PEALT: An Automated Reasoning Tool for Numerical Aggregation of Trust Evidence

Michael Huth and Jim Huan-Pu Kuo

Department of Computing, Imperial College London
London, SW7 2AZ, United Kingdom
{m.huth, jimhkuo}@imperial.ac.uk

Abstract. We present a tool PEALT that supports the understanding and validation of mechanisms that numerically aggregate trust evidence of potentially heterogenous sources. Such mechanisms are expressed in the policy composition language Peal and subjected to vacuity checking, sensitivity analysis of thresholds, and policy refinement. Verification code is generated by either compiling away numerical references prior to constraint solving or by delegating numerical reasoning to Z3, the common back-end constraint solver of PEALT. The former gives compact diagnostics but restricts value ranges and may be space intensive. The latter generates compact verification code, but gives verbose diagnostics, and may struggle with multiplicative reasoning. We experimentally compare code generation and verification running times of these methods on randomly generated analyses and on a non-random benchmark modeling majority voting. Our findings suggest both methods have complementary value and may scale up well for the analysis of most realistic case studies.

1 Introduction

Trust is a fundamental factor that influences decisions pertaining to human interactions, be they social or economic in nature. Mayer et al. [11] offer a definition of trust as *"... the willingness to be vulnerable, based on positive expectation about the behavior of others."* These expectations of the trustor would be informed by trust signals exchanged with the trustee of a planned interaction. Trust has an economic incentive, it avoids the use of costly measures that guarantee *assurance* in the absence of trust-enabled interaction. We note that assurance is the established means of realizing "IT security". Traditionally, trust signals (e.g. body language) could be observed both in spatial and temporal proximity to a planned interaction. Modern IT infrastructures, however, disembed agents in space and in time from such signals and interaction resources, making it hard to use existing trust mechanics such as those proposed in [17] in this setting [10].

This identifies a need for a *calculus* in which trust and distrust signals can be expressed and aggregated to support decision making in a variety of applications (e.g. financial transactions, software installations, and run-time monitoring of hardware). In our proposed methodology, signals of trust or distrust have no effect in their absence but evaluate to a score in their presence. These scores

may be determined by techniques suitable for the types of signals, e.g. machine learning if signals are features, metrics if signals indicate trustworthiness of IT infrastructures, etc. This then makes it challenging to devise a calculus for combining scores of different types in a manner that articulates the expectations in trust-mediated interactions. Let us give some examples of this.

Trust of an individual in an online transaction will depend, amongst other things, on the monetary value of that transaction, the reputation of the seller, and contextual information such as recommendations from friends. IT infrastructures in highly dynamic and volatile environments such as military operating theatres can no longer be secured in a binary "secure or insecure" manner. They have to react to risks in agile manners [1], suggesting the use of compositional metrics for run-time trust management. Similarly, run-time systems may want to monitor executing code by measuring signals from execution characteristics – such as the threat level of parsed input (e.g. input such as meta-data may serve as an attack surface [19]), the domain of a remote procedure call, etc. – and aggregate such evidence to control execution paths. We refer to [8] for a case study of such execution control in the Scala programming language, where methods are annotated with *expectation blocks* – a precursor of the language Peal [4] – whose aggregation computes what corresponds to the score of a policy set in Peal.

These examples suggest a trust calculus needs to express evidence that is not only rooted in trust (e.g. an asset value), needs to be extensible for domain-specific expressions of signals (e.g. those of a social network), and requires a means of calculating trust from observed signals (e.g. compositional metrics). In [4], such a language Peal was proposed in which signals are abstract predicates whose truth triggers a score, and where score aggregation captures reasoning about levels of trust. In [4], several analyses were also defined that assess if trust calculations perform as expected by specifiers. Verification of trust calculations is thus a key ingredient of such an approach, and the focus of this paper.

We here express the analysis of Peal expressions as constraints that can be analyzed with the SMT solver Z3, and so capture *logical dependencies* of (dis)trust signals. Specifically, we refine and extend the language Peal of [4] to support a richer calculus, we implement analyses proposed in [4] in the SMT solver Z3 on this richer language via two different methods of automated Z3 code generation in PEALT, and we experimentally explore the trade-offs of both methods.

Outline of paper. Section 2 contains background on Peal and the SMT solver Z3. Design and implementation of PEALT are outlined in Section 3. In Section 4, we describe two methods for converting conditions used in analyses into Z3 input. The validation of PEALT via experiments and other activities is reported in Section 5. Section 6 contains related work, and Section 7 concludes the paper.

2 Background

Peal: a **P***luggable* **E***vidence* **A***ggregation* **L***anguage.* The syntax of language Peal is shown in Figure 1. In Peal, a rule *rule* consists of a predicate or signal q_j and

its declared score s_j, has no effect if predicate q_j is false (no signal), and has score s_j as effect otherwise (signal present). Policies *pol* have form as in

$$p_i = op\ ((q_1\ s_1) \ldots (q_n\ s_n))\ default\ s \qquad \text{or} \qquad p_i = op\ ()\ default\ s \tag{1}$$

contain zero or more rules, a default score s, and an aggregation operator *op*. Policy p_i returns default score s if all its rules have false predicates; otherwise it returns the result of applying *op* to all scores s_j of true predicates q_j. The

$$\begin{aligned} op &::= min \mid max \mid + \mid * \\ rule &::= if\ (q)\ score \\ pol &::= op\,(rule^*)\ default\ score \\ pSet &::= pol \mid max(pSet, pSet) \mid min(pSet, pSet) \\ cond &::= th < pSet \mid pSet \leq th \end{aligned}$$

Fig. 1. Syntax of Peal where q ranges over some language of predicates, and *th* and *score* range over real numbers (potentially restricted by domains or analysis methods)

design of Peal is layered as in [4]. Supported aggregation operators are *min* (e.g. for *dis*trust signals), *max* (e.g. for trust signals), + (e.g. for accumulative signals), and $*$ (e.g. for aggregating independent probabilistic evidence). Policies are composed into policy sets (*pSet*) using *max* and *min*. Finally, policy sets are compared to thresholds *th* using inequalities in conditions *cond*. The intuition is that scores and thresholds are real numbers but that some analysis methods may constrain the ranges of said values. The latter is one reason why the PEALT input language under-specifies such design choices. The meaning of policy composition is context-dependent. For example, if a condition $th < min(pS1, pS2)$ is used in support of recommending an action, e.g., then *min* acts as a *pessimistic* composition since the score of any of its arguments may falsify this condition.

SMT solver Z3. Satisfiability modulo theories [5] is supported with robust and powerful tools, that combine the state-of-the-art of deductive theorem proving with that of SAT solving for propositional logic. The SMT solver Z3 has a declarative input language for defining constants, functions, and assertions about them [12]. Figure 2 shows Z3 input code to illustrate that language and its principal analysis directives. On the left, constants of Z3 type Bool and Real are declared. Then an assertion defines that the Boolean constant `q1` means that x is less than $y + 1$, and the next assertion insists that `q1` be true. The directives `check-sat` and `get-model` instruct Z3 to find a witness of the satisfiability of the conjunction of all visible assertions, and to report such a witness (called a model). On the right, we see what Z3 reports for the input on the left: `sat` states that there is a model; other possible replies are `unsat` (there cannot be a model), and `unknown` (Z3 does not know whether or not a model exists).

```
(declare-const q1 Bool)
(declare-const x,y Real)
(assert (= q1 (< x (+ y 1))))
(assert q1)
(check-sat)
(get-model)
```

```
sat
 (model
   (define-fun y () Real 0.0 )
   (define-fun q1 () Bool true )
   (define-fun x () Real 0.0
 )
```

Fig. 2. Left: Z3 input with directives to find and generate a model. Right: Z3 output for this input, a model that makes all input assertions true. (Both edited to save space.)

3 Workflow and input language of PEALT

The tool is rendered as a web application which accepts analysis declarations. The declared analyses can be converted to Z3 input code, followed by calling Z3 and getting feedback on running such code. The tool also allows generation of random declarations or creation of majority-voting condition instances – the latter stress test the explicit method for Z3 code generation described below. A typical workflow of using PEALT would be to generate/write/edit Peal conditions and their analyses, to run these analyses on the Z3 code the tool compiles, and to study the Z3 output to decide whether further such actions are needed. Analyses such as `different? c1 c2` have keywords ending in `?` and list conditions as arguments. Users may specify any number of analyses. Generated Z3 input code will execute each declared analysis in turn using a visibility stack discipline for assertions, as detailed in Section 4.

Example 1. Figure 3 shows an example of PEALT input that may model trust perceptions when downloading a software installation and where a non-matching hash of the download, e.g., is mitigated by the fact that the download was done in a browser `X` that may non-maliciously change file signatures in that process. In the example, both analyses have negative outcome.

Keywords `POLICIES` etc. divide declarations into sorts: policies, policy sets, conditions, domain-specific declarations, and analyses. Keyword *if* is omitted from rules in PEALT input for sake of succinctness. A simple naming construct `name = expr` is used to uniformly bind expressions from the syntactic categories for policies, policy sets, conditions, and analyses to names that can be referenced without any scope restrictions. The syntax for policies, policy sets, and conditions is hoped to be intuitive enough given the definition of Peal. Domain-specific declarations are written in zone `DOMAIN_SPECIFICS`, are expressed directly in Z3 code, and assume that all predicates within rules of declared policies are declared in Z3 input as Z3 type `Bool` already.

We implemented two different ways of generating Z3 input code for declarations entered into PEALT: an explicit and a symbolic one, whose details we will provide below. Intuitively, explicit code generation compiles away any references to numerical values to capture logically – without loss of arithmetic precision – the declared analyses; whereas symbolic code generation statically encodes the

```
POLICIES
b1 = min ((companyDevice 0.1) (uncertifiedOrigin 0.2) (nonMatchingHash 0.2)) default 1
b2 = + ((downloadWithBrowserX 0.1) (useIOS 0.2) (useLinux 0.1) (recentPatch 0.1)) default 0
POLICY_SETS
pSet = min(b1, b2)
CONDITIONS
cond1 = 0.2 < pSet
cond2 = 0.1 < pSet
DOMAIN_SPECIFICS
(declare-const numberOfDaysSinceLastPatch Real)
(assert (= recentPatch (< numberOfDaysSinceLastPatch 7)))
ANALYSES
ana1 = always_true? cond1
ana2 = equivalent? cond1 cond2
```

Fig. 3. Trust perceptions of software download in PEALT, with two analyses

operational semantics of Peal through use of numerical declarations in order for Z3 to be able to reason about all possible dynamic settings. Z3 code generation may produce an exponential blow-up in the explicit method whereas the symbolic one typically finds it harder to reason about multiplication.

Users can specify which code generation method (explicit or symbolic) to use, whether to just compile Z3 input code, and whether to also run it and display results. Users also have the option of downloading the generated Z3 code (as it may be large). For the explicit method, one may just generate results of all analyses in pretty-printed, minimal form. We don't offer this for the symbolic method as its code generation prevents the creation of minimal output models. PEALT is written in Scala 2.10.2 using the Lift web framework. After converting Peal declarations into Z3 input code, PEALT interfaces with the SMT solver Z3 (version 4.3.1) by launching it as an external process via Scala's *ProcessBuilder*.

4 Z3 code generation

Our tool only generates code for conditions that are *used*: i.e. that are declared in the input panel *and* occur in at least one declared analysis as argument. Let `c1` be the declared name of such a condition for declaration `c1 = cond`. We generate Z3 code that declares `c1` as Z3 type `Bool` and adds an assert statement that binds the name `c1` to $\phi[cond]$ via (`assert` (= `c1` $\phi[cond]$)) where $\phi[cond]$ is Z3 code for the logical formula generated for condition `cond`.

The code generated for $\phi[cond]$ explicitly or implicitly lists all signal scenarios that *may* occur if we ignore any logical dependencies between signals. This means that we delegate to our analysis backend, the Z3 SMT solver, the task of only generating scenarios in analyses that are also *logically feasible*. We now describe two methods for generating Z3 code for $\phi[cond]$, starting with the explicit one.

Explicit code generation. For sake of succinctness, we state $\phi[cond]$ here as a formula of propositional logic over predicates and not as Z3 input. The definition of $\phi[cond]$ is given by structural induction over the policy set argument in *cond*, as shown in Figure 4. In the first four equations, *min* and *max* compositions

of policy sets create disjunctions or conjunctions of simpler code generation problems, depending on the type of inequality in *cond*. The first equation, for example, expresses that the minimum of (the score of) two policy sets is less than or equal to a threshold iff that it the case for one of these two policy sets.

The next four equations define auxiliary predicates Q_1 to Q_4 that we can use to specify the remaining cases of conditions that involve only a sole policy. All such conditions first generate the code context for the *non-default* case: in (6), the default score of the sole policy in the condition is compatible with the inequality. Therefore, we generate a disjunction whose first disjunct captures the default case when all predicates of all rules are false, and whose second disjunct captures the non-default case. In (7), the default score of the sole policy is incompatible with the inequality of its condition *cond* and so only the non-default case may apply. Therefore, we generate a conjunction that forces at least one predicate and the formula generated for the non-default case to be true.

It remains to describe the code generation for the non-default case $\phi_{op}^{ndf}[cond]$: in (8), code generation of $\phi_{op}^{ndf}[cond]$ adds a top-level negation and reverts the condition type when Q_3 holds – where $dual(pol \leq th)$ equals $th < pol$ and $dual(th < pol)$ equals $pol \leq th$. This means that we only have to deal with the same inequality type in the remaining cases, that enumerate scenarios. The enumeration process for *max* and *min* in (9) is clear. For example, $\phi_{max}^{ndf}[th < pol]$ is a disjunction of all predicates in *pol* whose scores are strictly larger than th.

The code generation in (10) applies to conditions $pol \leq th$ for $*$ policies *pol*, and conditions $th < pol$ for $+$ policies *pol*. In these cases, we enumerate all minimal scenarios of present signals that make the condition true. These scenarios are minimal in that any smaller subset of present signals won't make the condition true. The code therefore generates a disjunction of monomials where each monomial describes such a minimal scenario. Concretely, as $+$ is monotone and the inequality is $th < pol$, we only need to generate *minimal* index sets X such that the sum of all s_i with i in X is above th. These X are the elements of set $\mathcal{M}_+$ which is computed by $enum_+$ in Figure 5. The Boolean guard in the *while*-loop of $enum_+$ makes use of the partial sums t_i to ensure that recursive calls to $enum_+$ are only made when they will still enumerate at least one new element of $\mathcal{M}_+$. The correctness proof for $enum_+$ is straightforward: all such minimal index sets X are generated in some recursive execution path (completeness), and all enumerated index sets are indeed minimal (soundness, which requires the scores to be sorted in ascending order). Algorithm $enum_*$ enumerates all minimal scenarios in the case of a $*$ policy in $pol \leq th$ and is dual to $enum_+$: it reverts all inequalities for th, lists scores in descending order, and therefore retains the requirement to compute *minimal* index sets. The correctness proof for $enum_*$ is that for $enum_+$ modulo that duality.

Let us discuss what restrictions use of this explicit code generation imposes on the PEALT input language. It requires that all scores within $*$ policies be within $[0, 1]$ so that $*$ is anti-tone; that all scores within $+$ policies be non-negative to get a correct interpretation of *minimal* index sets in $enum_+$; whereas scores within *max* and *min* policies may be any real numbers, since the inequalities

$$\phi[min(pS_1, pS_2) \leq th] \stackrel{\text{def}}{=} \phi[pS_1 \leq th] \vee \phi[pS_2 \leq th] \quad (2)$$

$$\phi[max(pS_1, pS_2) \leq th] \stackrel{\text{def}}{=} \phi[pS_1 \leq th] \wedge \phi[pS_2 \leq th] \quad (3)$$

$$\phi[th < min(pS_1, pS_2)] \stackrel{\text{def}}{=} \phi[th < pS_1] \wedge \phi[th < pS_2] \quad (4)$$

$$\phi[th < max(pS_1, pS_2)] \stackrel{\text{def}}{=} \phi[th < pS_1] \vee \phi[th < pS_2] \quad (5)$$

$$Q_1(pol, cond) \stackrel{\text{def}}{=} (s \leq th, cond = pol \leq th) \vee (th < s, cond = th < pol)$$

$$Q_2(pol, cond) \stackrel{\text{def}}{=} (th < s, cond = pol \leq th) \vee (s \leq th, cond = th < pol)$$

$$Q_3(op, cond) \stackrel{\text{def}}{=} (op \in \{+, max\}, cond = pol \leq th) \vee (op \in \{*, min\}, cond = th < pol)$$

$$Q_4(op, cond) \stackrel{\text{def}}{=} (op = *, cond = pol \leq th) \vee (op = +, cond = th < pol)$$

$$\phi[cond] \stackrel{\text{def}}{=} (\neg q_1 \wedge \cdots \wedge \neg q_n) \vee \phi_{op}^{ndf}[cond] \quad \text{(when } Q_1(pol, cond) \text{ is true)} \quad (6)$$

$$\phi[cond] \stackrel{\text{def}}{=} (q_1 \vee \cdots \vee q_n) \wedge \phi_{op}^{ndf}[cond] \quad \text{(when } Q_2(pol, cond) \text{ is true)} \quad (7)$$

$$\phi_{op}^{ndf}[cond] \stackrel{\text{def}}{=} \neg\phi_{op}^{ndf}[dual(cond)] \quad \text{(when } Q_3(op, cond) \text{ is true)} \quad (8)$$

$$\phi_{max}^{ndf}[th < pol] \stackrel{\text{def}}{=} \bigvee_{i|th<s_i} q_i \qquad \phi_{min}^{ndf}[pol \leq th] \stackrel{\text{def}}{=} \bigvee_{i|s_i \leq th} q_i \quad (9)$$

$$\phi_{op}^{ndf}[cond] \stackrel{\text{def}}{=} \bigvee_{X \in \mathcal{M}_{op}} \bigwedge_{i \in X} q_i \quad \text{(when } Q_4(op, cond) \text{ is true)} \quad (10)$$

Fig. 4. Explicit code generation (recursively): *pol* has form as in (1); predicates Q_1 to Q_4 drive the compilation logic; the computation of sets $\mathcal{M}_{op}$ is detailed in Figure 5.

```
enum+(X, acc, index, op) {
  if (th < acc) { output X; }
  else {
    j = index − 1;
    while ((0 ≤ j) ∧ (th < op(acc, tj))) {
      enum+(X ∪ {j}, op(acc, sj), j, op);
      j = j − 1; }}}
```

```
enum*(X, acc, index, op) {
    if (acc ≤ th) { output X; }
    else {
      j = index − 1;
      while ((0 ≤ j) ∧ (op(acc, tj) ≤ th) {
        enum*(X ∪ {j}, op(acc, sj), j, op);
        j = j − 1; }}}
```

Fig. 5. Left: algorithm $enum_+$ computes $\mathcal{M}_+$ where scores s_i are sorted in ascending order. Right: algorithm $enum_*$ computes $\mathcal{M}_*$ where s_i are sorted in descending order. Initial call context is $(\{\}, 0, n, +)$ for $enum_+$ and $(\{\}, 1, n, *)$ for $enum_*$.

in (9) have the intended meaning for all sign combinations. Z3 code generated for the PEALT input in Figure 3 is shown in Figure 6. PEALT uses the `push` and `pop` directives of Z3 in order to add constraints specific to an analysis onto the top of the assertion visibility stack that Z3 maintains, and to discharge these assertions before turning to the next analysis. The Z3 code generated for analyses is verbatim the same for the symbolic code generation to which we turn next.

```
(declare-const recentPatch Bool)
(declare-const useLinux Bool)
(declare-const uncertifiedOrigin Bool)
(declare-const companyDevice Bool)
(declare-const downloadWithBrowserX Bool)
(declare-const useIOS Bool)
(declare-const nonMatchingHash Bool)
(declare-const cond2 Bool)
(declare-const cond1 Bool)
(assert (= cond1 (and (or (and (not companyDevice) (not uncertifiedOrigin) (not nonMatchingHash))
              (not (or companyDevice uncertifiedOrigin nonMatchingHash)))
              (and (or downloadWithBrowserX useIOS useLinux recentPatch)
              (or (and useIOS recentPatch) (and useIOS useLinux) (and useIOS downloadWithBrowserX)
              (and recentPatch useLinux downloadWithBrowserX))))))
(assert (= cond2 (and (or (and (not companyDevice) (not uncertifiedOrigin) (not nonMatchingHash))
                 (not companyDevice)) (and (or downloadWithBrowserX useIOS useLinux recentPatch)
                 (or useIOS (and recentPatch useLinux) (and recentPatch downloadWithBrowserX)
                 (and useLinux downloadWithBrowserX))))))

(echo "Result of analysis [ana1 = always_true? cond1]:")
(push)
(declare-const always_true_ana1 Bool)
(assert (= always_true_ana1 cond1))
(assert (not always_true_ana1))
(check-sat)
(get-model)
(pop)

(echo "Result of analysis [ana2 = equivalent? cond1 cond2]:")
(push)
(declare-const equivalent_ana2 Bool)
(assert (= equivalent_ana2 (or (and cond1 (not cond2)) (and (not cond1) cond2))))
(assert equivalent_ana2)
(check-sat)
(get-model)
(pop)
```

Fig. 6. Explicitly generated code for input from Figure 3 (hand edited to save space)

Symbolic code generation. This method also binds the name `c1` of declaration `c1` = `cond` to its condition via (`assert` (= `c1` $\phi[cond]$)). But for each policy p_i occurring in *cond*, it also declares a constant $cond\_p_i$ of Z3 type Bool and then generates $\phi[cond]$ as a positive Boolean formula over the constants $cond\_p_i$. This process follows the same logic as for explicit code generation in (2) to (5). For each declared constant `cond_p_i` of Z3 type Bool, it then adds an assert statement (`assert` (= `cond_p_i` $\phi[cond\_p_i]$)) that defines the meaning of $cond\_p_i$. For policies p_i of form as in (1), the code generated is similar to the one of the explicit method when *op* equals *min* or *max* – we refer to [7] for further details.

Let *op* equal $*$ or $+$ and policy p_i occur in at least one condition within some declared analysis. Then the code generation for $\phi[cond\_p_i]$ in Figure 7 trades off the space complexity of enumerating elements in $\mathcal{M}_+$ and $\mathcal{M}_*$ with the time complexity of solving real-valued inequalities in the Z3 SMT solver. For each predicate q_j within p_i, we declare a constant $p_i\_score\_q_j$ of Z3 type Real, and add two assertions that, combined, model that the value of $p_i\_score\_q_j$ is s_j iff q_j is true, and that this value equals the unit of $+$ (respectively, $*$) iff q_j is false. This means that we can precisely model the *effect* of the non-default case (when at least one q_j is true) by aggregating all values $p_i\_score\_q_j$ with *op*, and by comparing that aggregated result to the threshold in the specified manner ($<$ or $\geq$). Crucially, the values of $p_i\_score\_q_j$ for predicates that happen to be false won't contaminate this aggregated value as they are units for operator *op*.

The encoding for symbolic code generation is therefore *linear* in the size of *cond*. Using this encoding, we can now express $\phi[cond\_p_i]$ in Z3 by directly encoding the "operational" semantics of $cond\_p_i$: either the default score satisfies the inequality and all policy predicates are false, or at least one policy predicate is true and the aggregation of all values $p_i\_score\_q_j$ with *op* satisfies the inequality. These Z3 declarations and expressions are stated in Figure 7.

```
(declare-const p_i_score_q_j Real)
(assert (implies q_j (= s_i p_i_score_q_j)))
(assert (implies (not (= <unit> p_i_score_q_j)) q_j))

(or (and (cop th s) (not (or q_1 ... q_n)))
    (and (or q_1 ... q_n)
         (cop th (op p_i_score_q_1 ... p_i_score_q_n))))
```

Fig. 7. Top: declarations for $p_i\_score\_q_j$ where `s_j` is s_j, and <`unit`> is `0.0` for $+$ policies p_i and `1.0` for $*$ policies p_i. Bottom: Z3 code for $\phi[cond\_p_i]$ for the first case in (1); comparison operator *cop* is $<$ for $th < p_i$ or $\geq$ for $th \geq p_i$, and `th` denotes th.

The symbolic code generation described above imposes no restrictions on the ranges of scores s_i. PEALT allows us to replace `s_i` with an arithmetic expression such as any real numbers c, real variables x, or products thereof $(c \cdot x)$.

Analyses. Analysis `implies`? checks whether the first condition logically implies the second one, which is a form of policy refinement. Analyses `always_false`? and `satisfiable`? are "equivalent" but capture different intent of the user, ditto for analysis `equivalent`? versus analysis `different`?. A typical use of analysis `different`? is to check whether conditions differ for $0.5 < pSet$ and $0.6 < pSet$, i.e. whether $pSet$ is sensitive to the increase of threshold value from 0.5 to 0.6.

Specification of domain specifics. Users may add domain-specific constraints or knowledge as Z3 code within zone `DOMAIN_SPECIFICS`: e.g. to declare variables with which one can then define the exact meaning of predicates used in rules

(e.g. as a means of adding parameters to signals), to encode required properties of the modeling domain, and to perhaps add assertions that guide the search of a model of some analysis. The use of raw Z3 code means that any code generation method will simply copy and paste this code into the generated Z3 input code. We realize that our decision to automatically generate Z3 declarations of all variables occurring in rules might confuse novice users, though, when they try to declare these as Z3 types within zone `DOMAIN_SPECIFICS` explicitly.

Witness generation. For each declared analysis, Z3 will try to decide it when running PEALT. If the Z3 output is `unsat`, then we know that there is no witness to the query – e.g. for `always_true`? this would mean that Z3 decides that the condition cannot be false, and so the answer is "yes, always true". If the Z3 output is `sat`, then we report the correct answer (e.g. for `always_true`? we say "no, not always true") and generate supporting evidence for this answer. For explicit code generation, the generated models tend to be very short (few crucial truth values of predicates q_i and supporting values of variables used to define these q_i if applicable). PEALT can post-processes this raw Z3 output to extract this information in pretty-printed form, an example thereof is seen in Figure 8. For symbolic code generation, model list truth values for *almost all* declared predicates q_i that occur in at least one $*$ or $+$ policy. The reason for this seems to stem from the assertions we declare for variables `p_i_score_q_j` in Figure 7. We mean to investigate how to shorten such evidence in future work.

```
Result of analysis [ana1 = always_true? cond1]
cond1 is NOT always true
For example, when useLinux is true, recentPatch is true,
nonMatchingHash is true, companyDevice is false
```

Fig. 8. Sample of pretty printed evidence for satisfiability witness computed from explicitly generated code for `always_true`? from Figure 3 (hand edited to save space).

Execution constraints. To summarize, explicit code generation of policies within analyzed conditions requires that no $*$ policy has scores outside $[0, 1]$ and that no other policy has negative or non-constant scores. For symbolic code generation, we only have to ensure that *min* and *max* policies have constant scores (negative ones are allowed), and we mean to lift the latter restriction in future work.

5 Validation

We report experimental results for code generation methods and execution of generated code on random and non-random analyses. We also discuss other tool validation activities we conducted. All experiments were run on a test server with two, 6-core, Intel E5 CPUs running at 2.5GHz and 48G of RAM.

Non-random benchmark. We use condition $0.5 < p_{mv(n)}$ with + policy $p_{mv(n)}$, default score 0, and n many rules each with score $1/n$. The condition is true when more than half of the predicates are true ("majority voting"). There are no logical dependencies of predicates in $p_{mv(n)}$ and the size of $\mathcal{M}_+$ is exponential in n. We can generate explicitly Z3 input code for values of n up to 27 (when code takes up half a gigabyte), and code generation takes more than five minutes for n being 23. By comparison, we could generate symbolically such code and verify that this condition is true, within five minutes each, for n up to 49408.

Randomly generated analyses. We also implemented a feature

$$randPeal\ n, m_{min}, m_{max}, m_+, m_*, p, th, \delta$$

that randomly generates a policy set $pSet$, two conditions $th < pSet$ and $th + \delta < pSet$ and analyses the first one with `always_true`?, the second one with `always_false`?, and then applies `different`? to both conditions. Predicates are randomly selected from a pool of p many predicates (with $n \leq p$). Scores are chosen from $[0, 1]$ uniformly at random. In $pSet$, there are n policies for each operator op of Peal (i.e. $4n$ policies in total) and each op policy has m_{op} many rules. For the maximal k with $2^k \leq 4n$, we combine 2^k policies using alternating max and min compositions on their full binary parse tree; the result is further composed with the remaining $4n - 2^k$ policies (if applicable) by grouping these in min pairs, and by adding these pairs in alternating min and max compositions to the binary policy tree. This stress tests policy composition above and beyond what one would expect in practical specifications.

We then conducted three experiments that share an execution and termination logic: experimental input to $randPeal$ has only one degree of freedom and we use unbounded binary search to see (within granularity of 10 and for five randomly generated condition pairs) whether both code generation methods can generate Z3 code within five minutes, and whether Z3 can perform each analysis within that same time frame. If this fails for one of these condition pairs, we stop binary expansion and go to a bisection mode to find the boundary.

Experiment 1 picks for operator min input headers $1, x, 1, 1, 1, 3x, 0.5, 0.1$ so it explores how many (x) rules a sole min policy can handle within five minutes. The same evaluation is done for the other three operators. We also investigated a variant of this experiment – Exp 1 (DS) – for which we also add as many assertions as there are declared predicates in the conditions, as described in [7]. This uses a function `calledBy` that models method call graphs with at most one incoming edge (using a `forall` axiom in Z3 code) and declares a third of these predicates to mean that a specific method called. The other two thirds define predicates as linear inequalities between real, respectively integer, variables (which may stem from method input headers) – please see [7] for details.

Experiment 2 picks for operator min the input headers $n, c, 1, 1, 1, 3c, 0.5, 0.1$ where c equals $x/10$ for the boundary value of x found in Experiment 1. We here explore how many min policies we can handle for a sizeable number of rules. The same evaluation is done for the other three operators. Experiment 3 picks

for operator *min* input headers $n, n, 1, 1, 1, 3n, 0.5, 0.1$ so that we explore how many (the n) *min* policies with the same number of rules we can handle within five minutes. The same evaluation is done for the other three operators.

Results of these experiments are displayed in Figure 9. In their discussion we need to recognize that random analyses can have very different analysis times for the same configuration type. So a termination "boundary" does not mean that we cannot verify larger instances within five minutes, it just means that we encountered an instance at the reported boundary that took longer than that.

Exp 1	ex min	sy min	ex max	sy max	ex *	sy *	ex +	sy +
rules	1867904	1802240	2101248	2162688	120	16	144	5784
code	26s	20s	32s	22s	5s	0.1s	14s	0.6s
Z3	110s	181s	74s	132s	48s	3s	72s	133s

Exp 1 (DS)	ex min	sy min	ex max	sy max	ex *	sy *	ex +	sy +
rules	8064	6280	6544	7240	136	16	128	1848
code	0.9	0.8	0.8s	0.8	8s	0.1s	1s	1s
Z3	133s	88s	136s	150s	60s	14s	40s	91s

Exp 2	ex min	sy min	ex max	sy max	ex *	sy *	ex +	sy +
pol,rul	48,186790	56,180224	40,210124	56,216268	65888,12	4192,2	17488,14	24,578
code	264s	76s	169s	87s	279s	84s	277s	0.8s
Z3 time	438s	205s	44s	249s	4s	108s	2s	160s

Exp 3	ex min	sy min	ex max	sy max	ex *	sy *	ex +	sy +
pol=rul	2128	2552	2136	2936	88	16	96	160
code	271s	71s	293s	99s	85s	0.2s	160s	1s
Z3	8s	63s	8s	120s	17s	144s	26s	23s

Fig. 9. Experimental results: columns show code generation method ("ex"plicit or "sy"mbolic) and operator; rows show number of rules for policies of chosen operator in analyses, time (rounded to seconds) to generate Z3 code, and time to execute Z3 code.

In the first experiment, Z3 code generation seems faster than execution of that Z3 code. We also see that up to two million rules can be handled for *min* and *max* for both code generation methods within two minutes. For $*$, explicit code generation seems to be one order of magnitude better than symbolic code generation, although the Z3 execution in the latter case appears to be faster. For $+$, on the other hand, symbolic code generation now seems to be an order of magnitude better than the explicit one – handling thousands of rules in just over two minutes. When we add the domain-specific constraints in Exp 1 (DS), we notice that *min* and *max* can only handle about seven-thousand rules in a similar amount of time (compared to two million beforehand). The results for $*$ for both methods and for $+$ for explicit code generation seem about the same

as without domain-specific constraints. But + now only can handle less than two-thousand rules for symbolic code generation. In the second experiment, the number of rules used for *max* and *min* is about two-hundred thousand. We can deal with about fifty policies with that many rules within five minutes, noting that code generation now takes more time. It is noteworthy that explicit code generation can handle over sixty-thousand $*$ policies with 12 rules each, but that this drops to less than twenty-thousand + policies; the symbolic approach does not scale that well in comparison. In the third experiment, both methods can handle between two to three thousand policies with that many rules for *max* and *min*. For operators $*$ and +, the explicit method spends most of its time in code generation whereas the symbolic one spends the bulk of its time in Z3 execution. For operator $*$, explicit code generation is still about an order of magnitude better whereas for + it is not significantly better.

Ideally, we would like to extend these experiments to larger data points. But such an attempt quickly reaches the memory boundary of our powerful server in explicit code generation. We also believe that practical case studies would not use more than a few dozen or hundreds of rules for each + and $*$ policy declared, and so both approaches may actually work well then.

Software validation and future work. We have not yet encountered a Z3 output `unknown` for PEALT analyses, although this is easy to achieve by adding complex constraints as domain specifics. We validated both code generation methods by running them side by side on randomly generated analyses and checking whether they would produce conflicting answers (`unsat` and `sat`). During the development of PEALT, we encountered a few of these conflicts which helped to identify implementation bugs. Of course, this does not mean what we proved the correctness of our Z3 code generator (written in Scala), and doing so would be unwise as this generator will evolve with the tool language. Therefore, we want to independently verify the evidence computed by Z3, in future work. This will also verify that no `double` rounding errors in Z3 corrupted analysis outcomes. In future work, we also want to understand whether we can construct proofs for outputs `unsat` such that these proofs are meaningful for the analyses in question.

6 Related work

The language in Figure 1 extends that in [4]: it supports policies without rules, $*$ policies, negative and non-constant scores for symbolic code generation, and logical dependencies of predicates q_i within PEALT. The symbolic code generation in PEALT uses the same enumeration process for + and $*$ on *minimal* index sets (and not maximal ones as in [4]). PEALT implements most analyses of [4] *with* logical dependencies, leaving more complex ones of [4] for future work.

The determination of scores is a fundamental concern in our approach, and where PEALT is meant to provide confidence in such scorings and their implications. The process of arriving at scores depends on the application domain, we offer two examples thereof from the literature. TrustBAC [3] extends role-based

access control with levels of trust, scores in $[-1, 1]$, that are bound to roles in RBAC sessions. These levels are derived from a trust vector that reflects user behavior, user recommendations, and other sources. No analysis of these levels and their implications is offered. In [16], we see an example of how a sole score may reflect the integrity of an information infrastructure, as a formula that accounts for known vulnerabilities, threats that can exploit such vulnerabilities, and the likelihood for each vulnerability to exist in the given infrastructure. We should keep in mind that any such metrics are heuristics, and so it is important to analyze their impact on decision making, especially if other factors also influence such decisions. PEALT allows us, in principle, to conduct such analyses. Extant work enriches security elements with quantities, e.g. credential chains [18], security levels [15], trust-management languages [2], reputation [9], and combinations of reputation and trust [13, 14]. But we are not aware of substantial tool support for analyzing the effect of such enrichments when combined with other aspects of evidence. Shinren [6] offers the ability to reason about both trust and distrust explicitly and in a declarative manner, with the support of priority composition operators for layers of trust and distrust. Although Peal is in principle expressive enough to encode most of this functionality, doing so would not constitute good engineering practice: this is a good example for when conditions of Peal would be expressions to be composed in upstream languages such as Shinren.

7 Conclusions

We have created a tool PEALT in which one can study different mechanisms of aggregating numerical trust evidence. We extended the policy-composition language Peal of [4] and modified the generation of verification conditions reported in [4] for Peal conditions to make them dischargeable with an SMT solver. We proposed two different means of generating such verification conditions and discussed both conceptual and experimental advantages and disadvantages of such methods. The explicit method compiles away any references to numerical values and so arrives at a purely logical formulation. The price for this may be an explosion in the length of the resulting formula and in the restriction of score ranges for certain policy composition operators (e.g. multiplication). The symbolic method creates formulas with only linear size in the conditions but shifts the computational burden to Z3 and its reasoning about linear arithmetic. Both methods delegate to Z3 logical feasibility checks of trust scenarios discovered in analyses. Our current PEALT prototype supports verification of policy refinement, vacuity checking, sensitivity analysis of thresholds in conditions, and non-constant scores (for symbolic code generation) to express metrics. We think PEALT is a good example of the benefits that can be gained by connecting to a powerful back-end such as the SMT solver Z3 for analyses. The version of the source code used in this paper is available on `https://bitbucket.org/jimhkuo/pealt`.

Acknowledgments. We thank Jason Crampton and Charles Morisset for very fruitful discussions on PEALT, anonymous reviewers for helpful comments, and Intel® Corporation for funding this work in its *Trust Evidence* research project.

References

1. Announcement of Cybersecurity Collaborative Research Alliance. Press Release, US Army Research Laboratory (15 October 2013)
2. Bistarelli, S., Martinelli, F., Santini, F.: A semantic foundation for trust management languages with weights: An application to the RT family. In: ATC. pp. 481–495 (2008), Springer.
3. Chakraborty, S., Ray, I.: TrustBAC: integrating trust relationships into the RBAC model for access control in open systems. In: Proceedings of the eleventh ACM symposium on Access control models and technologies. pp. 49–58. SACMAT '06, ACM, New York, NY, USA (2006)
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. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM 54(9), 69–77 (Sep 2011), ACM Press.
6. Dong, C., Dulay, N.: Shinren: Non-monotonic trust management for distributed systems. In: IFIPTM. pp. 125–140 (2010), Springer.
7. Huth, M., Kuo, J.H.P.: PEALT: A reasoning tool for numerical aggregation of trust evidence. Tech. Rep. 2013/7, Imperial College London, Department of Computing (2013), ISSN 1469-4166 (Print)
8. Huth, M., Kuo, J.H.P.: Towards verifiable trust management for software execution - (extended abstract). In: Proc. of TRUST. pp. 275–276 (2013), Springer.
9. Jøsang, A., Ismail, R.: The beta reputation system. In: Proceedings of the 15th Bled Conference on Electronic Commerce. Bled, Slovenia (17-19 June 2002)
10. Kirlappos, I., Sasse, M.A., Harvey, N.: Why trust seals don't work: A study of user perceptions and behavior. In: TRUST. pp. 308–324 (2012), Springer.
11. Mayer, R., Davis, J., Schoorman, F.D.: An integrative model of organizational trust. Academy of Management Review 20(3), 709–734 (1995), Academy of Management.
12. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340 (2008), Springer.
13. Mui, L.: Computational Models of Trust and Reputation: Agents, Evolutionary Games, and Social Networks. Ph.D. thesis, Massachusetts Institute of Technology (2002)
14. Muller, T., Schweitzer, P.: On beta models with trust chains. In: IFIPTM. pp. 49–65 (2013), Springer.
15. Ni, Q., Bertino, E., Lobo, J.: Risk-based access control systems built on fuzzy inferences. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security. pp. 250–260. ASIACCS '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1755688.1755719
16. Nurse, J.R.C., Creese, S., Goldsmith, M., Rahman, S.S.: Supporting human decision-making online using information-trustworthiness metrics. In: HCI (27). pp. 316–325 (2013), Springer.
17. Riegelsberger, J., Sasse, M.A., McCarthy, J.D.: The mechanics of trust: A framework for research and design. Int. J. Hum.-Comput. Stud. 62(3), 381–422 (2005)
18. Schwoon, S., Jha, S., Reps, T.W., Stubblebine, S.G.: On generalized authorization problems. In: CSFW. pp. 202–218 (2003), IEEE Computer Society.
19. Shapiro, R., Bratus, S., Smith, S.W.: "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In: Proceedings of the 7th USENIX Workshop on Offensive Technologies (WOOT 13). USENIX (2013), 12 pages