Quantitative threat analysis via a logical service [1]

Michael Huth and Jim Huan-Pu Kuo
Department of Computing, Imperial College London
London, SW7 2AZ, United Kingdom
{m.huth, jimhkuo}@imperial.ac.uk

29 September 2014

Abstract

It is increasingly important to analyze system security quantitatively using concepts such as trust, reputation, cost, and risk. This requires a thorough understanding of how such concepts should interact so that we can validate the assessment of threats, the choice of adopted risk management, etc.. To this end, we propose a declarative language *Peal+* in which the interaction of such concepts can be rigorously described and analyzed. *Peal+* has been implemented in *PEALT* using the SMT solver Z3 as analysis back-end. *PEALT*'s code generators target complex back-ends and evolve with optimizations or new back-ends. Thus we can neither trust the tool chain nor feasibly prove correctness of all involved artefacts. We eliminate the need to trust that tool chain by independently certifying scenarios found by back-ends in a manner agnostic of code generation and choice of back-end. This scenario validation is compositional, courtesy of Kleene's 3-valued logic and potential refinement of scenarios. We prove the correctness of this validation, discuss how *PEALT* presents scenarios to further users' understanding, and demonstrate the utility of this approach by showing how it can express attack-countermeasure trees so that the interaction of attack success probability, attack cost, and attack impact can be analyzed.

1 Introduction

It is well recognized that the analysis of threats to system security goes beyond the exposure and fix of vulnerabilities and that it also has to take account of contextual influences such as risks, trust assumptions, the reputation of domains, etc. However, it is often not clear how such different concepts interact in the threat space (which the attacker controls) or how they should interact in a system design space (which the designer thinks he controls). For example, when the Heartbleed vulnerability became known even security experts could not uniformly agree on whether users should immediately change their passwords on web accounts that used versions of OpenSSL vulnerable to this attack [19]: e.g., it was difficult to know whether the account was compromised, and renewing a password in a compromised account might leak that password to an attacker.

In general, threat analysts have a host of techniques and models at their disposal that allow them to assess security threats, let us mention here attack trees [18, 13, 12] and Stackelberg games for security (see e.g. [10]) as two prominent examples. Also, probabilistic risk analysis [2] offers a rich set of tools that threat analysts may use to study the interaction of factors that influence security. Alas, tools from risk analysis view attackers as passive environments (e.g. a random process modeling life expectancy of a light bulb) and not as active agents (e.g. a cyber

[1] Please cite this research note as *Michael Huth and Jim Kuo. Quantitative threat analysis via a logical service: usability, service certification, and case study. Technical Report 2014/10, Department of Computing, Imperial College London, ISSN 1469-4174, 2014.*

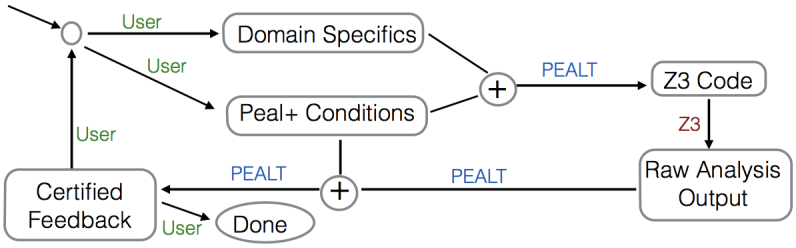

Figure 1: Overview of *PEALT* and our contributions: extension of *PEALT* to richer input language *Peal+*, certification of scenarios, and computation of usable feedback

terrorist who seeks access to programmable logic controllers in a SCADA system). We therefore would like support for modeling the interaction of concepts such as protection cost, impact of successful attacks, perception of risk, reputation of agents, etc., in a system exposed to active attacks. The active nature of attackers suggests to model action and reaction with AND/OR structures, e.g. as present in two-person games or first-order logic. The desire to study interaction of quantitative concepts suggests use of an expressive logical language with appropriate theories for reals; expressiveness means we can easily extend studies to new concepts or interaction modes, and theories enable us to do correct quantitative reasoning. We cannot assume, though, that threat analysts are trained logicians, so we require automated reasoning support for such logics to build *auto-interactive verifiers*. SMT solvers, e.g. Z3 [14], thus look like apt vehicles for expressing and analyzing such interaction in this manner.

Chosing an SMT solver as back-end also poses problems. Its input language is too complex and universal, since security analysts prefer languages specific to their modeling domain. For code generators from domain-specific languages into SMT back-ends we need assurance that results computed by back-ends are correct and sensible in the modeled domain. Security analysts want results communicated in forms appreciable to them. Finally, users may formulate conditions that are vacuously true, or vacuously false in the modeled domain. This may identify a specification error or it may instead validate that an analyst has realized an important invariant – e.g. that the risk is always below an acceptable threshold. Our paper presents results that directly address this set of problems.

Figure 1 shows how our contributions presented in this paper are realized in the tool *PEALT*. Users specify *Peal+* conditions to be analyzed, and domain-specific knowledge or assumptions; *PEALT* converts these specifications into Z3 code which the SMT solver Z3 solves; the raw output of Z3 results is then post-processed and analyzed over the *Peal+* conditions; and feedback is reported so that all scenarios are certified. The user may then inspect that feedback and either be satisfied or edit conditions or domain specifics for further analysis.

Outline of paper. In Section 2, we present language *Peal+* in which threats can be modeled and analyzed when quantitative information contains non-deterministic uncertainty; and we discuss automated vacuity checking. In Section 3, we present an algorithm which independently certifies that scenarios computed from analyses by back-ends such as Z3 are correct for the modeled problem – eliminating the need to trust our code generation methods or back-ends. In Section 4, we discuss how the implementation of *Peal+*, its analyses, and certification are supported by the use of partial evaluation to render certified scenarios to users in compact ways that should facilitate users' comprehension. In Section 5, we show *Peal+*'s utility as intermediate language for

$$
\begin{array}{rcll}
\mathcal{E}_e(op(pS_1, pS_2)) & = & op(\mathcal{E}_e(pS_1), \mathcal{E}_e(pS_2)) & \\
\mathcal{E}_e(op((q_1\ s_1)\ \ldots (q_n\ s_n))\ \mathit{default}\ s) & = & op(Z) & (\text{if } Z \neq \emptyset) \\
\mathcal{E}_e(op((q_1\ s_1)\ \ldots (q_n\ s_n))\ \mathit{default}\ s) & = & \mathcal{E}_e(s) & (\text{if } Z = \emptyset)
\end{array}
$$

$$
\begin{array}{rcll}
\mathcal{E}_e(a) & = & a & (\text{constant } a) \\
\mathcal{E}_e(x) & = & e(x) & (x \text{ not of form } p.\mathbf{sc}) \\
\mathcal{E}_e(a * x) & = & a \cdot e(x) & (x \text{ not of form } p.\mathbf{sc}) \\
\mathcal{E}_e(p.\mathbf{sc}) & = & \mathcal{E}_e(p) & (\text{evaluate policy } p) \\
\mathcal{E}_e(a * p.\mathbf{sc}) & = & a \cdot \mathcal{E}_e(p) & (\text{evaluate policy } p) \\
\mathcal{E}_e(r\ [l,u]) & = & \mathcal{E}_e(r) + e(p, q_i, [l,u]) & ([l,u] \text{ declared in } p \text{ for predicate } q_i) \\
\mathcal{E}_e(r\ [l,u]) & = & \mathcal{E}_e(r) + e(p, \mathit{default}, [l,u]) & ([l,u] \text{ declared in default score } s \text{ of } p)
\end{array}
$$

Figure 2: Semantics $\mathcal{E}_e(pSet)$ of policy sets (acyclic as in Def. 1), given an environment e that maps predicates to truth values, scores to reals, and resolves non-deterministic choice of uncertainty intervals. Scores r range over raw scores, a over constants, x over variables, and $p.\mathbf{sc}$ over policy scores. Set Z equals $\{\mathcal{E}_e(s_i) \mid 1 \leq i \leq n, e(q_i) = true\}$

analyzing the interaction of threat concepts in tree-like models. In Section 6, we further discuss and evaluate language *Peal+* and its implementation in tool *PEALT*. Section 7 features related work, and Section 8 concludes the paper.

2 Language and its vacuity checks

Figure 3 shows a formal grammar for our language *Peal+* that can express interaction of security aspects as well as the logical/quantitative analysis of such interaction. *Peal+* shares the coarse structure of its predecessor *Peal* [4, 7]: rules condition a score on a predicate, policies are built from rules, policy sets are built out of policies, conditions are formed out of policy set comparisons; and analyses have conditions as arguments. The meaning of analysis types is the intuitive one of their names seen in Figure 3. The meaning of conditions is given by that of propositional logic and of comparison operators over reals. Thus it suffices to define how policy sets evaluate to reals in an environment in which all predicates have truth values, all real variables have a real value, and all non-deterministic uncertainties are resolved – so all scores evaluate to a real number.

We state this semantics informally here, and formally in Figure 2. A rule *rule* returns its declared score when its declared predicate is true; otherwise, it has no effect. The meaning of a policy is then given as follows: if none of its rules has a true predicate, its meaning is that of its default score; otherwise, its meaning is obtained by first computing the meaning of all scores from its rules with true predicates, and then applying operator *op* to that set of computed reals.

The grammar for scores allows us to write expressions such as 0.45, -124.5, $0.67 * x$, $0.5 * p.\mathbf{sc}$, $0.4 * x\ [-0.1, 0.1]$, or $0.5 * p.\mathbf{sc}\ [-0.05, 0.05]$ where x is a real-valued identifier and p is a policy. In a given environment, the meaning of scores without intervals $[l, u]$ is that of normal arithmetic with variable values given by the environment. The meaning of expressions $s\ [l, u]$ is $x + y$ where

$$
\begin{array}{rcl}
alys & ::= & \textbf{always\_true}?\ cond \mid \textbf{equivalent}?\ cond\ cond \mid \textbf{different}?\ cond\ cond \mid \\
 & & \textbf{implies}?\ cond\ cond \mid \textbf{always\_false}?\ cond \mid \textbf{satisfiable}?\ cond \\
cond & ::= & q \mid \neg cond \mid cond \,||\, cond \mid cond \,\&\&\, cond \mid pSet \leq pSet \mid pSet < pSet \\
 & & \\
op & ::= & min \mid max \mid + \mid * \\
pSet & ::= & pol \mid op\ (pSet,\ pSet) \\
pol & ::= & op\,(rule^*)\ \textbf{default}\ score \\
rule & ::= & (q\ score) \\
 & & \\
score & ::= & rawScore \mid rawScore\ [realConst\,,\ realConst] \\
rawScore & ::= & realConst \mid realVar \mid realConst * realVar \\
realVar & ::= & identifier \mid pSet.\textbf{sc}
\end{array}
$$

Figure 3: Syntax of *Peal+* where q ranges over some language of predicates, constants and variables occurring in *score* expressions range over real numbers, and $[realConst\,,\ realConst]$ ranges over closed real intervals. For sake of clarity, keywords of *Peal+* are written in boldface here, e.g., $pSet.\textbf{sc}$ denotes the score of $pSet$

x is the meaning of s in the environment, and y from $[l, u]$ is the non-deterministic choice of the environment from interval $[l, u]$. To ensure consistency, we require $l \leq 0.0 \leq u$. For example, $u < l$ would be logically inconsistent and $l > 0$ would suggest to change $s\ [l, u]$ to the equivalent but more comprehensible $(s + l)\ [0.0, u - l]$ when $l \leq u$. The meaning of variable $pSet.\textbf{sc}$ is that of $pSet$ computed by the operational semantics just described. For this to be well-defined, the set of declared policy sets must not create cyclic dependencies in *Peal+*:

Definition 1 *Let p_1 and p_2 be in a set $\mathcal{P}$ of Peal+ policy sets. Then p_1* depends on *p_2 (written $p_2 \prec p_1$) if there is a score s in p_1 that contains or equals variable $p_2.\textbf{sc}$. Set $\mathcal{P}$ is* acyclic *if the transitive closure of $\prec$ over $\mathcal{P} \times \mathcal{P}$ is acyclic.*

Peal+ extends *Peal* is important ways: scores may have variables and non-deterministic uncertainty, policy sets have the same composition operators as policies, conditions subsume propositional logic and may compare policy sets, and the result of a policy set can be referred to as variable within a score expression. With these extensions, *Peal+* is expressive enough to capture metrics, tree-like models, cost functions, and basic probabilistic computations.

Let us illustrate the use of *Peal+* with an example modeling risks that a car rental company may face when renting out cars to clients. Figure 4 shows how rules, policies, policy sets, and conditions for this example are declared in the input language of our tool *PEALT*. Declarations are divided into blocks by keywords such as `POLICIES` and lines that begin with % are used for comments.

A notable feature of the tool input language is the declaration block `DOMAIN_SPECIFICS` in which specifiers can enter code from the input language of the SMT solver Z3 [14] to further constrain the model. This would typically be used to express assumptions or knowledge of the

modeled domain, and uses Z3 syntax since Z3 is the current back-end of our tool. For example, the model in Figure 4 uses this to express that luxury cars must not be rented out for off-road driving. It represents risk and trust as values in $[0, 1]$, and uses $f(x) = 1 - x$ to convert one into the other. More sophisticated relationships between trust and risk may be captured in *Peal+* as well. This *Peal+* model is conceptually similar to the use of score cards that assess risks in mortgage applications [16]. Next, we discuss vacuity checking and how our analyses support this.

Vacuity checking. The analyses `always_true`? and `always_false`? reduce to satisfiability checks but their intent is to check for so called vacuities [11]: a condition that is always true or always false may be a specification error (as in temporal logic verification of hardware [11]), evidence for a desired invariant or may desire further scrutiny of the specifier. Our tool automatically enforces both types of vacuity check on *all* declared conditions. The reason is that declared conditions are likely to contribute to input of a declared analysis, and so we want to alert users to those conditions that are vacuously true, or vacuously false.

For example, condition `c1` of the Car Rental Risks example in Figure 4 is reported to be always true, so the "insurance risk" which multiplies monetary loss with its associates risk is never above 50,000. If Z3 can't decide a vacuity check (output `UNKNOWN`), *PEALT* reports checked conditions as "may be" vacuities. *PEALT* only reports names of vacuously true or false conditions. Users who want more detailed feedback as described below need to "promote" such a vacuity analysis into the ANALYSES section, where more detailed feedback is provided. Users may turn automated vacuity checking on or off under "Settings". We recommend vacuity checks to be done at least once for model validation.

3 Scenario certification

Users from high assurance domains need compelling evidence that scenarios computed by back-ends from code *PEALT* generates are valid for analyzed *Peal+* conditions, and they want to be able to relate scenarios to conditions in a comprehensible manner. We report additional support for the latter below. As for the former, what if our Z3 code generation method contains logical mistakes? What if we make wrong assumptions about the operation of the tool Z3? What if some Z3 features we use contain implementation flaws? What if (perish the thought) the inference techniques used in Z3 were to contain mistakes? We think these questions make a compelling case for independently certifying the validity of a scenario discovered for a *Peal+* condition. Back-ends such as Z3 compute scenarios that are very compact in that they don't define values for some variables. Certification therefore needs to be able to reason that these are indeed "don't care" variables, making certification non-trivial.

Such a certification should be comprehensible to non-experts and efficient – giving it the flavor of an NP problem although the underlying decision problems may be undecidable. Our compositional certification of don't care variables may lose precision and so may have an inconclusive output. In the latter case, one of the predicates of the scenario may not have a specified truth value. We then set that value to *false* and repeat the certification algorithm on this refined scenario. This process is efficient as it examines conditions compositionally and greedily refines scenarios until it succeeds or not. Refined predicates are set to *false* and *not* to *true*: users want to see as few trees in the forest as possible, and false predicates only have an effect in a policy when all its predicates are false.

This certification process represents a scenario, a model returned by Z3, as a function I that

```
POLICIES
% policy capturing risk of financial loss dependent on type of rented car
b1 = max ((isLuxuryCar 150000) (isSedan 60000) (isCompact 30000)) default 50000
% policy capturing trust in rentee dependent on type of his or her driving license
b2 = min ((hasUSLicense 0.9) (hasUKLicense 0.6) (hasEULicense 0.7)
          (hasOtherLicense 0.4 [-0.1,0.1])) default 0
% policy that captures potential risk dependent on type of intended car usage
% this policy happens not to be used in the conditions below
b3 = max ((someOffRoadDriving 0.8) (onlyCityUsage 0.4) (onlyLongDistanceUsage 0.2)
          (mixedUsage 0.25)) default 0.3
% policy that accumulates some signals that may serve as additional trust indicators
b4 = + ((accidentFreeForYears 0.05*x) (speaksEnglish 0.05) (travelsAlone -0.2)
        (femaleDriver 0.1)) default 0
% convert trust b2 into risk b2 using f(x) = 1-x
b2_risk = +((True 1.0) (True -1*b2_score)) default 0.0
POLICY_SETS
% casting b2_risk into policy set
pSet0 = b2_risk
% policy set that multiplies risk with potential financial loss
pSet1 = *(b1,pSet0)
% casting policy p4 into a policy set
pSet_b4 = b4
CONDITIONS
% condition that the risk aware potential financial loss is below a certain bound
c1 = pSet1 <= 50000
% condition that the accumulated trust is above a certain threshold
c2 = 0.4 < pSet_b4
% condition that insists that two previous conditions have to hold
c3 = c1 && c2
DOMAIN_SPECIFICS
% real x models accident-free years of driving, 'truncated' at value 10
(assert (and (<= 0 x) (<= x 10)))
% capturing a company policy: luxury cars must not be used for off road driving
(assert (or (not isLuxuryCar) (not someOffRoadDriving)))
% capturing that the different types of rental cars are mutually exclusive
(assert (and (implies isLuxuryCar (and (not isSedan) (not isCompact)))
             (implies isSedan (and (not isLuxuryCar) (not isCompact)))
             (implies isCompact (and (not isSedan) (not isLuxuryCar)))))
% capturing that cars that are only used in cities are not used in a mixed sense
(assert (implies onlyCityUsage (not mixedUsage)))
% capturing that cars used only for longdistance driving are not used in a mixed sense
(assert (implies onlyLongDistanceUsage (not mixedUsage)))
% capturing domain constraints (or company policy?) that city driving cannot happen off road
(assert (implies onlyCityUsage (not someOffRoadDriving)))
% capturing that cars used only for longdistance driving must drive off road
(assert (implies onlyLongDistanceUsage (not someOffRoadDriving)))
ANALYSES
% is condition c1 always true? this would suggest an invariant
name1 = always_true? c1
% is condition c3 always true? this would suggest a specification error
name2 = always_true? c3
```


Figure 4: *Peal+* model of Car Rental Risks

maps real variables to real numbers or $\perp$, and predicates to *true*, *false* or $\perp$. Symbol $\perp$ models that the scenario did not specify a value for the variable in question. For predicates, $\perp$ ("unknown") is also the third truth value of Kleene's 3-valued logic [9]. Figure 8 shows how *PEALT* reports a scenario for analysis `name2` from Figure 4. To explain our certification, we need to define the refinement of environments, which are all well typed in that they map any variable either to value $\perp$ or a value of its declared data type – Real or Boolean.

Definition 2 *Let env_1, env_2 be environments over the same set of variables $\mathcal{V}$. Then env_2* refines *env_1 if for all x in $\mathcal{V}$, $env_1(x) \neq \perp$ implies $env_1(x) = env_2(x)$.*

This means that refinements can change $\perp$ values of variables to any value of their declared data type, but they cannot change non $\perp$ values.

Function $recursivelyCertify(c, I, v, \emptyset)$, seen in Figure 5, checks whether condition c has truth value v in the scenario/Z3 model I. It outputs *true* if this claim could be certified, *false* if a *logical* flaw in the claim was detected, and outputs $\perp$ otherwise. Wrapper function $certifyWrapper(c, I, v)$ in Figure 5 converts *true*, *false* and $\perp$ into certification *success*, *failure*, and *inconclusive*, respectively.

The truth value v used in $recursivelyCertify(c, I, v, \emptyset)$ is determined by the type of analysis. For example, if `always_false? c` returns `SAT`, it means the found scenario should be evidence for c being true, and so v equals *true*. The treatment of analyses with two arguments is similar. For example, for a `SAT` outcome of `implies? c1 c2`, the scenario should be evidence for c_1 being true and c_2 being false. So we need to achieve two certifications, $recursivelyCertify(c_1, I, true, \emptyset)$ and $recursivelyCertify(c_2, I, false, \emptyset)$ for this.

Function *recursivelyCertify* refines I into an environment env' by setting predicates to *false* or adding a statically inferred score to a policy. The latter means that environments are not only defined on predicates and real variables but may also map policy names to their inferred scores. At program point l2, such static inference of policy scores is delegated to function *collectCertifiablePolicyScores* in Figure 6. In this extended environment env', function *certCond*, shown in Figure 7, determines the truth value of the condition in that environment under Kleene's 3-valued logic [9]. If that value is $\perp$, we call *recursivelyCertify* again but with a refined environment that either inferred at least one new policy score or set a predicate to false. If the truth value of the condition is $\neq \perp$, function *recursivelyCertify* outputs that value.

The need for parameter *cp* and for checking its "progress" comes from construct *pol*.**sc**, where static inference of a policy score may then enable more such inferences for other policies. Function $collectCertifiablePolicyScores(env)$ initializes in *cp* an empty hash map. For each declared policy *pol* it stores in *score* the output of function $certPolicy(pol, env)$ depicted in Figure 6. Thus we statically infer the score of *pol* (rather than consulting $env(p\_score)$ if that were $\neq \perp$), so that policy scores are certified before their use in certification of policy scores they depend on. Then either an equality check of $certPolicy(pol, env)$ and $env(p\_score)$ is performed – whose failure will fail certification – or we check whether the static analysis returns a real value (i.e. not $\perp$), in which case we extend the hash map so that *pol* has key *score*. Finally, the hash map is returned.

Function $certPolicy(pol, env)$ first checks whether some predicate q within policy *pol* has unspecified truth value in environment *env*. If so, it returns $\perp$ since the score of *pol* cannot be determined. Otherwise, if all predicates in *pol* are false in environment *env*, the default case applies and the evaluation of the default score s in environment *env* is returned. Finally, if some predicates in *pol* are true (and none are then false), we return the application of *op* to the evaluation $eval(s_i, env)$ of all "true" score expressions s_i in environment *env*.

```
certifyWrapper(c, I, v) {  % condition c, scenario I, and v in {false, true}
    if (recursivelyCertify(c, I, v, ∅) == true) { return success; }
    if (recursivelyCertify(c, I, v, ∅) == false) { return failure; }
    elseif { return inconclusive; }
}

recursivelyCertify(c, env, v, cp) {  % returns true, false or ⊥
    cp' = collectCertifiablePolicyScores(env);
    env' = env + cp';  % program point l2
    o = certCond(c, env', v);
    if (o == ⊥) {
        if (cp ≠ cp') {
            return recursivelyCertify(c, env', v, cp');
        } elsif (∃q: env'(q) = ⊥) {
            pick one q with env'(q) = ⊥;
            env' = env' + [q ↦ false];
            return recursivelyCertify(c, env', v, cp');
        } else {return o; } % triggers exception upstream (not shown here)
    } else {   % program point l1
        return o;  % output true means success, false means failure
    }
}
```

Figure 5: Function $recursivelyCertify(c, I, v, \emptyset)$ checks whether condition c has truth value v in empty hash map cp and scenario I where it may refine the latter. Function $certifyWrapper$ wraps this into success, failure, or inconclusive result

```
collectCertifiablePolicyScores(env) {
% returns hash map of some policies, with their statically inferred scores as keys
    cp = ∅;
    for (all declared policies pol) {
        score = certPolicy(pol, env);
        if (env(pol_score) ≠ ⊥){
            if (score ≠ env(pol_score)){
                report certification exception; break;
            }
        }
        if (score ≠ ⊥) {cp = cp + [pol ↦ score];}
    }
    return cp;
}

certPolicy(pol, env) {  % returns statically inferred policy score or ⊥
    if (∃(q_i s_i) ∈ pol: env(q_i) = ⊥) { return ⊥;}
    elseif (X^pol_env == ∅) { return  eval(s, env);}
    else { return op(X^pol_env);}
}

eval(s, env) {
% s = t_1 or s = t_1 + t_2 with t_1 being constant a, variable x or product a * x
% and t_2 being variable x not of form p.sc (modeling uncertainty)
    if (t_1 of form a) {acc = a;}
    elseif (t_1 of form p.sc) {if (env(p) ≠ ⊥) {acc = env(p);} else {return ⊥;}}
    elseif (t_1 of form x) {if (env(x) ≠ ⊥) {acc = env(x);} else {return ⊥;}}
    elseif (t_1 of form a * p.sc) {
        if (a == 0.0) {acc = 0.0;}
        elseif (env(p) ≠ ⊥) {acc = a * env(p);}
        else {return ⊥;}        % here a non-zero but env(p) equals ⊥
    }
    elseif (t_1 of form a * x) {        % here x is not of form p.sc
        if (a == 0.0) {acc = 0.0;}
        elseif (env(x) ≠ ⊥) {acc = a * env(x);}
        else {return ⊥;}        % here a non-zero but env(x) equals ⊥
    }
    if (s of form t_1 + t_2) {
        if (env(t_2) ≠ ⊥) {acc = acc + env(t_2);}
        else {return ⊥;}        % here env(t_2) equals ⊥, strict for +
    }
    return acc;
}
```

Figure 6: Function *collectCertifiablePolicyScores*(*env*) returns hash map for policies *pol* with keys *score* statically inferred as result of *pol* in *env*. Function *certPolicy* certifies whether the score of policy *pol* of form *op* $((q_1\ s_1)\ldots(q_n\ s_n))$ *default* s or *op* () *default* s in environment *env* is inferrable. Set X^{pol}_{env} denotes $\{eval(s_i, env) \mid env(q_i) = true\}$ and function $eval(s, env)$ statically infers the value of score s in environment *env*

$certCond(c, env, v)$ { % returns *true*, *false* or $\perp$; comparisons to $\perp$ return $\perp$
  if (c *of form* q) { *return* $(v == env(q))$; }
  elseif (c *of form* $\neg c_1$) { *return* $certCond(c_1, env, \neg v)$; }
  elseif (c *of form* $(c_1 \wedge c_2)$) { *if* $(v == true)$ $\{lop = \wedge;\}$ *else* $\{lop = \vee;\}$
    return $certCond(c_1, env, v)\, lop\, certCond(c_2, env, v)$; }
  } *elseif* (c *of form* $(c_1 \vee c_2)$) { { *if* $(v == false)$ $\{lop = \wedge;\}$ *else* $\{lop = \vee;\}$
    return $certCond(c_1, env, v)\, lop\, certCond(c_2, env, v)$; }
  } *elseif* (c *of form* $(pS_1 \leq pS_2)$) {
    if$(v == true)$ { *return* $certPSet(pS_1, env) \leq certPSet(pS_2, env)$; }
    else { *return* $certPSet(pS_2, env) < certPSet(pS_1, env)$; }
  } *elseif* (c *of form* $(pS_1 < pS_2)$) {
    if$(v == true)$ { *return* $certPSet(pS_1, env) < certPSet(pS_2, env)$; }
    else { *return* $certPSet(pS_2, env) \leq certPSet(pS_1, env)$; }
  }
}

$certPSet(pSet, env)$ { % returns *true*, *false* or $\perp$; if $env(pol)$ not found, returns $\perp$
  if ($pSet$ *of form* pol) {*return* $env(pol)$; }
  } *elseif* ($pSet$ *of form* $op(pS1, pS2)$) { *return* $op(certPSet(pS_1, env), certPSet(pS_2, env))$; }
}

Figure 7: Function $certCond(c, env, v)$ decides whether condition c has truth value v in environment env, and $certPSet(pSet, env)$ covers this for policies and their composition

Function $eval(s, env)$ has two types of input for s depending on whether s is a raw score t_1 or contains an uncertainty interval $[l, u]$ that we translate into Z3 code as a real variable t_2. This function does a static analysis that consults $env(p)$ when evaluating variables of form $p.\mathbf{sc}$ and consults $env(x)$ for all other variables x. This consults $env(p)$ and not $env(p.\mathbf{sc})$ so that policy scores get certified based on certified scores of policies that they depend upon. Although $\perp$ is strict for $+$, we relax its strictness for $*$ in expressions $a * x$ when a evaluates to 0.0, in which case $a * x$ also evaluates to 0.0.

Last, but not least, we turn to function $certCond(c, env, v)$ in Figure 7. It compositionally evaluates over the structure of c whether this condition computes to truth value v in environment env. This makes use of 3-valued propositional logic of Kleene [9], where for example $\perp \vee x = x$ and $\neg\perp = \perp$. The intuition is that $\perp$ stands for either *true* or *false* and that equations are valid under this interpretation. This is an abstraction as $q \vee \neg q$ evaluates to $\perp$ in this logic whenever q has value $\perp$. We note that $\perp$ is strict for comparison operators $==$, $\leq$, and $<$ in function *certCond*. If the condition c is atomic q, we check whether claimed truth value v matches what the environment says about q. If c is $\neg c_1$, we reduce this to the certification that c_1 has the negated truth value $\neg v$ in the same environment. The cases of conjunction and disjunction are dual and need to consider whether v equals *true* or *false*. This structure is also seen in comparing policy sets in a condition, which compares their scores as computed by the environment in function *certPSet* ($\perp$ indicates no score is present).

The correctness theorem for certification refers to the meaning of *Peal+* in environments where all variables have a value from their declared data type Real or Boolean. This operational semantics was given in Section 2 and Figure 2.

Theorem 1 *Let c be a Peal+ condition such that the set of policy sets occurring in c is acyclic. Let v be a truth value true or false. Let I be a scenario produced for c from a back-end. Let function $recursivelyCertify(c, I, v, \emptyset)$ return true and let env' be the value of this environment at program point l1. Let env'' refine env' such that env'' maps no variable to $\perp$. Then condition c evaluates to true in environment env'' under the operational semantics of Peal+.*

Proof Sketch: We only have to show the claim for function *certCond*, given the code struture of *recursivelyCertify*. The claim is proved using structural induction over the condition c, noting that sub-conditions also have acyclic sets of policy sets. The cases rely on that fact that $\perp$ is strict for all algebraic operators with noted exception of $eval(0.0 * \perp, env) = 0.0$.

The cases that compare two policy sets require proof of an auxiliary lemma: *"Whenever the output of $certPSet(pS, env')$ is not equal to $\perp$, then that output is the score of policy set pS in all environments that refine environment env'."* This is shown for policies and composed policy sets by structural induction.

For the first case of a policy set being a policy, we require a second auxiliary lemma: *"Let pol be a policy and env' an environment such that $env'(pol)$ is not equal to $\perp$. Then $env'(pol)$ is the score of policy pol in all environments that refine environment env'."* The proof of this lemma appeals to the linear order in which statically inferred scores of policies are added as hash keys, where env' is of form $env + cp'$ as seen at program point l2 in function *recursivelyCertify*. Since the set of policies occurring in condition c is acyclic, this order is indeed well founded and so we can use well founded induction to prove this lemma. QED

The above theorem (proven in an appendix) says that successful certification of the computed environment env' means that all "completions" of env' that resolve $\perp$ values with any legal value of the respective data type will compute the claimed truth value for the condition in question. In particular, variables x with $env'(x) = \perp$ are genuine "don't care" variables for this successful certification.

Certification runs in polynomial time in its input: the number of recursions is bounded by $m + n$ where m is the number of declared policies and n the number of predicates occurring in rules. The static analysis of conditions evaluates their parsetree over 3-valued logic, where truth values of leaves are computed by static analyses that are linear in the size of the respective policy sets.

4 Feedback for users

As described in [7], we extract raw Z3 output and render it in pretty printed form, as seen in the initial part of Figure 8. But for larger case studies, it became hard to digest even pretty printed information: one often could not see the forest for all the trees. So we now also output for each analysis a summary of the scenario, its certification, and supporting information. Figure 8 shows typical such output for the Car Rental Risks example. Scenarios also report any non-deterministic choices of uncertainty as seen for variable `b2_hasOtherLicense_U` in that figure. PEALT reports the certification outcome and refinements of predicates and real variables that certification may have brought about (when applicable), lists scores of *all* policies that certification could statically infer, and then partially evaluates *only relevant* policies (not for `b3` in Figure 8) over the successfully certified scenario to then display them in this more compact and meaningful manner. For the latter, true predicates are grouped within square brackets and reported with aggregated score in red (colors not shown in figure), as this is the score for the policy as well. Rules with false

```
Result of analysis [name2 = always_true? c3]
c3 is (pSet1 <= 50000.0) && (pSet_b4 > 0.4)

c3 is NOT always true, for example, in the scenario in which:
accidentFreeForYears is True, femaleDriver is True, isLuxuryCar is True,
mixedUsage is True, speaksEnglish is True, travelsAlone is True, ...
hasEULicense is False, hasOtherLicense is False, hasUKLicense is False, ...,
b1_score is 150000, b2_hasOtherLicense_U is 0, b2_risk_score is 1, ...

Certification of analysis [name2] succeeded.
Additional predicates set to false for certification: Set(hasUSLicense, hasEULicense)

Policy scores statically inferred in this certification process:
b1 has score 150000, b2 has score 0.6, b2_risk has score 0.4,
b3 has score 0.25, b4 has score 0.55

Policies in analysis [name2] partially evaluated in certified scenario:
b1 = max (([isLuxuryCar] 150000)) default 50000 ...
b4 = + (([accidentFreeForYears speaksEnglish] 0.55)) default 0
```

Figure 8: Output format of analyses (handedited to save space): scenario (if applicable), certification status and possible refinements, policy scores inferred during certification, and policies partially evaluated in certified scenario. Variable `b2_hasOtherLicense_U` functions as t_2 in $eval(0.4\ [-0.1, 0.1], env)$ and models choice of value from $[-0.1, 0.1]$

predicates aren't shown; in particular, if all predicates are false, an empty policy with default score in red is shown. Rules whose predicates have truth value $\perp$ are shown individually (in green) where predicates end in "?" to mark that they signify *don't care* rules.

PEALT uses Z3's `push` and `pop` constructs for incremental solving of more than one analysis. The efficiency may also raise usability issues: the output in Figure 8 was obtained after all other analyses were commented out. If we run all these analyses in their declared sequence, however, the scenario reported for `name2` will be different. Similar effects may happen when automated vacuity checking changes its OFF/ON status. On the other hand, this seems at best to make the user temporarily confused and so we don't think this issue is serious enough to give up the efficiency gains of using the `push` and `pop` constructs.

5 Case study: attack-countermeasure trees

Peal+ and its tool *PEALT* can be used as an intermediate language into which domain-specific languages can be translated and analyzed. Such use has two benefits: analysis results can be certified, and *PEALT* may perform analyzes that are not supported within the frameworks of those domain-specific languages.

We illustrate these benefits for attack-countermeasure trees (ACTs) [17] by means of an example ACT for a BGP reset of a session as discussed in [17]. The *PEALT* input code for this example would not really be meant for human consumption, as it would just be an intermediate syntax for facilitating analyses. Our translation extends the semantics of ACTs in that we may turn attack leaves, detection mechanisms, and mitigation mechanisms "on" or "off" – without compromising

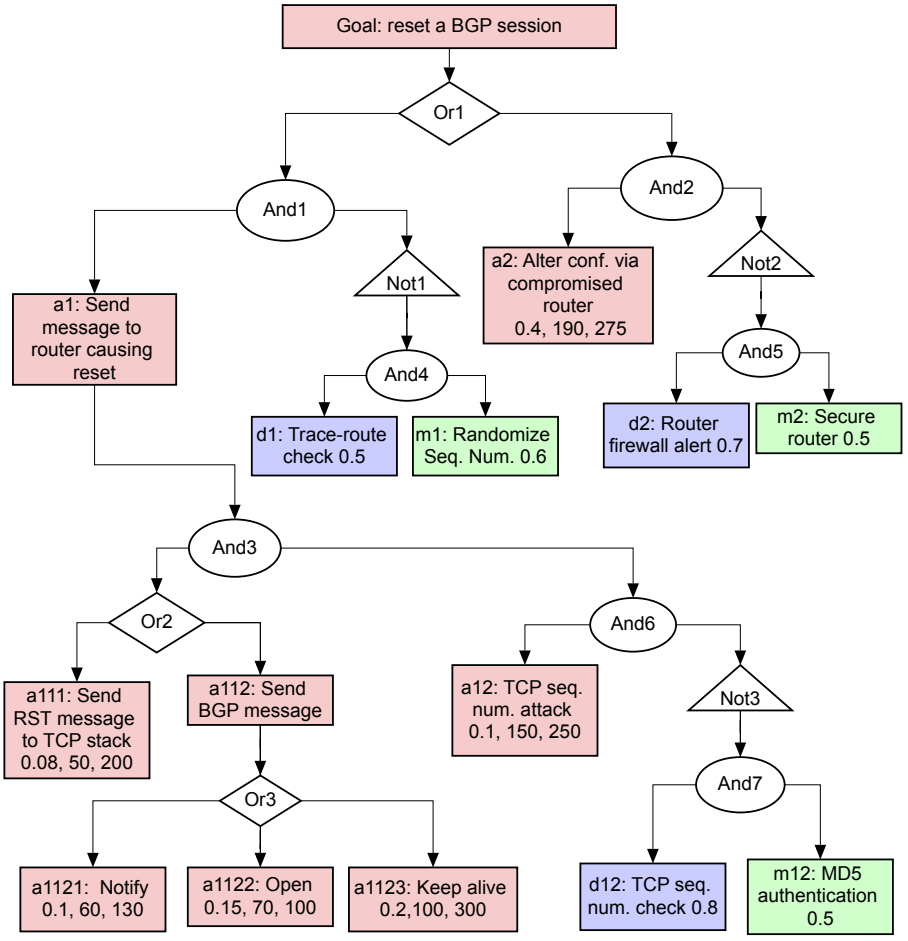

Figure 9: ACT from [17] for reset of a BGP session, with detection/mitigation leaves' probability of working and attack leaves' success probability, cost, and impact (resp.)

the computation of attack success probabilities, attack impact or attack cost. This, combined with the expressive conditions in $PEALT$, gives us richer analysis capabilities, discussed in detail below. The full $PEALT$ code for this case study is built into the $PEALT$ tool as an example case study.

Figure 9 shows the ACT taken from [17] where we merely annotated some of its nodes with policy names that we will use in our translation. This tree contains AND and OR nodes as familiar from attack trees [12]. But it also contains three NOT nodes that all feed into parent AND nodes the possible effects of a pair of detection and mitigation mechanisms. Qualitatively, this means that such a pair of working detection and mitigation mechanisms will feed *false* into the parent AND node. The probabilistic interpretation in [17] is that both mechanisms have a probability of working, and so NOT nodes take as probability the complement of the product of these two probabilities of working mechanisms [17].

The probability of attack success and impact cost are computed over the structure of the ACT [17], whereas attack cost is computed by first producing the set of all min-cuts (as used in fault tree analysis [2]) of the ACT [17]. This makes it hard to reason about the interaction of success probabilities, impact, and cost. Also, it faces scalability issues as the number of min-cuts may be exponential in the size of the ACT. We here want to demonstrate that the use of SMT solvers,

facilitated with *Peal+* and *PEALT* as intermediate language and tool, allows us to reason about such interactions and avoids the need to enumerate all min-cuts.

The declaration of policies for the probability of attack success, the result of policy `goal`, is shown in Figure 10. A predicate `True`, asserted to always be true, is used to compose results of children in the ACT. The probability at an OR node with n children x_i is $1 - \prod_{k=1}^{n}(1 - prob(x_k))$, and we expand this arithmetic term in stages using policy scores for stage composition, as seen for policy `or1`. The probability at an AND node with m children y_j is $\prod_{k=1}^{m} prob(x_k)$, and we similarly encode this arithmetic expression, as seen for policy `and1`.

For the encoding of attack leaves, their success probability is the score of a sole rule that captures that attack event. Since attack leaves are not under the scope of a NOT node, their default score is 0.0. The encoding of a NOT node is simply $1 - x$ where x is the result of its child AND node. For that AND node, the staged computation checks whether both detection and mitigation are present, in which case it computes the product of the probabilities of both mechanisms working; otherwise, it returns 0.0. This default score is sound as it makes the NOT node default to 1.0 which has no effect on its predecessors in the ACT (there is no NOT node in the scope of another NOT node). Thus this translation works for ACTs since they don't have nested NOT nodes.

In Figure 11, cost of attacks to an attacker and overall attack cost are specified. Default scores capture cost in the absence of attacks and so equal 0.0. In contrast to [17], overall cost is here the sum of all occurring, i.e. *true*, attacks since analyses ask whether attacks succeed within cost budgets and Z3 will search for such solutions by turning attack leaves "on" or "off" as desired.

The specification of attack impact (given in an appendix) reflects that the impact of an OR node is the maximum of the impact of all its children – modeling a worst-case scenario for the system [17]; and that the impact of an AND node is the sum of the impact of all its children. As in [17], NOT nodes don't contribute to impact of attack success, although it is noted in [17] that detection and mitigation mechanisms can reduce risk.

Finally, we may specify questions about this ACT in *Peal+*. Using basic conditions such as `549.0 < impact_overall` and binary conjunction, we express condition `c6` which asks whether the attack impact can be strictly above 549.0, the attack cost can be less than or equal to 440.0, and the probability of attack success can be strictly above 0.41199 – *all in the same scenario*. *PEALT* reports that this is possible in a scenario in which attacks `a1123`, `a2`, and `a12` occur (i.e. are *true*), as well as detection mechanisms `d1` and mitigation mechanism `m2`. The latter two may be unexpected. But in the scenario neither the mitigation mechanism `m1` of `d1` nor the detection mechanism `d2` of `m2` occur (i.e. are *false*). Therefore, none of the two respective NOT nodes contribute to the probability of attack success; and NOT nodes contribute neither to impact not to cost.

Threshold values chosen in condition `c6` are co-dependent: we can't decrease 440.0 by 1 or more, increase 549.0 by 1 or more, or increase 0.41199 by 0.00001 or more without making condition `c6` unsatisfiable. These values were determined by repeated analysis that adjusted these values with bisection search using SAT/UNSAT results to drive the bisection method. It would be of interest to automate such search over objective functions within *PEALT* in future work. If we add to condition `c6` a conjunct, saying that the detection/mitigation pair `d2` and `m2` also has to occur, *PEALT* informs us that this is now impossible.

We can also approximate maxima of security metrics, for example a measure of expected system damage $f(p, i, c) = p \cdot max(0, 2i - c)$ for attack success probability p, attack cost c, and attack impact i – exploiting that p, i, and c are expressed as policies. For example, $271.919999999999 < f(p, i, c)$ is satisfiable for this ACT wheras $271.92 < f(p, i, c)$ is not. All scenarios found in this case study

```
goal = +((True or1_score)) default 1.0
or1 = +((True 1.0) (True -1.0*or1_aux_score)) default 1.0
or1_aux = *((True or1_aux1_score) (True or1_aux2_score)) default 1.0
or1_aux1 = +((True 1.0) (True -1.0*and1_score)) default 1.0
or1_aux2 = +((True 1.0) (True -1.0*and2_score)) default 1.0
and1 = *((True and3_score) (True not1_score)) default 1.0
and3 = *((True or2_score) (True and6_score)) default 1.0
or2 = +((True 1.0) (True -1.0*or2_aux_score)) default 1.0
or2_aux = *((True or2_aux1_score) (True or2_aux2_score)) default 1.0
or2_aux1 = +((True 1.0) (True -1.0*a111_score)) default 1.0
or2_aux2 = +((True 1.0) (True -1.0*or3_score)) default 1.0
a111 = +((sendRSTmessageToTCPStack 0.08)) default 0.0
or3 = +((True 1.0) (True -1.0*or3_aux_score)) default 1.0
or3_aux = *((True or3_aux1_score) (True or3_aux2_score)
                    (True or3_aux3_score)) default 1.0
or3_aux1 = +((True 1.0) (True -1.0*a1121_score)) default 1.0
or3_aux2 = +((True 1.0) (True -1.0*a1122_score)) default 1.0
or3_aux3 = +((True 1.0) (True -1.0*a1123_score)) default 1.0
a1121 = +((notify 0.1)) default 0.0
a1122 = +((open 0.15)) default 0.0
a1123 = +((keepAlive 0.2)) default 0.0
not1 = +((True 1.0) (True -1.0*and4_score)) default 1.0
and4 = +((traceRouteCheck and4_aux1_score)) default 0.0
and4_aux1 = +((randomizeSequenceNumbers and4_aux2_score)) default 0.0
and4_aux2 = *((True 0.5) (True 0.6)) default 1.0
```

Figure 10: Policies that compute probability of attack success, even when certain attacks, detection mechanisms or mitigation mechanisms may be absent. Policies for sub-ACTs And2 and And6 are similar to those for And1 and now shown

```
cost_a111 = +((sendRSTmessageToTCPStack 50.0)) default 0.0
cost_a1121 = +((notify 60.0)) default 0.0
cost_a1122 = +((open 70.0)) default 0.0
cost_a1123 = +((keepAlive 100.0)) default 0.0
cost_a12 = +((TCPsequenceNumberAttack 150.0)) default 0.0
cost_a2 = +((alterConfigurationViaCompromisedRouter 190.0)) default 0.0
cost_overall = +((True cost_a111_score) (True cost_a1121_score)
                        (True cost_a1122_score) (True cost_a1123_score)
                        (True cost_a2_score) (True cost_a12_score)) default 0.0
```

Figure 11: Computing cost of attack leaves and overall cost of occurring attacks

certified successfully without refining any predicates.

6 Discussion and Evaluation

We analyzed and certified about 20,000 random conditions with uncertainties but a few of these conditions failed to certify. We isolated the source of these failures to be an anomaly of the Z3 `push` command. With help of Arie Gurfinkel, Nikolaj Bjorner was able to attribute this to Z3 work item 108 (see `http://z3.codeplex.com/workitem/108`): if some constraints are non-linear , use of `push` invokes a legacy solver that may report incorrect models for SAT outcomes. Since *PEALT* won't use `push` when a sole analysis executes, we can eliminate this Z3 bug as source of certification failure by turning off vacuity checking and commenting out all other analyses. We think *PEALT* therefore strikes a good balance between performance (which use of `push` on more than one analysis greatly improves) and correctness (since failed certifications are rare and caused by this bug or by typos as discussed next).

If a user declares a policy `p` but also writes `p` in a score instead of `p_score`, the SMT solver may find a real value for real variable `p` (implicitly declared in that rule!) and so $env(p)$ would have that value. If this is not the value one would statically infer for policy `p`, such aliasing will fail certification. Also, spelling mistakes in variable names may declare new variables that can result in inconclusive certification. Anecdotal evidence suggests that almost all failed or inconclusive certifications are results of such typos, which won't occur whenever *PEALT* is used as intermediate language by code generators.

The certification process in *PEALT* only works for scenarios (whose reported values for policy scores are ignored in certification), not for a claim that no scenario exists. We first focused our efforts on scenarios as they are likely to be more useful to specifiers, and since certification of non-existence of scenarios involves formal proofs extracted from back-ends (e.g. [3]), but general specifiers cannot be expected to understand complex proofs.

The scope of certification does not expand into section `DOMAIN_SPECIFICS`. For example, assume that a predicate occurs in no rule but is cast to a condition and declared in section `DOMAIN_SPECIFICS`, which also defines its meaning. Our certification will not inspect this definition of meaning as it is expressed outside of *Peal+* in an expressive logic. We did not find this to be limiting when writing and certifying *PEALT* models, but it means that certification is a relative notion in *PEALT*. On the other hand, it seems feasible to extend our 3-valued certification to cover `DOMAIN_SPECIFICS` as well for fragments of Z3's input language.

Our implementation of *Peal+* requires that policies be cast into policy sets (when needed), predicates be cast into conditions (when needed), and operators for policies, policy sets, and conditions be unary or binary (not n-ary). The latter is a good thing, since it means that all subconditions of conditions are explicitly declared and so subject to vacuity checking. *PEALT* does not check whether predicates within a policy occur more than once. The latter is an issue when two or more such occurrences have scores with uncertainty as this "binds" the non-deterministic choice made for these expressions to the same value. Our BGP case study with uncertainties in *PEALT* addresses this be using `True1`, `True2`, etc. to disambiguate this.

PEALT has no explicit ability to model state spaces and their transition; one may see this as a weakness and opportunity for future work, or as a strength as it avoids state space explosions.

7 Related work

We are not aware of much publicly accessible literature on the independent certification of formal methods results and report two results of that type here. For model checking, Namjoshi developed deductive techniques in [15] that can independently verify the results of model checks for formulas of the modal mu-calculus and where these proofs can be extracted from an (instrumented) model checking run. For theorem proving, Gonthier [5] simplified the proof of the famous 4-color theorem, and proved it in the theorem prover Coq in such a manner that the proof itself could be certified as well.

Jha et al. [8] use model checking to automatically generate attack graphs with nodes representing network states, develop techniques for choosing minimal number of security measures and for trading off attack likelihood and attack probability. Attack graphs that express dependencies of vulnerabilities instead, such as those of Albanese et al. [1], have more scalable analyses than state-based ones. Attack graph models in the literature appear to have a fixed model signature, whereas *PEALT* can extend modeling domains as and when needed.

Language *Peal+* extends *Peal* [4] and the tool *PEALT* over its version in [7]: *PEALT* now supports the richer language *Peal+*, automated vacuity checking of all declared analyses, the automated certification of all scenarios generated by Z3 for analyses, and the partial evaluation of policies over scenarios so that users can comprehend scenario information directly on relevant policies.

In [6], we sketched *Peal+* and illustrated it with mock-up syntax for a "score card" model very similar to that from Figure 4. Although that paper discussed usability issues, it focussed on the design of *Peal+* and did not cover usability issues of a supporting tool and its user feedback.

8 Conclusions and future work

In this paper we presented a language *Peal+* in which the interaction of concepts that inform security and threat analysis can be formally expressed and analyzed. We reported its implementation in tool *PEALT* that statically analyzes such conditions with two principal aims: to determine whether specified conditions meet expectations of how security-related concepts influence decision making; and to validate that the expectations that users have do not have unintended consequences when expressed and enforced in such conditions.

PEALT reflects the methodology of *auto-interactive verification* (see Fig. 1). This means users can rely on automated verification tools that provide easily comprehended feedback which may trigger subsequent modeling and automated verification. And this process would be repeated until users are satisfied to have captured conditions as desired. This paper developed foundations for language *Peal+* and implemented them in tool *PEALT*, using the SMT solver Z3 as back-end for automated reasoning and scenario generation to realize this methodology.

We created support for validating scenarios computed for conditions expressed in *Peal+*: an independent certification of the correctness of scenarios with respect to the domain and policies in which they should be interpreted. We stress that our certification is agnostic to the manner in which code for analysis in back-ends is generated (since certification operates on *Peal+* expressions directly) and agnostic to the choice of back-end (apart from an interface for the scenario to be certified and for variables modelling uncertainty). All policies that certification seems to rely upon, *PEALT* partially evaluates with respect to the certified scenario and provides this as auxilliary feedback, so that modelers may more easily assess the impact of policies certified in possibly refined

scenarios.

We illustrated the utility of *Peal+* and these support mechanisms by first discussing a Car Rental Risks example and then attack-countermeasure trees. We showed how ACTs can be translated into *Peal+* so that we can reason about interaction of the probability of attack success, attack cost, and attack impact whilst at the same time allow the model to turn attack, detection, and mitigation leaves "on" or "off" at will. Therefore, our ACTs actually represent an entire set of ACTs and we can verify invariants of such interaction over that set of ACTs.

It will be of interest to extend *Peal+* with judicious support for integer variables (a potential performance bottleneck for SMT solvers) and optimization with respect to non-linear objective functions. We also mean to develop auxiliary tools that can translate other threat modeling formalisms into *PEALT* for richer analysis, as illustrated for ACTs in this paper. Finally, we mean to research how we can extend *Peal+*, *PEALT*, and our certification to state transitions and to conditions that analyse state changes through operators of temporal logic.

Acknowledgements

We are grateful that this work was supported by funding from Intel® Corporation within its *Trust Evidence* research project. Charles Morriset and Jason Crampton suggested some of the terminology of *Peal* and helped with designing some early versions of code generators for *Peal*. Jason is also thanked for suggesting to improve the presentation of this paper.

References

[1] Albanese, M., Jajodia, S., Noel, S.: Time-efficient and cost-effective network hardening using attack graphs. In: Proc. of the 42nd Ann. IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). pp. 1–12. IEEE Computer Society (2012)

[2] Bedford, T., Cooke, R.: Probabilistic Risk Analysis: Foundations and Methods. Cambridge University Press (2001)

[3] Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: ITP. pp. 179–194 (2010)

[4] Crampton, J., Huth, M., Morisset, C.: Policy-based access control from numerical evidence. Tech. Rep. 2013/6, Imperial College London, Department of Computing (October 2013), ISSN 1469-4166 (Print), ISSN 1469-4174 (Online)

[5] Gonthier, G.: The four colour theorem: Engineering of a formal proof. In: 8th Asian Symp. on Computer Mathematics. p. 333. LNCS 5081, Springer (2007)

[6] Huth, M., Kuo, J.H.P.: On designing usable policy languages for declarative trust aggregation. In: HCI (24). pp. 45–56 (2014)

[7] Huth, M., Kuo, J.H.P.: PEALT: An automated reasoning tool for numerical aggregation of trust evidence. In: Proc. of TACAS 2014. Lecture Notes in Computer Science, vol. 8413, pp. 109–123. Springer (2014)

[8] Jha, S., Sheyner, O., Wing, J.: Two formal analys s of attack graphs. In: Proc. of 15th IEEE Workshop on Comp. Sec. Found. pp. 49–63. IEEE Comp. Soc. (2002)

[9] Kleene, S.C.: Introduction to Metamathematics. North Holland (1952)

[10] Korzhyk, D., Yin, Z., Kiekintveld, C., Conitzer, V., Tambe, M.: Stackelberg vs. nash in security games: An extended investigation of interchangeability, equivalence, and uniqueness. J. Artif. Int. Res. 41(2), 297–327 (May 2011)

[11] Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. In: 10th IFIP WG 10.5 Advanced Research Working Conference (CHARME'99). pp. 82–96. LNCS 1703, Springer (1999)

[12] Mauw, S., Oostdijk, M.: Foundations of attack trees. In: Proc. of 8th Int'l Conf. on Information Security and Cryptology. pp. 186–198. LNCS 3935, Springer (2006)

[13] Moore, A., Ellison, R., Linger, R.: Attack modeling for information security and survivability. Tech. Rep. Technical Note CMU/SEI-2001-TN-00, Software Engineering Institute, Carnegie Mellon University (2000)

[14] de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340 (2008)

[15] Namjoshi, K.S.: Certifying model checkers. In: 13th Int'l Conf. on Computer Aided Verification. pp. 2–13. LNCS 2102, Springer (2001)

[16] Pavlidis, N.G., Tasoulis, D.K., Adams, N.M., Hand, D.J.: Adaptive consumer credit classification. Journal of the Operational Research Society 63(12), 1645–1654 (2012)

[17] Roy, A., Kim, D.S., Trivedi, K.S.: Attack countermeasure trees (act): towards unifying the constructs of attack and defense trees. Security and Communication Networks 5(8), 929–943 (2012)

[18] Schneier, B.: Secrets and Lies: Digital Security in a Networked World. John Wiley and Sons Inc. (2000)

[19] Wood, M.: Flaw calls for altering passwords, experts say. The New York Times (Technology) (9 April 2014)

A *PEALT* source code and installation

Tool *PEALT* is written in the Scala programming lanugage. The latest version of tool *PEALT* and its installation instructions are found at

`http://www.doc.ic.ac.uk/~hk2109/PEALT/`

B Formal semantics of policy sets

We here discuss the operational semantics given in Figure 2 in more detail. Let e be an environment for the evaluation of a set of policy sets $\mathcal{P}$. Then e is a function that maps predicates q_i to truth values *false* or *true* (written $e(x)$), variables x not of form $p.\mathbf{sc}$ to a real number ($e(x)$), and declared uncertainty intervals $[l, u]$ to some element of $[l, u]$. It should be noted that intervals are uniquely identified by their policy and predicate (or by their policy if the interval is declared in the default score). So formally, we write $e(pol, q_i, [l, u])$ respectively $e(pol, default, [l, u])$ to also identify the respective location of these intervals.

The operational semantics $\mathcal{E}_e$ for policy sets in $\mathcal{P}$ in environment e is then given in Figure 2. At the top of that figure, we see the computation of the meaning of policy sets using structural induction (policies as base case, and composed policies as other case). For policies, this distinguishes between the case in which no predicate is true in the environment e (the default cause in which Z is empty) and the case when Z is non-empty. The former case also gives meaning to policies that contain no rules. At the bottom of the figure, we see a definition of how score expressions evaluate in environment e. This makes use of the contextual information that maps intervals to their declared locations.

It should be clear that this is a well defined semantics whenever the set of policy sets if acyclic.

C Proof of Theorem 1

It suffices to prove the claim for environment env', provided that the argument made never has to appeal to $\perp$ outputs of env'. Given the structure of the code body for *recursivelyCertify*, we only have to show the respective claim for function *certCond*. That is to say, we need to show that condition c has truth value v in environment env' whenever $certCond(c, env', v)$ outputs *true*. We proceed by structural induction over the condition c.

- Let c be a predicate q. Then $certCond(q, env', v)$ outputs $v == env'(q)$ which is *true*. Since $\perp$ is strict for such equality checks, we infer from this that $env'(q) \neq \perp$ and so this is a truth value that must therefore equal v, which is either *true* or *false*. This shows that condition q has truth value v in environment env', as claimed.

- Let c be $\neg c_1$. Then function $certCond(\neg c_1, env', v)$ returns whatever $certCond(c_1, env', \neg v)$ returns, which is *true*. By induction on c, we infer from this that condition c_1 has truth value $\neg v$ in environment env'. Therefore, $\neg c_1$ has truth value $\neg\neg v = v$ in environment env', as claimed.

- Let c be $c_1 \wedge c_2$. There are two cases, one for each value of v:

  - Let v be *true*. Then the value of function $certCond(c_1 \wedge c_2, env', true)$ equals $certCond(c_1, env', true) \wedge certCond(c_2, env', true)$ which outputs *true*. Therefore, we know that both $certCond(c_1, env', true)$ and $certCond(c_2, env', true)$ output *true*. By induction, we infer from this that c_1 has truth value *true* in environment env', and that c_2 has truth value *true* in environment env'. Therefore, $c_1 \wedge c_2$ has truth value *true* in environment env', as claimed.
  - Let v be *false*. Then the value of function $certCond(c_1 \wedge c_2, env', false)$ equals $certCond(c_1, env', false) \vee certCond(c_2, env', false)$ which outputs *true*. Therefore,

there is some j in $\{1, 2\}$ such that $certCond(c_j, env', false)$ outputs $true$. By induction, we infer from this that c_j has truth value $false$ in environment env'. Therefore, $c_1 \wedge c_2$ has truth value $false$ in environment env', as claimed.

- Let c be $c_1 \vee c_2$. There are two cases, one for each value of v:
  - Let v be $true$. Then the value of function $certCond(c_1 \vee c_2, env', true)$ equals $certCond(c_1, env', true) \vee certCond(c_2, env', true)$ which outputs $true$. Therefore, we know that there is some j in $\{1, 2\}$ such that $certCond(c_j, env', true)$ outputs $true$. By induction, we infer from this that c_j has truth value $true$ in environment env'. Therefore, $c_1 \vee c_2$ has also truth value $true$ in environment env', as claimed.
  - Let v be $false$. Then the value of function $certCond(c_1 \vee c_2, env', false)$ equals $certCond(c_1, env', false) \wedge certCond(c_2, env', false)$ which outputs $true$. Therefore, we know that both $certCond(c_1, env', false)$ and $certCond(c_2, env', false)$ output $true$. By induction, we infer from this that c_1 has truth value $false$ in environment env', and that c_2 has truth value $false$ in environment env'. Therefore, $c_1 \vee c_2$ has truth value $false$ in environment env', as claimed.

- Let c be $pS_1 \leq pS_2$. There are two cases, one for each value of v:
  - Let v be $true$. Then the value of $certCond(pS_1 \leq pS_2, env', true)$ equals $certPSet(pS_1, env') \leq certPSet(pS_2, env')$ which outputs $true$. Since $\perp$ is strict for $\leq$, we infer from this that both $certPSet(pS_1, env')$ and $certPSet(pS_2, env')$ are different from $\perp$. By Lemma 1, this implies that $certPSet(pS_j, env')$ is the score of policy set pS_j in environment env' for all j in $\{1, 2\}$. Therefore, it follows that condition $pS_1 \leq pS_2$ has truth value $true$ in environment env', as claimed, given that $certPSet(pS_1, env') \leq certPSet(pS_2, env')$ outputs $true$.
  - Let v be $false$. Then the value of $certCond(pS_1 \leq pS_2, env', false)$ equals $certPSet(pS_2, env') < certPSet(pS_1, env')$ which outputs $true$. Since $\perp$ is strict for $<$, we infer from this that both $certPSet(pS_1, env')$ and $certPSet(pS_2, env')$ are different from $\perp$. By Lemma 1, this implies that $certPSet(pS_j, env')$ is the score of policy set pS_j in environment env' for all j in $\{1, 2\}$. Therefore, it follows that condition $pS_1 \leq pS_2$ has truth value $false$ in environment env', as claimed, given that $certPSet(pS_2, env') < certPSet(pS_1, env')$ outputs $true$.

- Let c be $pS_1 < pS_2$. There are two cases, one for each value of v:
  - Let v be $true$. Then the value of $certCond(pS_1 < pS_2, env', true)$ equals $certPSet(pS_1, env') < certPSet(pS_2, env')$ which outputs $true$. Since $\perp$ is strict for $\leq$, we infer from this that both $certPSet(pS_1, env')$ and $certPSet(pS_2, env')$ are different from $\perp$. By Lemma 1, this implies that $certPSet(pS_j, env')$ is the score of policy set pS_j in environment env' for all j in $\{1, 2\}$. Therefore, it follows that condition $pS_1 < pS_2$ has truth value $true$ in environment env', as claimed, given that $certPSet(pS_1, env') < certPSet(pS_2, env')$ outputs $true$.
  - Let v be $false$. Then the value of $certCond(pS_1 < pS_2, env', false)$ equals $certPSet(pS_2, env') \leq certPSet(pS_1, env')$ which outputs $true$. Since $\perp$ is strict for

$<$, we infer from this that both $certPSet(pS_1, env')$ and $certPSet(pS_2, env')$ are different from $\perp$. By Lemma 1, this implies that $certPSet(pS_j, env')$ is the score of policy set pS_j in environment env' for all j in $\{1, 2\}$. Therefore, it follows that condition $pS_1 < pS_2$ has truth value *false* in environment env', as claimed, given that $certPSet(pS_2, env') \leq certPSet(pS_1, env')$ outputs *true*. QED

We state and prove the first auxiliary lemma needed in the proof of Theorem 1 above.

Lemma 1 *Let pS be a policy set and env' an environment such that the output of $certPSet(pS, env')$ is not equal to $\perp$. Then that output is the score of policy set pS in all environments that refine environment env'.*

Proof: Again, it suffices to prove this for environment env', provided that the argument made never appeals to $\perp$ outputs of env'. We do structural induction of the policy set pS. There are two cases to consider:

- Let pS be a policy pol. Then the output of $certPSet(pS, env')$ is the output of $certPolicy(pol, env')$ which is $env'(pol)$, and so the latter output is not equal to $\perp$. By Lemma 2, this implies that the score of policy pol is the outout of $certPolicy(pol, env')$, as claimed, as this is the output of $certPSet(pS, env')$.

- Let pS be of form $op(pS_1, pS_2)$ for op in $\{min, max, +, *\}$ and policy sets pS_1 and pS_2. Then the output of $certPSet(op(pS_1.pS_2), env')$ is the output of $op(certPSet(pS_1, env'), certPSet(pS_2, env'))$, which is therefore not equal to $\perp$. Since $\perp$ is strict for all operators op, we infer from this that both $certPSet(pS_1, env')$ and $certPSet(pS_2, env')$ are not equal to $\perp$. By induction, this implies that for all j in $\{1, 2\}$ the output of $certPSet(pS_j, env')$ is the score of policy set pS_j. Therefore, the operational semantics of $Peal+$ implies that the score of policy set $op(pS_1, pS_2)$ is the output of $op(certPSet(pS_1, env'), certPSet(pS_2, env'))$, as claimed. QED

We state and prove the second auxiliary lemma needed in the proof of Lemma 1 above.

Lemma 2 *Let pol be a policy and env' an environment such that $env'(pol)$ is not equal to $\perp$. Then $env'(pol)$ is the score of policy pol in all environments that refine environment env'.*

Proof: Again, it suffices to prove this for environment env', provided that the argument made never appeals to $\perp$ outputs of env'. Let us write env' in the form $env + cp'$, as seen in program point l2 for function *recursivelyCertify*. Then $env'(pol)$ equals $cp'(pol)$ which in turn equals $certPolicy(pol, env)$ as computed in function $collectCertifiablePolicyScores(env)$. So we know that $certPolicy(pol, env)$ is different from $\perp$, meaning that we have only two cases to consider in its body. In these cases, we apply well founded induction on the (linear) order in which hash keys $env'(p) \neq \perp$ were added to env' for declared policies p. Since there are no circular dependencies in the use of score variables for policies, this order is indeed well founded.

- **First case:** Let X^{pol}_{env} be empty. Then $env'(pol)$ equals $eval(s, env)$ where s is the default score of policy pol. In particular, $eval(s, env)$ is different from $\perp$. We do a case analysis over the structure of score s:

- Let s be a constant real a. Then $eval(a, env)$ is defined to be a and this is the score of *pol* as all rules in that policy have false predicates in environment env, and so also in its refinement env'.
- Let s be a policy score variable p_score for some declared policy p. Then $eval(p\_score, env)$ is not $\perp$ and so it is defined as $env(p)$. In particular, $env(p)$ is not equal to $\perp$. So this key must have been added to env prior to the key for *pol*. We can therefore appeal to well founded induction on the order in which such keys are added to conclude that $env(p)$ is the score of policy p in environment env, and so in its refinement env' as well. But then this is also the score of *pol* in these environments.
- Let s be a real variable x not of form p_score. Then $eval(x, env)$ is not equal to $\perp$ and so it equals $env(x)$. The latter then is the score of *pol* as all rules in that policy have false predicates in environment env, and so also in its refinement env'.
- Let s be of form $a * p\_score$ for a constant a and some declared policy p. Then $eval(a * p\_score, env)$ is not equal to $\perp$. This implies that $env(p)$ is not equal to $\perp$ or that a equals 0.0. In the latter case, it is clear that 0.0 is the score of policy *pol* in env and so in all of its refinements. It remains to consider the case when a is non-zero and $env(p)$ is not $\perp$. Then $eval(a * p\_score, env)$ equals $a * env(p)$. Since $env(p)$ is not equal to $\perp$, this key $env(p)$ must have been added to env prior to the key for *pol*. We can therefore appeal to well founded induction on the order in which such keys are added to conclude that $env(p)$ is the score of policy p in environment env. Thus, $a * env(p)$ is the score of policy *pol* in these environments.
- Let s be of form $a * x$. Then $eval(a * x, env)$ is not equal to $\perp$. In particular, $env(x)$ is not equal to $\perp$ or a equals 0.0. In the latter case, it is clear that 0.0 is the score of policy *pol* in env and so in all of its refinements. It remains to consider the case when a is non-zero and $env(x)$ is not $\perp$. Then $a * env(x)$ is the default score of policy *pol* in environment env and all its refinements, as claimed.
- Let s be of form $t_1\ [l, u]$ where t_1 is one of the raw score expressions of the previous items and variable t_2 models the non-deterministic value from interval $[l, u]$. Since $eval(s, env)$ is not equal to $\perp$, we know that $env(t_2)$ is not equal to $\perp$ and that $eval(s, env)$ equals $eval(t_1, env) + env(t_2)$. In particular, $eval(t_1, env)$ is not equal to $\perp$ as $\perp$ is strict for $+$. And we already reasoned that in this case $eval(t_1, env)$ is the default score of the version of *pol* that won't contain interval $[l, u]$ in its default score. Thus, adding $env(t_2)$ to this yields the default score of policy *pol* as claimed.

- **Second case:** Let X^{pol}_{env} be non-empty. Then $certPolicy(pol, env)$ outputs $op(X^{pol}_{env})$. As the latter is not equal to $\perp$ and $\perp$ is strict for op, we infer from this that $eval(s_i, env)$ is not equal to $\perp$ for all s_i with $env(q_i) = true$. As in the first case when X^{pol}_{env} is empty, we can analyze the structure of each score s_i to show that $eval(s_i, env)$ is the score of rule $(q_i\ s_i)$ in policy *pol* in environment env'. Thus, $op(X^{pol}_{env})$ equals the score of policy *pol* in environment env, and so it its refinement env'. QED

D Example of certification failure due to `push` Z3 bug

Figure 12 shows *PEALT* input that, when run with option "General scores", will successfully certify the first analysis but where certification fails for the second analysis. If we then turn

automated vacuity checking OFF under "Settings", comment out the first analysis, and run this again, *PEALT* will now successfully verify the second analysis. The reason is that only one analysis in total then executes, which *PEALT* will realize and so it will avoid the use of the `push` construct in this case.

E Termination of certification

The criterion that policies within conditions are acyclic is sufficient but not necessary. For example, the condition in Figure 13 is cyclic but our tool will find and successfully certify a scenario for its analysis – *PEALT* does not do any dependency analysis. But adding as domain-specific constraint that `q1` and `q3` are true makes the certification not terminate, and so no output will be reported unless users instead generate Z3 code and raw Z3 output without certification.

F Computation of attack impact for our case study

Figure 14 shows how our *PEALT* input models the computation of the impact of an attack.

```
POLICIES
b0 = min ((q14 0*vl) (q0 0.7850)) default 0.8919
b1 = * ((q6 0.6819) (q7 0.7271)) default 0.5390
b2 = * ((q12 0.3504) (q0 0.4032)) default 3*vp
b3 = + ((q3 0.3078) (q7 0.1332)) default 0.7163
b4 = max ((q3 0.0948 [-0.1435,0.4347]) (q11 0.1327)) default 0.8418
b5 = * ((q3 0.3235) (q9 2*v0 [-0.3561,0.7747])) default 0*vy
b6 = max ((q14 0.5613) (q4 0.6564)) default 0.6351
b7 = + ((q8 vx) (q2 0.9709)) default 0.5696 [-0.8854,0.0560]
b8 = * ((q14 0.7031) (q5 0.6469)) default 0.3753
b9 = min ((q11 0.3598) (q7 0.4311)) default 0.0868
b10 = * ((q10 0.1041) (q12 0.6119)) default 0.8983
b11 = + ((q1 0.4650) (q4 0.6019)) default 0.3478
b12 = min ((q3 3*vu [-0.2797,0.6717]) (q9 2*vu)) default 0.0223
b13 = max ((q5 0.9802) (q2 0.9236)) default 0.8039
b14 = min ((q6 2*vk) (q13 0.4516)) default 2*vl
b15 = max ((q13 0.3230) (q12 0.9485)) default 0.6186
b16 = + ((q1 0.8562 [-0.9047,0.6472]) (q11 3*vb)) default 0.6468
b17 = + ((q4 2*vl) (q11 0.8956)) default 0.5290
b18 = min ((q6 0.0261) (q4 0.8287)) default 0.9865
b19 = max ((q0 0.0663) (q10 0.5418)) default 0.7368
POLICY_SETS
p0_1 = min(b0,b1)
p2_3 = min(b2,b3)
p4_5 = min(b4,b5)
p6_7 = min(b6,b7)
p8_9 = min(b8,b9)
p10_11 = min(b10,b11)
p12_13 = min(b12,b13)
p14_15 = min(b14,b15)
p0_3 = max(p0_1,p2_3)
p4_7 = max(p4_5,p6_7)
p8_11 = max(p8_9,p10_11)
p12_15 = max(p12_13,p14_15)
p0_7 = +(p0_3,p4_7)
p8_15 = +(p8_11,p12_15)
p0_15 = *(p0_7,p8_15)

p16_17 = min(b16, b17)
p18_19 = +(b18, b19)
p0_15_0 = min(p0_15,p16_17)
p0_15_1 = max(p0_15_0,p18_19)
CONDITIONS
cond1 = 0.50 < p0_15_1
cond2 = 0.60 < p0_15_1
ANALYSES
analysis2 = always_false? cond2
analysis3 = always_false? cond2
```

Figure 12: *PEALT* input for which the second analysis fails certification

```
POLICIES
b1 = +((q1 b2_score) (q2 0.5)) default 0.0
b2 = max((q3 b1_score) (q4 0.4)) default 1.0
POLICY_SETS
pSet1 = min(b1,b2)
CONDITIONS
c1 = 0.0 < pSet1
ANALYSES
name1 = satisfiable? c1
```

Figure 13: Two policies with cyclic dependencies but where the condition generates a successfully certified scenario by making `q1` or `q3` false. However, the above won't terminate in $PEALT$ if we add (`assert` (`and q1 q3`)) in `DOMAIN_SPECIFICS`

```
impact_or2 = max((sendRSTmessageToTCPStack 200.0) (notify 130.0) (open 100.0)
                        (keepAlive 300.0)) default 0.0
impact_and3 = +((True impact_or2_score) (TCPsequenceNumberAttack 250.0))
                       default 0.0
impact_and1 = +((True impact_and3_score)) default 0.0
impact_and2 = +((alterConfigurationViaCompromisedRouter 275.0)) default 0.0
impact_overall = max((True impact_and1_score)
                            (True impact_and2_score)) default 0.0
```

Figure 14: Computing impact of attack leaves and overall impact of occurring attacks