Variability meets Security

Quantitative Security Modeling and Analysis of Highly Customizable Attack Scenarios

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy
maurice.terbeek@isti.cnr.it

Axel Legay
UCLouvain, Louvain-la-Neuve, Belgium
axel.legay@uclouvain.be

Alberto Lluch Lafuente
DTU, Lyngby, Denmark
albl@dtu.dk

Andrea Vandin
DTU, Lyngby, Denmark
Sant'Anna School of Advanced Studies, Pisa, Italy
anvan@dtu.dk

ABSTRACT

We present a framework for quantitative security modeling and analysis of highly customizable attack scenarios, which resulted as a spin-off from our research in software product line engineering. The graphical security models are based on attributed attack-defense diagrams to capture the structure and properties of vulnerabilities, defenses and countermeasures—with notable similarities to feature diagrams—and on probabilistic models of attack behavior, capable of capturing resource constraints and attack effectiveness. In this paper, we provide an overview of the framework that is described in full technical detail in twin papers, which present the formal syntax and semantics of the domain-specific language and showcase the associated tool with advanced IDE support for performing analyses based on statistical model checking. The properties of interest range from average cost and success probability of attacks to the effectiveness of defenses and countermeasures. Here we illustrate the capabilities of the DSL and the tool by applying them to an example scenario from the security domain. This shows how techniques from variability modeling can be applied to security. We conclude with a vision and roadmap for future research.

CCS CONCEPTS

• **Software and its engineering** → **Specification languages**; **Formal methods**; **Software product lines**; **Extra-functional properties**; • **Security and privacy** → **Formal methods and theory of security**; **Formal security models**.

KEYWORDS

Variability models, graphical security models, attack-defense trees, quantitative security, statistical model checking, formal analysis tools

VaMoS '20, February 5–7, 2020, Magdeburg, Germany

ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377041

ACM Reference Format:
Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2020. Variability meets Security: Quantitative Security Modeling and Analysis of Highly Customizable Attack Scenarios. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20), February 5–7, 2020, Magdeburg, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377024.3377041

1 INTRODUCTION

The scope of this paper is to show how variability modeling and analysis techniques can be applied in the security domain. We are primarily interested in graphical representations for attack scenarios, and in analyzing the feasibility of an attack scenario on a specific system. An attack scenario is a set of attack steps, each of them representing the element of a successful attack. Depending on the product under attack, on existing vulnerabilities, and on available resources, the scenario may slightly vary. To make a parallel with software product lines, one can view each attack scenario as a product, and a set of attack scenarios as a product line.

Graph-based security models offer an intuitive mean to represent vulnerabilities in complex software-intensive systems, which can then be used for formal analysis. Many such models are based on attack trees [29, 33], which are conceptually very similar to feature diagrams as we know them. Essentially, an attack tree is an and/or tree, whose nodes represent goals and whose sub-trees represent sub-goals. For instance, when modeling the risk assessment of malicious system access, the root represents the main threat under analysis, i.e. the system being accessed by an attacker, with each of its children representing possible ways of enacting such a threat, and possibly decorated with an estimation of the cost that the attacker would have to pay to succeed in enacting the corresponding action. Classical analyses of such trees concern computing properties related to the cost for an attacker to succeed.

Attack trees suffer from the same limitation as feature diagrams, that is they do not permit to specify an order in variability selection. This is problematic as attack scenarios must be evaluated on concrete systems, which are reacting in a dynamic defense manner to successive and ordered attack steps. In order to mitigate this problem, we combine attack trees with a behavioral model, just like feature diagrams have been combined with featured transition systems [14, 15]. More precisely, the graphical security model for quantitative security modeling we propose here is based on *declarative* attributed attack-defense trees [21, 22] to model the structure

and properties (in the form of attributes) of vulnerabilities, defenses and countermeasures on the one hand, and on *procedural* probabilistic models [1, 4, 13, 26, 31] to capture attack behavior, modeling resource constraints and attack effectiveness, on the other hand. To make a parallel with software product lines, we are in the situation where attack trees represent variability in the products, and then probabilistic models represent the behavioral process to select attack steps towards a successful attack scenario of the tree, while dynamically reacting to countermeasures.

One limitation of the current approach is that security engineers do not have the skills to work with those formal representations. This calls for a new DSL model that encompasses both variability and behavioral aspects of both attack scenarios and systems' reactivity. To do so, we propose to extend a domain specific language (DSL) that we recently offered in the context of software product lines [9, 35]. Our goal really is to offer a high-level DSL equipped with a formal semantics, which is hidden to the end user by a multi-platform one-click-install integrated development environment (IDE) for the DSL. Through tool-chaining with the statistical analyzer MultiVeStA [34], the overall framework offers a means to study properties like the success probability and average cost of attacks or the effectiveness of defenses and countermeasures. As such, the framework we propose can be seen as a recast for the security domain of a recent approach for software product line engineering [9, 35], which was indirectly applied to a confined security scenario—yet with limitations that required intermediate encoding into security notions.

Over the last five years we developed an integrated modeling and analysis approach for configurable systems such as software product lines, based on a family of formal specification languages (FLan [10], PFLan [6], and QFLan [7], surveyed in [8] and culminating in [9, 35]). In [10], we defined a basic feature-oriented language FLan. Its distinguishing modeling feature that was maintained in all improvements of the language to date is a clean separation between configuration (in terms of features) and behavior (in terms of actions). This is achieved by a rich process algebra with an associated store of constraints to dynamically confine the possible configurations and behavior. In [6], we moved to probabilistic modeling in PFLan by equipping actions with a rate to capture notions like uncertainty and failure that permit the analysis of the likelihood of installing features or of probabilistic behavior, resulting in the first application of statistical model checking in software product line engineering. In [7], we enriched PFLan with advanced quantitative constraint modeling options regarding the cost of features (i.e. non-functional feature attributes like price or reliability). We defined quantitative constraints as arithmetic relations among attributes, giving rise to rich action constraints and propositions constraining the presence of features. In [9, 35], we presented the full QFLan framework, including mature Java-based tool support with a modern IDE that extends the previous prototypical implementations based on a Maude interpreter combined with the distributed statistical model checker MultiVeStA [34] and SAT/SMT solving with Z3. We refer to [9] for a detailed presentation of related approaches.

The aforementioned limitations in applying QFLan to the security domain motivated the redevelopment from scratch of a DSL and accompanying tool based on the main intuitions underlying [9, 35], now tailored towards more general security scenarios. The novelty of our approach concerns the framework's unique (as far as we know) dual declarative-procedural nature inherited from the FLan family of feature-oriented languages, by which the combination of structural diagrams and dynamic behavior allow to study the vulnerabilities of a system for specific classes of attackers. The dynamic behavior is inspired by so called attack profiles specified in the form of automata, describing possible attack steps and their costs (cf., e.g., [17, 19, 23, 25, 28]). It is important to note, however, that our attack behavior is driven by global quantitative constraints on costs, like "*step A can be performed because the total costs of the steps performed so far is less than X*", which is novel. The DSL combines common features from established approaches, such as countermeasures [32], attack detection rates [2], ordered attacks [12, 30], and further common features from vulnerability analysis like attack effectiveness. Furthermore, to remedy a limitation common to most approaches, the DSL allows nodes to have multiple parents, which is convenient to specify an attack (defense) node that affects multiple defenses (attacks), or an attack (defense) node that refines many attacks (countermeasures), without the need to duplicate nodes.

The statistical model checking analysis capabilities, via tool-chaining with MultiVeStA, are based on executing a sufficient (and minimal) set of probabilistic simulations of the behavioral attack model to obtain statistical evidence (with a predefined level of statistical confidence) of the quantitative properties being verified. Properties are formulated in MultiVeStA's property specification language MultiQuaTEx [34]. The advantages of statistical model checking over exhaustive (probabilistic) model checking are manifold. First, there is no need to generate the entire state space, thus offering improved scalability by avoiding the combinatorial state-space explosion problem typical of model checking. In case of highly customizable scenarios, this outweighs the disadvantage of refraining from precise results (100% confidence) with exact analysis techniques like (probabilistic) model checking. Second, statistical model checking is known to scale better with hardware resources, since the set of simulation runs to be executed offers an intrinsic way to parallelize and distribute. In fact, MultiVeStA can run on multi-core machines, clusters or distributed computers with a nearly linear speedup. Finally, MultiVeStA can use one and the same set of simulation runs to check multiple properties at a time, resulting in further reductions of computing time.

Synopsis. In this paper, we give an overview of the framework that will be described in full detail in twin papers where we will (i) present the formal syntax of the domain-specific language with its semantics based on weighted transition systems and (ii) showcase the associated tool with advanced IDE support for performing analyses based on statistical model checking. In Sect. 2, we illustrate the capabilities of the framework by applying it to an example scenario from the security domain. This shows how techniques from variability modeling can be applied in that domain. Sect. 3 concludes the paper with a vision and roadmap for future research. Related work is cited throughout the paper wherever appropriate.

2 EXAMPLE SCENARIO

Our example is based on the safe lock scenario from Schneier's seminal work on attack trees [33], which we extend slightly to better illustrate our approach.

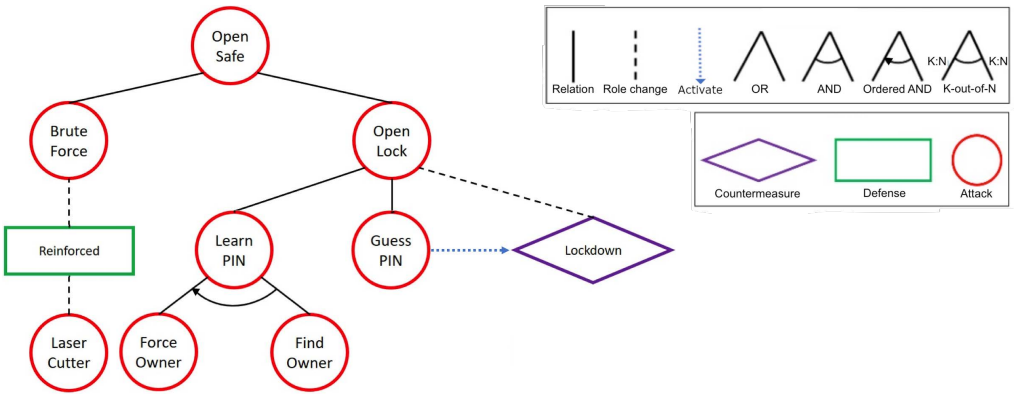

Figure 1: Attack-defense diagram

According to Schneier, an attack tree is a tree whose nodes represent attacks, with the root representing an attacker's global goal. Child nodes represent refinements of this goal, meaning leaf nodes represent attacks that cannot be refined any further. Schneier distinguishes only two types of nodes, OR nodes and AND nodes, modelling alternative (disjunctive) ways of achieving the same goal and different (conjunctive) steps towards achieving the same goal, respectively. Upon the addition of attributes, like the cost of an attack, the value of attacks (such as the cost of the cheapest attack or all the attacks that less than a certain value) can be synthesized.

2.1 Structure

The example scenario, an extension of Schneier's safe lock scenario from [33], is depicted as an attack-defense diagram in Figure 1. Circles denote *attack nodes*, i.e. (sub-)goals to be accomplished to succeed in the attack. The goal is to open a safe, the root `OpenSafe`, which can be accomplished by `BruteForce` or `OpenLock` attacks. The latter can in turn be obtained via `LearnPIN` or `GuessPIN` attacks. `LearnPIN` requires two conditions, in a given order: first to find the owner of the PIN, and then to force her/him to reveal it.

In most related approaches (such as, e.g., ADTrees [21] supported by the ADTool [20]), attack nodes are divided into attack actions, which represent operations performed by an attacker, and attack goals, which are implicitly activated under the conditions specified by the attack diagram's structure (i.e., an attack goal is automatically activated when the necessary sub-goals become activated). We, instead, require attack nodes to be explicitly activated by attack behavior. Thus, an attacker that has accomplished `GuessPIN` must still explicitly accomplish also `OpenLock`. This implies very fine-grained modeling of the dynamics of an attack and of reactive defenses, since the activation of `GuessPIN` might activate the countermeasure `Lockdown`, which in turn will affect `OpenLock`.

Besides attack nodes, our framework also exhibits *defense* and *countermeasure* nodes, to model static and dynamic defensive measures, respectively. The *defense* `Reinforced` (denoted by a rectangle) is permanently active against `BruteForce` attacks. Defenses can be disabled by attack nodes: if the attacker uses a `LaserCutter`, s/he can break `Reinforced`. However, some defenses are reactive, responding to the detection of an attack attempt (cf. [32]). Such is the *countermeasure* `Lockdown` (denoted by a diamond), which can be triggered by `GuessPIN` attacks (denoted by a dotted arrow) and which, once it has been activated, affects `OpenLock` attacks (denoted by a dashed line). Countermeasures offer limited, reactive defender behavior, whereas defense nodes are considered active by default. These defensive measures enable interesting analyses on the effectiveness and cost/benefits of a defense or of an attack, like "is it worth defending/attacking the system in a given way?".

DSL Code 1 shows how the nodes of our example scenario are declared. Note that when declaring a countermeasure node it is also necessary to specify which attack nodes can trigger it.

```
begin attack nodes
 OpenSafe BruteForce OpenLock GuessPIN
 LearnPIN FindOwner ForceOwner LaserCutter
end attack nodes

begin defense nodes
 Reinforced
end defense nodes

begin countermeasure nodes
 Lockdown = { GuessPIN }
end countermeasure nodes
```

Code 1: Nodes

In our framework, the diagrams structure nodes according to two types of relations: *refinements* specify how an offensive (defensive) node is structured into offensive (defensive) sub-nodes, while *role-changes* specify how an offensive (defensive) node is opposed by a defensive (offensive) node. Each node has at most one refinement and at most one role-change. In Figure 1, edges are implicitly directed downwards. Different from most approaches, nodes can have multiple parents, which is convenient to specify one attack (defense) node that affects multiple defenses (attacks), or an attack (defense) node that refines many attacks (countermeasures).

Lines 2-4 of DSL Code 2 set the hierarchical constraints of our example. The squared brackets of `OAND` indicate that order matters: `LearnPIN` requires `FindOwner` and `ForceOwner` *in that order*.

```
1 begin attack diagram
2  OpenSafe -> { BruteForce, OpenLock }
3  OpenLock -> { GuessPIN, LearnPIN }
4  LearnPIN -OAND-> [ FindOwner, ForceOwner ]
5  BruteForce -> { Reinforced }
6  Reinforced -> { LaserCutter }
7  OpenLock -> { Lockdown }
8 end attack diagram
```

Code 2: Attack-defense diagram

Inspired by other formalisms that support both attack and defensive mechanisms (cf., e.g., [21]), our framework offers a *role-changing* relation to describe which attack a countermeasure or defense works against (e.g. `Reinforced` defends against `BruteForce`) or vice versa (e.g. `LaserCutter` neutralizes `Reinforced`). Lines 5-7 of DSL Code 2 show that attack, defense, and countermeasure nodes can additionally have a *role-changing* relation with a child of the opposite role, an opponent node that affects its activation.

Our framework offers `OR`, `AND`, `OAND` (ordered `AND`), and `k-out-of-n` refinements for attack and countermeasure nodes, whereas defense nodes cannot be refined as they model static, atomic defenses. Actually, countermeasures are also atomic, but we allow to refine them with defense nodes to permit *reactive* defense nodes: defense nodes refining a countermeasure become effective only upon (attack detection and) activation of the refined countermeasure.

`AND` and `OR` refinements originate from the seminal works on attack trees [33]. `OpenSafe` and `OpenLock` have `OR` refinements. The `k-out-of-n` refinements are inspired by [32], expressing that only a certain number of the sub-attacks are required, which is useful syntactic sugar for modeling scenarios related to voting or sensors. `OAND` refinements stem from [12, 30], adding a sequential aspect. This can be interpreted in two ways: either the children can only be activated in a given order or they can be activated in any order, but only the correct order satisfies the relation. Our `OAND` refinements adopt the latter, since orders of attacks can be modeled in an attack behavior, and it might be interesting to model an attacker that does not know the right order of attacks. Learning the PIN requires to first find the owner and then force her/him.

As in many other approaches (cf., e.g., [22]), attack nodes can be decorated with attributes such as cost or detection rates of attacks. The cost of (attempting) an attack can be used to impose constraints, like the maximum cost, which facilitates quantitative analyses that are gaining in popularity [5, 6, 27]. DSL Code 3 shows the attribute `Cost` used in our example to denote the cost of attempting an attack. The default value is 0, e.g. `Cost(GuessPIN) = 0`.

```
begin attributes
 Cost = { LaserCutter = 200, FindOwner = 20,
          ForceOwner = 10, Reinforced = 250 }
end attributes
```

Code 3: Attributes

One is usually interested in computing the cumulative attribute value for the entire scenario. Often, the cost associated to a (subsystem rooted in a) node is the sum of the costs of its active descendants [33]. However, the total cost of an attack is not given only by the costs of *successful* sub-attacks, as this would be a best-case scenario where the attacker never fails. Therefore, in our framework the actual value of an attribute of an attack node considers both successful and failed attack attempts. Furthermore, we allow to specify attributes also for defensive nodes.

An attack detection rate determines the probability for an attack attempt, i.e. the execution of a `succ(·)` or `fail(·)` action, to be detected, and it triggers the activation of the affected countermeasures, in the sense that higher detection rates lead to more likely activation of countermeasures. The default value is 0, i.e. an attack is undetectable. DSL Code 4 specifies this property for our example.

```
begin attack detection rates
 BruteForce = 0.5, OpenLock = 0.1,
 GuessPIN = 0.8, LearnPIN = 0.3,
 FindOwner = 0.6, ForceOwner = 0.7,
 LaserCutter = 0.6
end attack detection rates
```

Code 4: Attack detection rates

Countermeasures are triggered when attacks are detected. Inspired by the notion of *noticeability* from [2], the detection rate of an attack can be explicitly specified to denote how likely it is for the defender to notice that an attack was attempted, affecting the activation of countermeasures. In [2], this property is also used to constrain the stealthiness of an attacker. We therefore allow the addition of explicit constraints on the attack behavior (see below).

In [20, 21], an attack node is *disabled* if it is affected by a defense. However, a common conception in security is that nothing is ever 100% secure. Therefore, we offer the notion of *defense effectiveness* to specify the effectiveness of a defensive node against any combination of attack nodes and attack behavior. The rationale is that different attackers might be affected differently by a defense, even when attempting the same attack (e.g. a security guard is efficient against a thief, but not against a military attack). Defense effectiveness is given as a probability of how likely an attack is thwarted. The default value is 0, i.e. the defense has no effect. The defense effectiveness in our example is given by DSL Code 5.

```
begin defense effectiveness
 Reinforced(ALL, BruteForce) = 0.95
 Lockdown(ALL, OpenLock) = 0.8
end defense effectiveness
```

Code 5: Defense effectiveness (`ALL` denotes any attacker)

2.2 Behavior

Defensive behavior is *reactive* while an attacker is *proactive*. Our framework allows to fine tune the described security scenario by defining *attack behavior*, implicitly constrained by the attack-defense diagram. Explicit attack behavior enables the analyses of specific attacker types such as script kiddies, insiders, hackers, and government agencies. The main advantage in analyzing specific attacker types is the ability of evaluating system vulnerabilities for the attacker types that make more sense for the considered security scenarios. Furthermore, it allows for novel types of analysis that complement the classical best- and worst-case evaluations of attack graphs (e.g. the bottom-up evaluation in ADTool [20]).

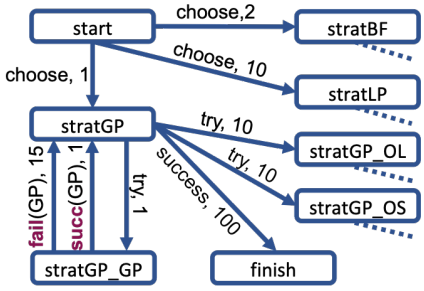

Figure 2: Attack behavior

The combination of explicit (probabilistic) attacker behavior and attack-defense diagrams was mainly motivated by our work on configurable systems, like product lines [9, 35]. We applied it to a restricted class of attack trees in [9], with limitations that motivated the present work. The diagrams in [9, 35] denote sets of products sharing optional features. Each feature is a node which is *installed* by a configurator process similarly to how attack nodes are added by an attacker. However, the diagrams are simpler than those considered here, with only one type of node and one type of refinement and without notions corresponding to defense effectiveness, attack detection rates, `fail(·)` actions, and transition guards (see below).

Attack behavior is modeled as a transition system whose transitions are decorated with the action being executed, weights (used to compute the probability of executing the action), and possibly guards (conditions on the action's executability). Figure 2 sketches the behavior of an attacker with three possible strategies: `stratGP`, `stratBF`, and `stratLP`, which involve `GuessPIN`, `BruteForce`, and `LearnPIN` attacks, respectively. The first of these strategies, `stratGP`, tries the following attacks: `GuessPIN` (abbreviated by `stratGP_GP`), `OpenLock` (`stratGP_OL`), and `OpenSafe` (`stratGP_OS`).

Our framework supports *user-defined actions* for scenario-specific behavior not directly related to the activation of nodes, like `choose`, which denotes the action of choosing a strategy, as well as *built-in actions*, like `succ(·)` or `fail(·)`, which denote the success or failure of an attack attempt, respectively. An attack *attempt* is modeled by a probabilistic choice between `succ(·)` and `fail(·)` actions, where transition weights determine the success likelihood, together with (the effectiveness of) the involved defenses. This is exemplified in Figure 2 by the transitions outgoing from state `stratGP_GP`. Attack behavior can also model attempts of an attacker to obtain information about the sceanrio (like, e.g., whether an attack succeeded). This is done using actions `query(·)`. Finally, a succeeded attack can be removed with `remove(·)`, which may turn out useful in the presence of `OAND` relations, since it allows an attacker to apply backtracking strategies to try different attack sequences.

DSL Code 6 shows the user-defined actions of our example: `choose` is executed when selecting the attack strategy, `try` when attempting an attack, and `success` when the attack has succeeded.

```
begin actions
 choose try success
end actions
```

Code 6: Actions

Informally, a behavior is executed as follows: at each step we consider the outgoing transitions from the current state of the behavior which are admitted by the attack diagram and by further constraints discussed next. Among those, we use their relative weights to probabilistically choose one (e.g., from `start` to `stratGP` with probability $\frac{1}{1+2+10}$).

Attack behavior may be constrained by hierarchical constraints, quantitative constraints, action constraints, and transition guards. Hierarchical constraints are due to attack diagrams. Quantitative constraints are Boolean expressions involving (inequalities or arithmetic expressions over) reals, attributes, and variables. In the example, we constrain the total cost of attacks and the number of attempts, which is particularly interesting when combined with the fact that attack behavior may model failed attacks, by the quantitative constraints given in DSL Code 7: "*the accumulated cost of an attack may not exceed 250*" and "*an attacker may not attempt more than 15 attacks*".

```
begin quantitative constraints
 { sum(Cost) < 250 }
 { AttackAttempts < 15 }
end quantitative constraints
```

Code 7: Quantitative constraints

Finally, action constraints act as guards on any transition executing the given action, while transition guards constrain single transitions. DSL Code 8 shows the action constraints of our example: "*an attacker is not allowed to attempt using a laser cutter if the cost already exceeded 100*".

```
begin action constraints
 do(succ(LaserCutter)) -> {sum(Cost) < 100}
 do(fail(LaserCutter)) -> {sum(Cost) < 100}
end action constraints
```

Code 8: Action constraints

The framework's real-valued variables model context information, thus allowing for rich descriptions of the state of the system, of an attacker, and of the defenses, and they greatly facilitate the analysis phase. Variables can be updated as *side effects* when executing the model, i.e. memory updates which label transitions. This allows one to model scenarios where variable updates are not tied to specific nodes or actions. DSL Code 9 shows the only variable of our example, used to count the attack attempts.

```
begin variables
 AttackAttempts = 0
end variables
```

Code 9: Variables

Attack behavior is completed by specifying the attacker to use and pre-accomplished attacks. This enriches expressiveness, because one can assign an initial advantage to an attacker. Indeed, an attack-defense diagram models all possible attacks, but some attackers (e.g., insiders) might already have access to critical components. This is convenient because it allows one to ignore sub-trees of the diagram without explicitly removing them.

```
1  begin attacker behavior
2   begin attack
3    attacker = clever
4    states = start, finish, stratForce, stratF_OpenSafe, stratF_BruteForce, stratF_LaserCutter,
5      stratLearnPIN, stratLP_OpenSafe, stratLP_OpenLock, stratLP_LearnPIN, stratLP_FindOwner,
6      stratLP_ForceOwner, stratGuessPIN, stratGP_GuessPIN, stratGP_OpenSafe, stratGP_OpenLock
7    transitions =
8     //Pick a strategy:
9     start -(choose,  1)-> stratGuessPIN,
10    start -(choose,  2)-> stratForce,
11    start -(choose, 10)-> stratLearnPIN,
12    //Strategy GuessPIN:
13    stratGuessPIN -(try, 1, allowed(GuessPIN) and !has(GuessPIN))-> stratGP_GuessPIN,
14    stratGP_GuessPIN -(succ(GuessPIN),  1, {AttackAttempts=AttackAttempts+1})-> stratGuessPIN,
15    stratGP_GuessPIN -(fail(GuessPIN), 15, {AttackAttempts=AttackAttempts+1})-> stratGuessPIN,
16    stratGuessPIN -(try, 10, allowed(OpenLock) and !has(OpenLock))-> stratGPOpenLock,
17    stratGP_OpenLock -(succ(OpenLock), 10, {AttackAttempts=AttackAttempts+1})-> stratGuessPIN,
18    stratGP_OpenLock -(fail(OpenLock),  1, {AttackAttempts=AttackAttempts+1})-> stratGuessPIN,
19    stratGuessPIN -(try, 10, allowed(OpenSafe) and !has(OpenSafe))-> stratGP_OpenSafe,
20    stratGP_OpenSafe -(succ(OpenSafe), 10, {AttackAttempts=AttackAttempts+1})-> stratGuessPIN,
21    stratGP_OpenSafe -(fail(OpenSafe),  1, {AttackAttempts=AttackAttempts+1})-> stratGuessPIN,
22    stratGuessPIN -(success, 100, has(OpenSafe))-> finish
23    //Strategies Force and LearnPIN ...
24  end attack
25 end attacker behavior
26
27 begin init
28  clever = { FindOwner }
29 end init
```

Code 10: Attack behavior

DSL Code 10 sketches the attack behavior `clever`. It has a `start` and a `finish` state, and some intermediate ones (Lines 4-6). Lines 9-11 show that `start` can evolve into three states: `stratGuessPIN`, `stratForce`, and `stratLearnPIN`, each implementing an attack strategy involving only the corresponding attacks. As shown in Line 11, the latter strategy has the highest weight, viz. 10, and hence it is chosen with higher probability. Lines 12-22 detail the simplest strategy, `stratGuessPIN`. The transitions in Lines 13-15 try to add `GuessPIN`: if the constraints allow it (`allowed(GuessPIN)`) and it has not been added yet (`!has(GuessPIN)`), then Line 13 executes the user-defined action `try` to move to the intermediate state `stratGP_GuessPIN`. Once there, given that it is unlikely to guess a PIN, it either `succeeds` with weight 1 or `fails` with weight 15 (Lines 14-15) and then returns to `stratGuessPIN`. Notably, the transitions with `succ` and `fail` increment variable `AttackAttempts`, which hence counts the attack attempts performed so far. Lines 16-18 and Lines 19-21 are similar, but regard `OpenLock` and `OpenSafe`. Lines 27-29 complete the specification with an *initial status*: the specific attacker used, `clever`, and the pre-accomplished attacks, `FindOwner`.

2.3 Analysis

We now illustrate two quantitative security analysis capabilities of our framework's tool on the example's customizable probabilistic attack scenario. Recall that statistical model checking is offered via tool-chaining with MultiVeStA, which allows to estimate properties like the average of real-valued observations on the model behavior.

To begin with, we compute the probability for each attack that it is attempted first and that it succeeds, as well as the average steps performed to attempt the first attack. DSL Code 11 expresses these nine properties (one probability per attack node plus the average number of steps, cf. Lines 3-5): `query` specifies the properties to be `evaluated` in the first state `when` `AttackAttempts == 1`. Each property can be an arithmetic expression of nodes (evaluating to 1 or 0 if the node is active or not, respectively), variables, attributes (evaluated with respect to the attacker), or `steps`, evaluated as the performed simulation steps. The guard is a Boolean expression with (in)equalities of such arithmetic expressions.

```
1 begin analysis
2  query = eval when { AttackAttempts == 1 } :
3   { OpenSafe, BruteForce, OpenLock, Guess-
4     PIN, LearnPIN, FindOwner, ForceOwner,
5     LaserCutter, steps[delta = 0.5] }
6  default alpha = 0.05 delta = 0.1
7   parallelism = 1
8 end analysis
```

Code 11: First kind of analysis supported

MultiVeStA estimates the expected value $E[x]$ of each property as the mean $\overline{x}$ of n independent probabilistic simulation runs, with n large enough (but minimal) to satisfy an (α, δ) *confidence interval* (CI): $E[x]$ belongs to $[\overline{x} - \delta/2, \overline{x} + \delta/2]$ with statistical confidence $(1 - \alpha) \cdot 100\%$. The CI is specified with `default alpha` and `delta` (but a property-specific δ can be used, as for `steps`). The keyword `parallelism` specifies the number of local processes to be launched to distribute the simulation runs.

Table 1 shows the results for the analysis of first attack. The probability of achieving `OpenSafe`, `OpenLock`, or `LearnPIN` on first attempt is 0. This is due to the hierarchical constraints from the attack-defense diagram (e.g. accomplishing `LearnPIN` requires accomplishing `FindOwner` and `ForceOwner`). The probability of accomplishing `BruteForce` or `GuessPIN` is low because the attacker is more likely to try the strategy for `LearnPIN`. Also, `BruteForce` attempts are likely to fail due to the high defense effectiveness of `Reinforced`. `FindOwner`, part of the initial configuration (cf. DSL Code 10), has probability 1. On average, 3 steps are necessary to first attempt an attack. This is consistent with DSL Code 10: first a strategy is chosen (Lines 9-11), then `try` is executed towards an intermediate state (e.g. Line 13) where an attack is attempted (e.g. Lines 14-15). The analysis required 400 simulation runs and ran in 1.22 seconds

Next, we analyse properties over time. In DSL Code 12, we compute the probability for each attack that it is successfully attempted and the probability of activating the two defensive nodes, while we are no longer interested in the average `steps` (cf. Lines 3-5). As shown in Line 2, this requires to change the condition `when` with the steps of interest: `from 1 to 50`, increasing `by 1`. As a result, we are evaluating 50×10 properties. The analysis required 560 simulation runs and ran in 19 seconds.

```
1 begin analysis
2  query = eval from 1 to 50 by 1 :
3   { OpenSafe, BruteForce, OpenLock, Guess-
4     PIN, LearnPIN, FindOwner, ForceOwner,
5     LaserCutter, Lockdown, Reinforced }
6  default alpha = 0.05 delta = 0.1
7   parallelism = 1
8 end analysis
```

Code 12: Second kind of analysis supported

Figure 3 depicts the probability over time of successful attacks. The probability of `OpenSafe` grows, until approaching 1 at step 30. By that point in time, all probabilities have stabilized and hence we truncated the plots. Note that `GuessPIN` has the lowest probability, closely followed by `BruteForce` and `LaserCutter`, while the other probabilities stabilize close to 0.8.

Figure 4 shows that the probabilities of disabling `Reinforced` and activating `Lockdown` are low. This is because attacks affecting both of these defensive nodes are performed with a low probability.

In the future, we intend to investigate and integrate additional analysis capabilities. We also plan to increase the tool's usability by providing a collection of attack libraries and import and export support for existing formats, considering at least the model-driven engineering approach of the generic ATTop tool [24].

3 VISION AND ROADMAP

We conclude by presenting several parts of our approach for which we envision possibilities for extensions and improvements and possible means to achieve them.

One part concerns the model itself. Currently, our framework considers a dynamic model of attack behavior and a structural model of available attack and defense features. First, let us look at the attack/defense model. The model is very useful to describe correlations between goals, such as "*if a goal A is accomplished, then subsequently a goal B must be accomplished*". This correlation of goals is viewed by [3] as a 'correlation of steps'. We could make this more explicit in our DSL by adding temporal steps in the feature description (like $\Diamond$ or $\Box$, inspired by temporal logic [4, 13]).

Furthermore, so far we can make hypotheses concerning the presence or absence of attack and defense attributes in a system, but it would be interesting to add further variability explicitly in the DSL. The goal would be to be able to distinguish products (system variants or scenarios) *with* from products *without* those specific attack or defense attributes and, moreover, to provide a temporal correlation among those attributes. In the end, this would allow reasoning on presence/absence, quantities, and temporalities, which could be described in a domain-specific logic. This would require us to distinguish between action constraints and attribute constraints (and define the latter explicitly as language constructs in the DSL).

Concerning temporalities we could consider introducing temporalities between attack and defense steps. This would allow us to describe fine-grained correlations such as "*if attack step (or, in fact, any step) A is performed, then defense step B is performed in less than X units of time*". In such case, X could even be parameterized per product (system variant, i.e. customized security scenario). This would lead to a notion of 'reactive features'.

Another aspect that would be interesting to extend is the behavioral attack model. Our current model is purely probabilistic. It would be worthwhile to investigate the introduction of non-determinism aspects typical of Markov decision processes. The non-deterministic transitions could be labeled with Boolean formulae, similar to the feature expressions that label the transitions of featured transition systems [14, 15]. This would allows us to consider different kind of problems such as that of synthesizing the best attacker or to formulate statements like "*all attackers with feature A satisfy a given property*". The latter could be achieved by combining our approach with Uppaal Stratego [16].

Regarding requirements we believe it should be possible to consider the full linear temporal logic LTL. Technically, this can be done by combining our behavioral models with Büchi automata and look for lassos (cf. [18]). This would allow us to express more generic properties such as "*if a person has the authorization to enter and wants to enter, then (s)he will eventually enter*".

Finally, most of the properties we have checked with our framework so far are with respect to logical requirements. Recently, statistical model checking has also been used to compare the behavior of two systems via simulation [25]. We could imagine to compare the behavior of two attackers via simulation, or the effect of two attackers on two different attack-defense diagrams. It would also be interesting to see whether we can derive the attackers with the best chance of success (e.g. leave some of the weights off the edges and rather derive them).

Table 1: Results for the analysis of first attack (cf. DSL Code 11)

OpenSafe	BruteForce	OpenLock	GuessPIN	LearnPIN	FindOwner	ForceOwner	LaserCutter	steps
0	0.013	0	0.013	0	1	0.27	0.056	3

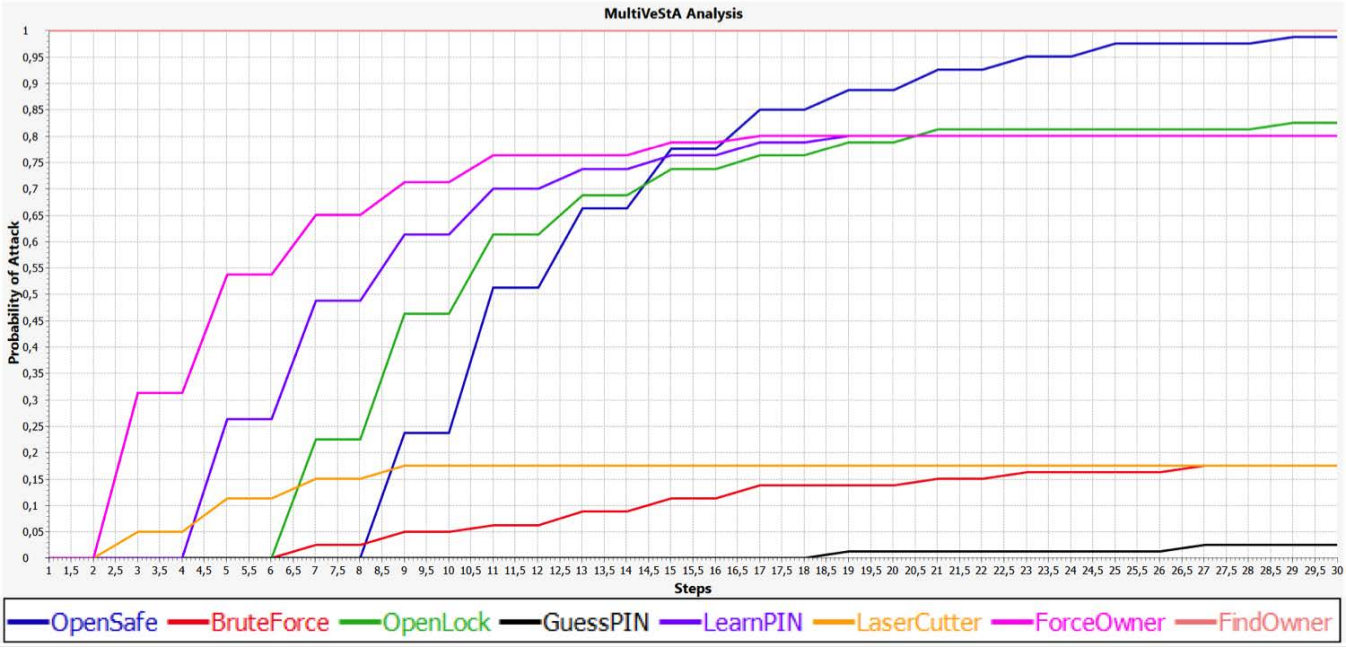

Figure 3: Probability of successful attacks (over time, cf. DSL Code 12)

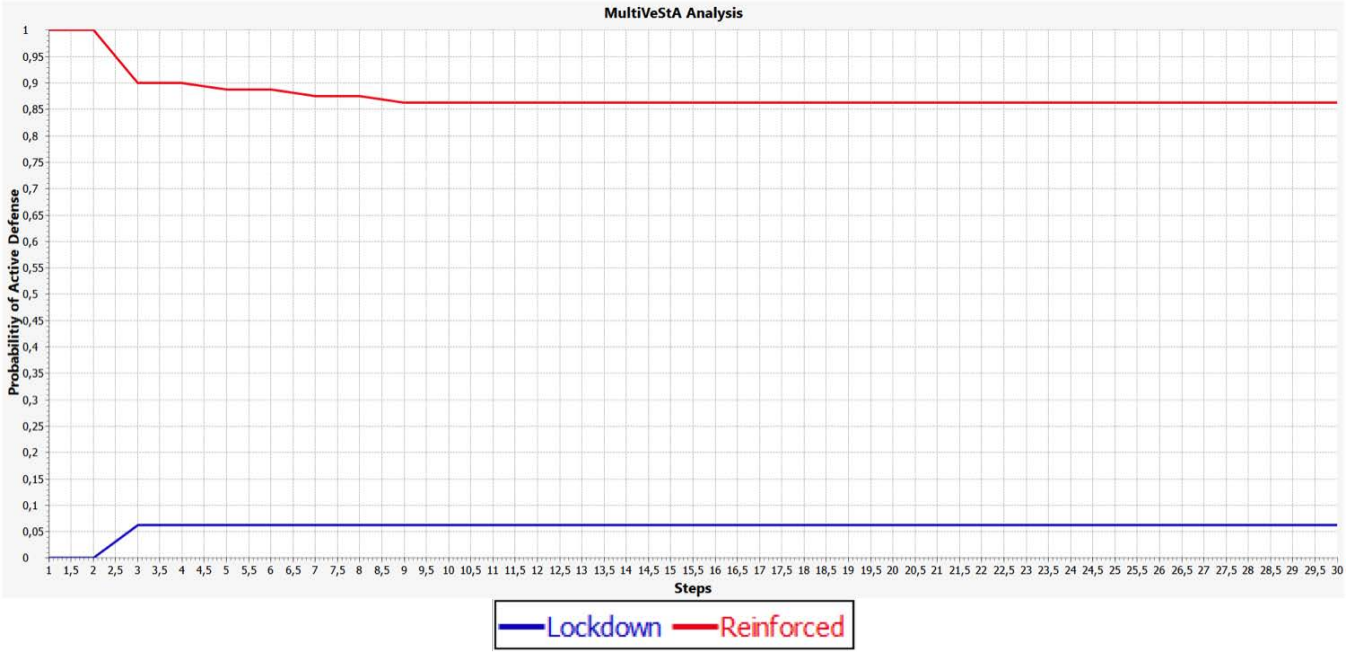

Figure 4: Probability of active defensive nodes (over time, cf. DSL Code 12)

ACKNOWLEDGMENTS

Work partially supported by the EU H2020-SU-ICT-03-2018 Project No. 830929 CyberSec4Europe and by the MIUR PRIN 2017FTXR7S project IT MaTTerS (Methods and Tools for Trustworthy Smart Systems).

The example in the paper was worked out by Christian Bach and Christian Toftemann Bæk for their M.Sc. thesis [11].

All experiments were run on a Mac with a 2.4 GHz Intel Core i5 and 4 GB of RAM.

REFERENCES

[1] Gul Agha and Karl Palmskog. 2018. A Survey of Statistical Model Checking. *ACM Trans. Model. Comp. Simul.* 28, 1 (2018), 6:1–6:39. https://doi.org/10.1145/3158668

[2] Amenaza 2006. *The SecuITree® BurgleHouse Tutorial (a.k.a., Who wants to be a Cat Burglar?).* Amenaza. https://www.amenaza.com/downloads/docs/Tutorial.pdf

[3] Zaruhi Aslanyan, Flemming Nielson, and David Parker. 2016. Quantitative Verification and Synthesis of Attack-Defence Scenarios. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF'16)*. IEEE, 105–119. https://doi.org/10.1109/CSF.2016.15

[4] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. The MIT Press. http://mitpress.mit.edu/books/principles-model-checking

[5] Maurice H. ter Beek and Axel Legay. 2019. Quantitative Variability Modeling and Analysis. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'19)*, Gilles Perrouin and Danny Weyns (Eds.). ACM, 13:1–13:2. https://doi.org/10.1145/3302333.3302349

[6] Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2015. Quantitative Analysis of Probabilistic Models of Software Product Lines with Statistical Model Checking. *Electr. Proc. Theor. Comput. Sci.* 182 (2015), 56–70. https://doi.org/10.4204/EPTCS.182.5

[7] Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2015. Statistical Analysis of Probabilistic Models of Software Product Lines with Quantitative Constraints. In *Proceedings of the 19th International Software Product Line Conference (SPLC'15)*. ACM, 11–15. https://doi.org/10.1145/2791060.2791087

[8] Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2016. Statistical Model Checking for Product Lines. In *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA'16) (LNCS)*, Tiziana Margaria and Bernhard Steffen (Eds.), Vol. 9952. Springer, 114–133. https://doi.org/10.1007/978-3-319-47166-2_8

[9] Maurice H. ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. 2018. A framework for quantitative modeling and analysis of highly (re)configurable systems. *IEEE Trans. Softw. Eng.* (2018). https://doi.org/10.1109/TSE.2018.2853726

[10] Maurice H. ter Beek, Alberto Lluch Lafuente, and Marinella Petrocchi. 2013. Combining Declarative and Procedural Views in the Specification and Analysis of Product Families. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, Vol. 2. ACM, 10–17. https://doi.org/10.1145/2499777.2500722

[11] Christian Toftemann Bæk and Christian Bach. 2018. *A Framework for Quantitative Security Modeling and Analysis of Customizable Attack Scenarios.* Master's thesis. Technical University of Denmark. https://findit.dtu.dk/en/catalog/2436333772

[12] Seyit Ahmet Çamtepe and Bülent Yener. 2007. Modeling and detection of complex attacks. In *Proceedings of the 3rd International Conference on Security and Privacy in Communications Networks (SecureComm'07)*. IEEE, 234–243.

[13] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8

[14] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Softw. Eng.* 39, 8 (2013), 1069–1089. https://doi.org/10.1109/TSE.2012.86

[15] Maxime Cordy, Xavier Devroey, Axel Legay, Gilles Perrouin, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Jean-François Raskin. 2019. A Decade of Featured Transition Systems. In *From Software Engineering to Formal Methods and Tools, and Back*, Maurice H. ter Beek, Alessandro Fantechi, and Laura Semini (Eds.). LNCS, Vol. 11865. Springer, 285–312. https://doi.org/10.1007/978-3-030-30985-5_18

[16] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. 2015. Uppaal Stratego. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15) (LNCS)*, Christel Baier and Cesare Tinelli (Eds.), Vol. 9035. Springer, 206–211. https://doi.org/10.1007/978-3-662-46681-0_16

[17] Nicolas David, Alexandre David, Rene Rydhof Hansen, Kim G. Larsen, Axel Legay, Mads Chr. Olesen, and Christian W. Probst. 2015. Modelling Social-Technical Attacks with Timed Automata. In *Proceedings of the 7th ACM CCS International Workshop on Managing Insider Security Threats (MIST'15)*. ACM, 21–28. https://doi.org/10.1145/2808783.2808787

[18] Radu Grosu and Scott A. Smolka. 2005. Monte Carlo Model Checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05) (LNCS)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.), Vol. 3440. Springer, 271–286. https://doi.org/10.1007/978-3-540-31980-1_18

[19] Holger Hermanns, Julia Krämer, Jan Krcál, and Mariëlle Stoelinga. 2016. The Value of Attack-Defence Diagrams. In *Proceedings of the 5th International Conference on Principles of Security and Trust (POST'16) (LNCS)*, Frank Piessens and Luca Viganò (Eds.), Vol. 9635. Springer, 163–185. https://doi.org/10.1007/978-3-662-49635-0_9

[20] Barbara Kordy, Piotr Kordy, Sjouke Mauw, and Patrick Schweitzer. 2013. ADTool: Security Analysis with Attack-Defense Trees (Extended Version). *CoRR* abs/1305.6829 (2013). arXiv:1305.6829 http://arxiv.org/abs/1305.6829

[21] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. 2011. Foundations of Attack-Defense Trees. In *Proceedings of the 7th International Workshop on Formal Aspects in Security and Trust (FAST'10) (LNCS)*, Pierpaolo Degano, Sandro Etalle, and Joshua Guttman (Eds.), Vol. 6561. Springer, 80–95. https://doi.org/10.1007/978-3-642-19751-2_6

[22] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. 2014. DAG-based attack and defense modeling: Don't miss the forest for the attack trees. *Comput. Sci. Rev.* 13–14 (2014), 1–38. https://doi.org/10.1016/j.cosrev.2014.07.001

[23] Rajesh Kumar, Enno Ruijters, and Mariëlle Stoelinga. 2015. Quantitative Attack Tree Analysis via Priced Timed Automata. In *Proceedings of the 13th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'15) (LNCS)*, Sriram Sankaranarayanan and Enrico Vicario (Eds.), Vol. 9268. Springer, 156–171. https://doi.org/10.1007/978-3-319-22975-1_11

[24] Rajesh Kumar, Stefano Schivo, Enno Ruijters, Bugra Mehmet Yildiz, David Huistra, Jacco Brandt, Arend Rensink, and Mariëlle Stoelinga. 2018. Effective Analysis of Attack Trees: A Model-Driven Approach. In *Proceedings of the 21st International Conference on Fundamental Approaches to Software Engineering (FASE'18) (LNCS)*, Alessandra Russo and Andy Schürr (Eds.), Vol. 10802. Springer, 56–73. https://doi.org/10.1007/978-3-319-89363-1_4

[25] Kim G. Larsen, Axel Legay, Marius Mikučionis, Brian Nielsen, and Ulrik Nyman. 2017. Compositional Testing of Real-Time Systems. In *ModelEd, TestEd, TrustEd*, Joost-Pieter Katoen, Rom Langerak, and Arend Rensink (Eds.). LNCS, Vol. 10500. Springer, 107–124. https://doi.org/10.1007/978-3-319-68270-9_6

[26] Axel Legay, Anna Lukina, Louis-Marie Traonouez, Junxing Yang, Scott A. Smolka, and Radu Grosu. 2019. Statistical Model Checking. In *Computing and Software Science: State of the Art and Perspectives*, Bernhard Steffen and Gerhard J. Woeginger (Eds.). LNCS, Vol. 10000. Springer, 478–504. https://doi.org/10.1007/978-3-319-91908-9_23

[27] Axel Legay and Gilles Perrouin. 2017. On Quantitative Requirements for Product Lines. In *Proceedings of the 11th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'17)*, Maurice H. ter Beek, Norbert Siegmund, and Ina Schaefer (Eds.). ACM, 2–4. https://doi.org/10.1145/3023956.3023970

[28] Aleksandr Lenin, Jan Willemson, and Dyan Permata Sari. 2014. Attacker Profiling in Quantitative Security Assessment Based on Attack Trees. In *Proceedings of the 19th Nordic Conference on Secure IT Systems (NordSec'14) (LNCS)*, Karin Bernsmed and Simone Fischer-Hübner (Eds.), Vol. 8788. Springer, 199–212. https://doi.org/10.1007/978-3-319-11599-3_12

[29] Sjouke Mauw and Martijn Oostdijk. 2005. Foundations of Attack Trees. In *Proceedings of the 8th International Conference on Information Security and Cryptology (ICISC'05) (LNCS)*, Dong Ho Won and Seungjoo Kim (Eds.), Vol. 3935. Springer, 186–198. https://doi.org/10.1007/11734727_17

[30] Wen ping Lv and Wei min Li. 2011. Space Based Information System Security Risk Evaluation Based on Improved Attack Trees. In *Proceedings of the 3rd International Conference on Multimedia Information Networking and Security (MINES'11)*. IEEE, 480–483. https://doi.org/10.1109/MINES.2011.94

[31] Sheldon M. Ross. 2001. *Probability Models for Computer Science.* Academic Press.

[32] Arpan Roy, Dong Seong Kim, and Kishor S. Trivedi. 2012. Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees. *Secur. Commun. Netw.* 5, 8 (2012), 929–943. https://doi.org/10.1002/sec.299

[33] Bruce Schneier. 1999. Attack Trees. *Dr. Dobb's Journal* (1999). https://www.schneier.com/academic/archives/1999/12/attack_trees.html

[34] Stefano Sebastio and Andrea Vandin. 2013. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. In *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools (ValueTools'13)*. ACM, 310–315. https://doi.org/10.4108/icst.valuetools.2013.254377

[35] Andrea Vandin, Maurice H. ter Beek, Axel Legay, and Alberto Lluch Lafuente. 2018. QFLan: A Tool for the Quantitative Analysis of Highly Reconfigurable Systems. In *Proceedings of the 22nd International Symposium on Formal Methods (FM'18) (LNCS)*, Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink (Eds.), Vol. 10951. Springer, 329–337. https://doi.org/10.1007/978-3-319-95582-7_19