PEALT: A Reasoning Tool for Numerical Aggregation of Trust Evidence [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

4 October 2013

Abstract

We present a tool that supports the understanding and validation of mechanisms that numerically aggregate trust evidence – which may stem from heterogenous sources such as geographical information, reputation, and threat levels. The tool is based on a policy composition language Peal [3] and can declare Peal expressions and intended analyses of such expressions as input. The analyses include vacuity checking, sensitivity analysis of thresholds, and policy refinement. We develop and implement two methods for generating verification conditions for analyses, using the SMT solver Z3 as backend. One method is explicit and space intense, the other one is symbolic and so linear in the analysis expressions. We experimentally investigate this space-time tradeoff by observing the Z3 code generation and its running time 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 sufficiently 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. [7] 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 is hard to use existing trust mechanics such as those proposed in [11] in this setting [6].

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). Thus a language of such a calculus needs plugs for domain-specific expressions of signals, and means of calculating trust from observed signals. In [3], we proposed such a language where signals are abstract predicates that – when true – trigger a score, and where the aggregation of such scores captures reasoning about levels of trust. That paper also introduced several analyses that assess whether used trust calculations do indeed

[1] Please cite this research note as *Michael Huth, Jim Kuo, and Jason Crampton. PEALT: A Reasoning Tool for Numerical Aggregation of Trust Evidence.* Technical Report 2013/7, Department of Computing, Imperial College London, ISSN 1469-4174, 2013.

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

Figure 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)

capture desired intent (e.g. as represented in an organizational risk posture say). Verification of trust calculations is thus a key ingredient of such an approach, and the focus of this paper.

To realize this, we combine methods such as logical synthesis, enumeration algorithms, and SMT solving for the generation and analysis of verification conditions in this setting of numerically aggregated evidence. Specifically, the contributions of this paper are as follows: (i) we refine and extend the language of [3] (Peal) to support a richer calculus, (ii) we implement analyses proposed in [3] in the SMT solver Z3 on this richer language and so capture dependencies in trust signals, (iii) we validate our tool (called PEALT) through detailed experiments, and (iv) we provide two different means of Z3 code generation in PEALT – one of which significantly extends the scope of analyses to aggregations that allow for weighted sums as used, e.g., in information security metrics.

Outline of paper. In Section 2, we provide background on the target language 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 compilation of Z3 code for analyses and the reporting and validation of witness information is discussed in Section 5. The validation of PEALT via experiments and other activities is reported in Section 6. Section 7 contains related work, and Section 8 concludes the paper.

2 Background

Peal: a Pluggable Evidence Aggregation Language. The syntax of Peal is defined in Figure 1. Rules *rule* are essentially pairs of predicates q_j and their associated scores s_j. The meaning of a rule (q_j, s_j) is that is has no effect if its predicate q_j is false (no signal), and has score s_j (signal present) as effect otherwise. Policies (*pol*) p_i have form as in (1) and contain zero or more rules, a default score s, and an aggregation operator *op*. Policy p_i returns the default score 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.

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

The design of Peal is layered. The supported aggregation operators are *min* (for pessimistic views), *max* (for optimistic views), + (for accumulative views), and $*$ (e.g. for aggregating independent probabilistic evidence). Policies are composed into policy sets (*pSet*) using *max* and *min* with the same optimistic and pessimistic views as for rule composition. 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.

```
(declare-const q1 Bool)
(declare-const x Real)
(declare-const 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
 )
```

Figure 2: Left: sample Z3 input code with a directive to find and to generate a model. Right: raw Z3 output for the left input code (edited to save space), saying that the conjunction of all assertions is satisfiable, and supporting this claim with a model.

We note that conditions *cond* may well act as arguments in upstream languages, e.g. in order to inform access-control decisions in IT systems.

SMT solver Z3. Satisfiability modulo theories [4] 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. If we want to know, e.g., whether a Peal condition *cond* is always true (a form of *vacuity checking*), our tool allows us to declare and name such an analysis and to generate Z3 input code that, when run, will try to answer this whilst reflecting logical dependencies of predicates.

Z3 has a declarative input language for defining constants, functions, and assertions about them [8]. Figure 2 shows Z3 input code to illustrate that language and its key 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).

3 The tool PEALT: its input language based on Peal

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 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.

The format for declarations is indicated in Figure 3. This example may model risk 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 the browser Chrome (which non-maliciously changes files 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. Note that the keyword *if* is omitted from rules in PEALT for sake of succinctness. A simple naming construct `name = expr` is used to bind policies, policy sets, conditions, and analyses to names which we can refer to without any scope restrictions. The syntax

```
POLICIES
b1 = min ((companyMachine 0.1) (uncertifiedOrigin 0.2) (nonMatchingHash 0.2)) default 1
b2 = + ((usedChromeForDownload 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 30)))
ANALYSES
ana1 = always_true? cond1
ana2 = different? cond1 cond2
```

Figure 3: Sample input to PEALT with two analyses.

for policies, policy sets, and conditions is hoped to be intuitive enough given the definition of Peal. Domain-specific declarations are written directly in Z3 and assume that all predicates within rules of declared policies are declared in Z3 input as Z3 type `Bool` already.

Analyses such as `different? c1 c2` have keywords ending in `?` and list conditions as arguments. Users may specify any number of analyses and our generated Z3 input code will investigate each declared analysis in turn, as detailed below.

We implemented two different ways of generating Z3 input code for declarations entered into the tool: *explicit synthesis* and *symbolic synthesis*, whose details we will provide below. Intuitively, explicit synthesis compiles away any references to numerical values to capture – without loss of precision – the logical essence of the declared analyses; whereas symbolic synthesis statically encodes the 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 explicit synthesis but we will see below that this method also has its advantages.

Users can specify which synthesis method (explicit or symbolic) to use and whether to just compile Z3 input code or whether to also run it and display results. For explicit synthesis, users also have the option of downloading the generated Z3 code (as it may be large) or to just generate results of all analyses in pretty-printed, minimal form. For symbolic synthesis, we don't offer pretty printing as here models specify truth values of *all* predicates and so we cannot, at present, compute minimal diagnostic information.

PEALT is written in Scala 2.10.2 using the Lift web framework. After converting Peal declarations into Z3 input code, it interfaces with the SMT solver by launching Z3 (version 4.3.1) as an external process via Scala's *ProcessBuilder*.

4 Synthesis of verification conditions

Our tool will only synthesize 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 synthesized for condition `cond`. We now describe two

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

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

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

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

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

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

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

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

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

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

$$\phi^{ndf}_{op}[cond] \stackrel{\text{def}}{=} \neg\phi^{ndf}_{op}[dual(cond)] \qquad (Q_3(op, cond) \text{ true}) \quad (8)$$

$$\phi^{ndf}_{op}[cond] \stackrel{\text{def}}{=} \bigvee_{X \in \mathcal{M}_{op}} \bigwedge_{i \in X} q_i \qquad (Q_4(op, cond) \text{ true}) \quad (9)$$

$$\phi^{ndf}_{max}[th < pol] \stackrel{\text{def}}{=} \bigvee_{i | th < s_i} q_i \qquad \phi^{ndf}_{min}[pol \leq th] \stackrel{\text{def}}{=} \bigvee_{i | s_i \leq th} q_i \quad (10)$$

Figure 4: Recursive definition of explicit synthesis: *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.

methods for generating Z3 code for $\phi[cond]$, starting with the explicit one.

Explicit synthesis. 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 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 synthesis problems, depending on the type of inequality in *cond*. The next four equations define predicates Q_1 to Q_4 that drive the compilation logic of the remaining induction. In (6–7), synthesis for conditions that contain a sole policy is reduced to the synthesis of its non-default case $\phi^{ndf}_{op}[cond]$ (when at least one predicate of a rule is meant to be true). In (6), the default score is compatible with the inequality so this reduction creates a disjunction whose first disjunct captures the default case when all predicates of all rules are false. In (7), the default score is incompatible with the inequality of *cond* and so only the non-default case may apply; we wrap that case into a conjunction that forces that at least one predicate be true (that conjunct is needed as our validation testing of this implementation confirmed).

In (8), synthesis of $\phi^{ndf}_{op}[cond]$ adds a top-level negation and reverts the condition type (where $dual(pol \leq th) = th < pol$) and $dual(th < pol) = pol \leq th$)); that way we can use the *same* (potentially exponential) enumeration process in (9) for $+$ and $*$. The enumeration process for *max* and *min* in (10) is different and linear in the number of rules, e.g. $\phi^{ndf}_{max}[th < pol]$ is a disjunction of all predicates in *pol* whose scores are strictly larger than *th*.

The synthesis in (9) generates a disjunction of monomials that can be represented as an index

```
enumMon(X, acc, index, op) {
  if (th < acc) { output X; }
  else {
    j = index − 1;
    while ((0 ≤ j) ∧ (th < op(acc, t_j))) {
      enumMon(X ∪ {j}, op(acc, s_j), j, op);
      j = j − 1; }}}
```

```
enumAnt(X, acc, index, op) {
  if (acc ≤ th) { output X; }
  else {
    j = index − 1;
    while ((0 ≤ j) ∧ (op(acc, t_j) ≤ th)) {
      enumAnt(X ∪ {j}, op(acc, s_j), j, op);
      j = j − 1; }}}
```

Figure 5: Left: pseudo-code for algorithm $enumMon$ computing $\mathcal{M}_{op}$ for monotone op where scores s_i are sorted in ascending order. Right: pseudo-code for $enumAnt$ computing $\mathcal{M}_{op}$ for anti-tone op where s_i are sorted in descending order. In both cases, initial call context is $(\{\}, unit, n, op)$, $unit$ is the unit of op, and t_i is $op(\{s_1, \ldots, s_i\})$.

set of their predicates. Since + 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 $enumMon$ in Figure 5. The Boolean guard in the *while*-loop of $enumMon$ makes use of the partial sums t_i to ensure that recursive calls to $enumMon$ are only made when they will still enumerate at least one new element of $\mathcal{M}_+$. The correctness proof for $enumMon$ 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 $enumAnt$ is dual to $enumMon$: it reverts all inequalities for th, lists scores in descending order, but retains the requirement to compute *minimal* index sets. The correctness proof for $enumAnt$ is that for $enumMon$ modulo that duality. We stress that both algorithms remain to be correct for more general operators: $enumMon$ for *monotone* operators (where $X \subseteq Y$ implies $op\{s_i \mid i \in X\} \leq op\{s_j \mid j \in Y\}$); and $enumAnt$ for *anti-tone* operators (where $X \subseteq Y$ implies $op\{s_i \mid i \in X\} \geq op\{s_j \mid j \in Y\}$). Thus we may implement additional such operators in the same manner in PEALT with little overhead.

Let us discuss what restrictions use of this explicit synthesis imposes on the input language. Explicit synthesis requires that (i) all scores within $*$ policies be within $[0, 1]$ so that $*$ is anti-tone; (ii) all scores within + policies be non-negative to get a correct interpretation of *minimal* index set in $enumMon$; but (iii) scores within *max* and *min* policies may be any real numbers, since the inequalities in (10) have the intended meaning for all sign combinations. The above restrictions do not apply to symbolic synthesis, to which we turn next.

Symbolic synthesis. This method also binds the name `c1` of declaration `c1 = cond` to its condition via (`assert` (= `c1` $\phi[cond]$)). But the definition of $\phi[cond]$ changes: for each policy p_i occurring in *cond*, we declare a constant *cond_p_i* of Z3 type Bool and then generate $\phi[cond]$ as a positive Boolean formula over the constants *cond_p_i*. This generation process is depicted in Figure 6 where *cond* is of form *th cop pSet* or *pSet cop th* with comparison operator *cop*. For example, if *pSet* is just a policy p_i, then $\phi[cond]$ is just *cond_p_i*; and if *cond* is $th < max(pS_1, pS_2)$, then $\phi[cond]$ is a disjunction of $\phi[th < pS_1]$ and $\phi[th < pS_2]$.

For each declared constant `cond_p_i` of Z3 type Bool, we then add an assert statement (`assert` (= `cond_p_i` $\phi[cond\_p_i]$)) that defines the meaning of *cond_p_i*. It therefore remains to describe how the formula $\phi[cond\_p_i]$ is being synthesized symbolically. For p_i of form as in (1) we depict one case of code generated for $\phi[cond\_p_i]$ when *op* equals *max* or *min* in Figure 7. This is a variant of the explicit synthesis for these operators, but it makes explicit how empty conjunctions and disjunctions cause code optimizations, and it shows that we now explicitly code up the

$\phi[cond] = genSym(cop, pSet)$ % *cond* of form *th cop pSet* or of form *pSet cop th*
$genSym(cop, pS)$ {
if $(pS == p_i)$ { *output* 'cond_p_i'; }
if $(((pS\ is\ max(pS_1, pS_2)) \wedge (cop\ is\ <)) \vee (pS\ is\ min(pS_1, pS_2)) \wedge (cop\ is\ \leq))$ {
output '(or ' $genSym(cop, pS_1)$ $genSym(cop, pS_2)$ ')'; }
if $(((pS\ is\ min(pS_1, pS_2)) \wedge (cop\ is\ <)) \vee (pS\ is\ max(pS_1, pS_2)) \wedge (cop\ is\ \leq))$ {
output '(and ' $genSym(cop, pS_1)$ $genSym(cop, pS_2)$ ')'; }
}

Figure 6: Generation of positive Boolean Z3 formula (in red in ' ') that defines the synthesis of *cond* in terms of the syntheses of *cond_*p_i for policies p_i occurring in *cond*.

Let *op* be *max* and let *cond* be $th < p_i$. Then we set

```
(assert (= cond_p_i
           (or (and (< th s) (not (or q1 q2 ... qn)))
               (or qi1 qi2 ... qik))))
```

where q_{i_j} are those precidates in p_i whose scores s_{i_j} satisfy $th < s_{i_j}$. If there are no such predicates in *p_i*, then the above is replaced with

```
(assert (= cond_p_i
           (and (< th s) (not (or q1 q2 ... qn)))))
```

Figure 7: One case of synthesized Z3 input code for *cond_*p_i for policies p_i as in (1)

inequalities for the default score, e.g. as in (< th s).

We now specify how to generate $\phi[cond\_p_i]$ if *op* equals $*$ or $+$ and policy p_i occurs in at least one condition within some declared analysis (see Figure 8). This is where we trade 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_i, we declare a constant $p_i\_score\_q_i$ of Z3 type Real, and add two assertions that, combined, model that the value of $p_i\_score\_q_i$ is s_i iff q_i is true, and that this value equals the unit of $+$ (respectively, $*$) iff q_i is false. This means that we can precisely model the *effect* of the non-default case (when at least one q_i is true) by aggregating all values $p_i\_score\_q_i$ with *op*, and by comparing that aggregated result to the threshold in the specified manner ($<$ or $\geq$). Crucially, the values of $p_i\_score\_q_i$ for predicates that happen to be false won't contaminate this aggregated value as they are units for operator *op*.

The encoding for symbolic synthesis 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 $p_i\_score\_q_i$ with *op* satisfies the inequality. These Z3 declarations and expressions are stated in Figure 8.

The symbolic synthesis specified above imposes no restrictions on the ranges of scores s_i, they may be any machine representable real numbers that Z3 can handle. For *explicit* synthesis of $+$ policies, PEALT allows us to replace s_i with an arithmetic expression such as any real numbers c, real variables x, or products thereof $(c \cdot x)$. For $+$ policies, we can in this manner express metrics such as $th < \sum_{i=1}^{n} c_i \cdot x_i$, where the semantics of Peal policies gives us the additional ability to "turn

```
(declare-const p_i_score_q_i Real)
(assert (implies q_i (= s_i p_i_score_q_i)))
(assert (implies (not (= <unit> p_i_score_q_i)) q_i))

(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))))
```

Figure 8: Top: declarations for $p_i\_score\_q_i$ where `s_i` is s_i, 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 such policies; the comparison operator *cop* is $<$ for $th < p_i$ or $\geq$ for $th \geq p_i$, and `th` denotes th.

```
ANALYSES
a1 = always_true? c1      (push)
a2 = always_false? c1     (declare-const <analysisType>_<declaredName> Bool)
a3 = satisfiable? c1      ...
a4 = equivalent? c1 c2    (check-sat)
a5 = different? c1 c2     (get-model)
a6 = implies? c1 c2       (pop)
```

Figure 9: Left: types of analyses currently supported in PEALT. Right: common Z3 input frame generated for each line of code on the left.

off" some of these summands by making their predicates be false. We note that we did not make the definition of $\phi[cond\_p_i]$ in Figure 8 the basis for all policy operators *op* in explicit synthesis since Z3 does not seem to support *max* and *min* in an n-ary version, and doing so may incur a performance penalty over explicit synthesis.

5 Analyses

We have seen how the two methods of synthesis above generate declarations of constants and assert statements in Z3. Now, we describe how we generate code that implements declared analyses. Figure 9 lists examples of the analyses currently supported. For example, the analysis `implies`? checks whether the first condition logically implies the second one, which is a form of policy refinement. Names `a1` etc. will be used to refer to these analyses in code generation. Analyses `always_false`? and `satisfiable`? are "equivalent" but capture different intent of the user, ditto for `equivalent`? versus `different`?.

We explain the code generation for analyses for the first declaration on the left in Figure 9, this is shown in Figure 10; this generation is the same for both synthesis methods, as generated code only refers to the condition names and so is independent of how these conditions are constrained or generated. Therefore PEALT can and does enrich condition expressions of Figure 1 with propositional operators on *names* of conditions, as in `!c1 && c2`, to be used in analyses.

Each analysis generates the same code frame as shown on the right of Figure 9, where <`analysisType`> is the string of the analysis without the question mark (e.g. `always_true`), and `declaredName` is the name to which that instance of the analysis was bound (e.g. `a1`). This creates a unique internal name for a constant of Z3 type Bool that can then be instrumented with

```
(push)
(declare-const always_true_a1 Bool)
(assert (= always_true_a1 c1))
(assert (not always_true_a1))
(check-sat)
(get-model)
(pop)
```

Figure 10: Z3 input code for the first declaration made on the left in Figure 9.

```
Result of analysis [name2 = always_false? cond1]
cond1 is NOT always false
For example, when q2 is true, q1 is false, y is (- 1.0), and x is 0.0
```

Figure 11: Sample of pretty printed evidence for satisfiability witness generated by explicit synthesis for an instance of `always_false`? (hand edited to save space).

assertions in "..." to ask the intended satisfiability question to the SMT solver. The `push` and `pop` make these instrumentation assertions only visible in the scope of this analysis, declarations and assertions made prior to analysis code are visible in all analyses.

A typical use of `different`? is to check whether conditions differ for $0.5 < pSet$ and $0.6 < pSet$, i.e. whether this increase in threshold value from 0.5 to 0.6 matters – suggesting that these are different trust levels.

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, to encode requires 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 synthesis 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 initial users, though, when they try to declare these as Z3 types explicitly.

Witness generation. For each declared analysis, Z3 will try to decide it when running PEALT. If the Z3 ouput 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 synthesis, 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 11.

For symbolic synthesis, the model lists truth values for *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_i` in Figure 8. We mean to investigate how we can shorten such evidence in future work.

Execution constraints. Both synthesis methods need to constrain their input. For explicit

synthesis of policies within analyzed conditions, we need that no $*$ policy has scores outside $[0, 1]$ and that no policy has negative or non-constant scores. For explicit synthesis, we only have to ensure that *min* and *max* policies have constant scores, and we mean to lift the latter restriction in future work.

6 Validation

We here report experimental results for both synthesis methods on random and non-random analyses, and discuss other tool validation activities we conducted.

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. Explicit synthesis generates 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. Symbolic synthesis could generate 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 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 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.

On these randomly generated analyses, we conducted three experiments that share an execution and termination logic: an 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 synthesis 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. Figure 12 illustrates this. We use a function `calledBy` that models method call graphs with at most one incoming edge (the `forall` axiom) and declare 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) as shown in the figure.

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

```
(declare-fun calledBy (MethodName) Bool)
(assert (forall ((n MethodName) (m MethodName))
                (or (= m n) (implies (calledBy m) (not (calledBy n))))))
(assert (= q0 (calledBy n0)))
(assert (= q4 (< x0 (* x1 0.99679510))))
```

Figure 12: Some of the domain specific declarations and constraints generated by the call $randPeal\,3, 2, 1, 3, 2, 7, 0.5, 0.1$ and used in experiment Exp 1 (DS)

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 13. 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.

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 synthesis methods within two minutes. For $*$, explicit synthesis seems to be one order of magnitude better than symbolic synthesis, although the Z3 execution in the latter case appears to be faster. For $+$, on the other hand, symbolic synthesis 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 synthesis seem about the same as without domain-specific constraints. But $+$ now only can handle less than two-thousand rules for symbolic synthesis.

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 synthesis 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, we see that both methods can handle between two to three thousand policies with that many rules for *max* and *min*. For $*$ and $+$, explicit synthesis spends its bulk time in code generation whereas symbolic synthesis spends its bulk time in Z3 execution. For $*$, explicit synthesis is still about an order of magnitude better. For $+$, symbolic synthesis seems better than the explicit one but not significantly so.

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 the code generation for explicit synthesis. 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 our analyses, although this is easy to achieve by adding complex constraints as domain specifics. We validated both synthesis methods by running them side by side on randomly generated anal-

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

Figure 13: Experimental results: columns show used synthesis ("ex"plicit or "sy"mbolic) and choice of 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.

yses 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.

7 Related work

The language in Figure 1 differs from the one in [3]: it supports $*$ policies, negative and non-constant scores for symbolic synthesis, and the capture of logical dependencies of predicates q_i within PEALT. The symbolic synthesis for PEALT uses the same enumeration process for $+$ and $*$ on *minimal* (and not maximal as in [3]) index sets. PEALT implements most of the analyses suggested in [3], more complex ones are subject of 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 [2] 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 [10], 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 [12], security levels [9], and trust-management languages [1]. But we are not aware of substantial tool support for analyzing the effect of such enrichments.

Shinren [5] 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.

8 Conclusions

We have written a tool in which one can study different mechanisms of aggregating numerical trust evidence. This is achieved by implementing the condition expressions of a policy composition language as verification conditions that can be discharged 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 that 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. Our current tool prototype is available for experimentation on a machine with a dual-core CPU and 8G of RAM. The URL is `http://delight.doc.ic.ac.uk:55555/`. It is expected to be available until the first half of December 2013. Please contact Jim Kuo for support issues. We plan to make a first public alpha release of PEALT in early 2014; it presently supports verification of policy refinement, vacuity checking, sensitivity analysis of thresholds in conditions, and non-constant scores (for symbolic synthesis) to express metrics.

References

[1] 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)

[2] 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)

[3] 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)

[4] De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM 54(9), 69–77 (Sep 2011)

[5] Dong, C., Dulay, N.: Shinren: Non-monotonic trust management for distributed systems. In: IFIPTM. pp. 125–140 (2010)

[6] 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)

[7] Mayer, R., Davis, J., Schoorman, F.D.: An integrative model of organizational trust. Academy of Management Review 20(3), 709–734 (1995)

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

[9] 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`

[10] 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)

[11] 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)

[12] Schwoon, S., Jha, S., Reps, T.W., Stubblebine, S.G.: On generalized authorization problems. In: CSFW. pp. 202–218 (2003)