the attacker moves to the ATM location and performs the action.

6 Related Work

System models such as ExASyM [5,7] and Portunes [10] also model infrastructure and data, and analyse the modelled organisation for possible threats. However, Portunes supports mobility of nodes, instead of processes, and represents the social domain by low-level policies that describe the trust relation between people to model social engineering. Pieters *et al.* consider policy alignment to address different levels of abstraction of socio-technical systems [11], where policies are interpreted as first-order logical theories containing all sequences of actions and expressing the policy as a "distinguished" prefix-closed predicate in these theories. In contrast to their use of refinement for policies we use the security refinement paradox, *i.e.*, security is *not* generally preserved by refinement.

Attack trees [12] specify an attacker's main goal as the root of a tree; this goal is then disjunctively or conjunctively refined into sub-goals until the reached subgoals represent basic actions that correspond to atomic components. Disjunctive refinements represent alternative ways of achieving a goal, whereas conjunctive refinements depict different steps an attacker needs to take in order to achieve a goal. Techniques for the automated generation of attack graphs mostly consider computer networks only [13,14]. While these techniques usually require the specification of atomic attacks, in our approach the attack consists in invalidating a policy, and the model just provides the infrastructure and methods for doing so.

7 Conclusion

Threats on systems are often described by attack trees, which represent a possible attack that might realise the described threat. These attack trees are usually collected by experts based on a combination of experience and brainstorming. Earlier work has tried to formalise this approach for threats on technical systems. The increasingly important human factor is often not considered in these formalisations, since it is not part of the model.

In this work we have formalizes attack tree generation *including* human factors using recent advances in system models. Our approach supports all kinds of human factors that can be instantiated once an attack has been identified. To the best of our knowledge this is the first formalisation of an approach to generating attack trees including steps on the technical and social level.

The generated attacks include all relevant steps from detecting the required assets, obtaining them as well as any credentials needed to do so, and finally performing actions that are prohibited in the system. The generated attacks are precise enough to illustrate the threat, and they are general enough to hide the details of individual steps. The generated attacks are also complete with respect to the model; whenever an attack is possible in the model, it will be found. Our approach is also sound; all results of our generator do represent attacks.

The combination of system model and automated generation enables us to trade in precision of the model for details in the attack trees. For example, the modelling of the ATM is very imprecise in the example from Figure 1. A more detailed formalisation would represent that an actor puts the card and the pin code into the ATM and receives money after a check with the bank. In this model, the attack tree generator is able to find out that one can obtain the pin code from the ATM, since it is input into the system. Note that the first action in the second line only verifies whether the pin code entered into the ATM is the one stored on the card. To handle more general global policies that might, for example, prohibit to own money obtained by some other actor, we can use techniques such as tainting to trace which actor or credentials have been used to obtain or handle an asset. In the example described in Section 2.1, using Alice's credit card would result in the withdrawn money being tagged with her id.

We are currently working on further evaluations and domain-specific languages to extend the model's expressivity, and are extending the attack generation to simplify the generated attack tree during generation.

Acknowledgment. Part of the research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318003 (TREsPASS). This publication reflects only the authors' views and the Union is not liable for any use that may be made of the information contained herein.

References

1. Schneier, B.: Attack Trees: Modeling Security Threats. Dr. Dobb's Journal of Software Tools **24**(12), 21–29 (1999)
2. Vigo, R., Nielson, F., Nielson, H.R.: Automated generation of attack trees. In: Proceedings of the 27th Computer Security Foundations Symposium (CSF), pp. 337–350. IEEE (2014)
3. Kammüller, F., Probst, C.W.: Invalidating policies using structural information. In: 2nd International IEEE Workshop on Research on Insider Threats (WRIT 2013). IEEE (2013)
4. Kammüller, F., Probst, C.W.: Combining generated data models with formal invalidation for insider threat analysis. In: 3rd International IEEE Workshop on Research on Insider Threats (WRIT 2014). IEEE (2014)
5. Probst, C.W., Hansen, R.R.: An extensible analysable system model. Information Security Technical Report **13**(4), 235–246 (2008)
6. Dimkov, T., Pieters, W., Hartel, P.: Portunes: representing attack scenarios spanning through the physical, digital and social domain. In: Armando, A., Lowe, G. (eds.) ARSPA-WITS 2010. LNCS, vol. 6186, pp. 112–129. Springer, Heidelberg (2010)
7. Probst, C.W., Hansen, R.R., Nielson, F.: Where can an insider attack? In: Dimitrakos, T., Martinelli, F., Ryan, P.Y.A., Schneider, S. (eds.) FAST 2006. LNCS, vol. 4691, pp. 127–142. Springer, Heidelberg (2007)
8. de Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. IEEE Trans. Softw. Eng. **24**(5), 315–330 (1998)
9. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
10. Dimkov, T.: Alignment of Organizational Security Policies - Theory and Practice. University of Twente (2012)
11. Pieters, W., Dimkov, T., Pavlovic, D.: Security policy alignment: A formal approach. IEEE Systems Journal **7**(2), 275–287 (2013)

12. Salter, C., Saydjari, O.S., Schneier, B., Wallner, J.: Toward a secure system engineering methodology. In: Proceedings of the 1998 Workshop on New Security Paradigms (NSPW), pp. 2–10 (September 1998)
13. Phillips, C., Swiler, L.P.: A graph-based system for network-vulnerability analysis. In: Proceedings of the 1998 workshop on New security paradigms NSPW 1998, pp. 71–79 (1998)
14. Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.M.: Automated generation and analysis of attack graphs. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P 2002), vol. 129, pp. 273–284 (2002)