Evidence for Computed PDP Decision

michael.huth@xain.io

February 2019

1 Introduction

This note contains som thoughts on how we might compute evidence that supports why the PDP made a decision on a particular request under a policy. This evidence is thought to be used by programmers, users who issue requests or other human agents. Therefore, it is important that we can produce evidence that is interpretable by humans.

We will not explore here UI/UX aspects of such evidence but aim to produce evidence as a list or tree of facts. We will also assume that a sole policy is being evaluated. This should give us a good grounding for how such evidence may be computed for vertical and horizontal composition in the policy store.

The motivation for this work is a blog entitled "XACML is dead" which mentions the inability to provide such evidence as one of the main reasons for the decline or failure to rise of XACML:

`https://go.forrester.com/blogs/13-05-07-xacml_is_dead/`

2 Representation of Circuits

For sake of exposition, we assume that the Boolean circuits are generated by a grammar of form

$$C ::= aC \;\mid\; \neg C \;\mid\; C \;\&\&\; C \;\mid\; C \;||\; C \tag{1}$$

where aC ranges over atomic conditions given through relations or equations of expressions that contain or refer attributes.

3 Types of Evidence Sought

Recall that a request is processed by the PDP against two circuits, GoC() and DoC(). The decision is `conflict` if both evaluate to `tt`; it is `undef` if both evaluate to `ff`. Otherwise, the decision is `grant` if GoC() evaluates to `tt` and DoC() evaluates to `ff`, and it is `deny` if conversely DoC() evaluates to `tt` and GoC() evaluates to `ff`.

From this we can see that we may compute evidence for a decision as a simple combination of evidence computed for a circuit evaluating to a certain truth value.

It seems that this composition is needed for the decisions `conflict` and `undef`:

- For decision `conflict`, we need to compute evidence E_g for GoC() being `tt`, and evidence E_d for DoC() being `tt`. Then the ordered pair (E_g, E_d) serves as evidence for the decision to be `conflict`.
- Similarly, for decision `undef` we need to compute evidence E'_g for GoC() being `ff`, and evidence E'_d for DoC() being `ff`. Then the ordered pair (E'_g, E'_d) serves as evidence for the decision to be `undef`.

We may of course proceed similarly for the evidence of decisions `grant` and `deny`. But a user may not be interested in the evidence that prevented a conflict in these cases. For example, the evidence E_g for GoC() being `tt` may suffice as evidence for the decision to be `grant`. Similarly, E_d as evidence for the truth of DoC() may suffice as evidence for decision `deny`.

4 Evidence for a Circuit to have a Truth Value

The idea is as follows. We are given an assignment ρ that allows us to evaluate a circuit to a truth value and where ρ and the circuit may take on either of the values `tt`, `ff` or $\perp$.

The evidence E is first computed as a subtree of that circuit. Intuitively, subtrees that are discovered to not contribute to the computed truth value for the decision are being removed and the resulting tree serves as evidence E.

Of course, one may get similar effects if the circuit is encoded as an assertion in an SMT solver, and where all defined truth values of α (where $\rho(x)$ does not equal $\perp$) are added as additional assertions, e.g. "(assert = x true)". Then the computation of a model (satisfiability witness) for these assertions may serve as such evidence.

But we here mean to propose a simple static computation that renders such information, initially in tree form. A subtree could then be transformed into its list of atomic conditions and their truth values if desired.

The computation we specify is presented as a top down recursion. A bottom up approach may be better if we want to use metrics of subtrees to decide which subtree to remove as evidence. For example, if a && node in the circuit has truth value `ff`, then both child subtrees have that value as well, but we only need evidence from one of these subtrees. Different metrics are possible here, depending on the particular need; e.g. the height of these subtrees, the number of attributes referred to in their atomic conditions, combinations of such metrics, and so forth.

Below, we write $\rho(C) = v$ to denote that the (sub)circuit C evaluates to truth value v in $\{\texttt{tt}, \texttt{ff}, \perp\}$ under assignment ρ.

It is clear that we can compute evidence for all four decisions from evidence we can compute for facts of the form $\rho(C) = \texttt{tt}$ and $\rho(C) = \texttt{ff}$.

4.1 Evidence for a Circuit Being True/False

We do a case analysis over the type of node of the root of C and the type of truth value. Let us begin with $\rho(C) = \texttt{tt}$:

- T1 Let C be an atomic condition aC. Then we make no changes to that circuit.
- T2 Let C be a circuit of form $\neg C'$. Then we have that $\rho(C') = \texttt{ff}$. We therefore retain the root Negation node and compute recursively now evidence for $\rho(C') = \texttt{ff}$, which will transform C' into a subtree of C' within circuit C.
- T3 Let C be a circuit of form $C_1 \,\&\&\, C_2$. Then we know that $\rho(C_i) = \texttt{tt}$ for all $i = 1, 2$. We then recursively compute evidence for these C_i being true. This yields two subtrees that then render a subtree of C in context.
- T4 Let C be a circuit of form $C_1 \,||\, C_2$. Then we have that there is some i with $\rho(C_i) = \texttt{tt}$, let us say $i = 1$ for sake of illustration. Then we remove the entire subtree C_2 from C and recursively compute the evidence for $\rho(C_1) = \texttt{tt}$.[1]

Note that the case T2 above computes evidence for a circuit to be true as a function of evidence of a subtree to be false. Let us now consider that case, when a circuit is false:

- F1 Let C be an atomic condition aC. Like in T1, we then make no changes to that circuit.
- F2 Let C be a circuit of form $\neg C'$. Then we have that $\rho(C') = \texttt{tt}$. We therefore retain the root Negation node and compute recursively now evidence for $\rho(C') = \texttt{tt}$, which will transform C' into a subtree of C' within circuit C.
- F3 Let C be a circuit of form $C_1 \,\&\&\, C_2$. Then we have that there is some i with $\rho(C_i) = \texttt{ff}$, let us say $i = 1$ for sake of illustration. Then we remove the entire subtree C_2 from C and recursively compute the evidence for $\rho(C_1) = \texttt{ff}$.[2]
- F4 Let C be a circuit of form $C_1 \,||\, C_2$. Then we know that $\rho(C_i) = \texttt{ff}$ for all $i = 1, 2$. We then recursively compute evidence for these C_i being false. This yields two subtrees that then render a subtree of C in context.

Note that the above assumes that the circuit does not satisfy $\rho(C) = \bot$. This is consistent with our approach: the PDP only computes a decision if both circuits evaluate to a value other than ρ even though atomic conditions may

[1] This is a case in which it may be beneficial to have metrics for subtrees, to decide which subtree to remove in case that both are true.

[2] This is a case in which it may be beneficial to have metrics for subtrees, to decide which subtree to remove in case that both are false.

take on this value – as detailed in the FROST Yellow Paper. In particular, a top down computation of evidence never needs to consider subtrees C' with $\rho(C') = \bot$, as they will be removed if encountered as described in the cases T1-4 and F1-4 above.

In a bottom up approach, we may have to propagate $\bot$ values upwards whilst eliminating subtrees that lead to such an upwards propagation.

Also note that a pair of such evidence (E_g, E_d), e.g., would then be a pair of such subtrees. Moreover, these subtrees would be annotated with useful information, e.g. truth values of atomic conditions. And this information may be enriched with values of relevant attributes, that gave rise to assignment ρ.

5 Evidence for Policies Written in FROST

Above, we computed evidence on the compiled circuits, not on the FROST policies that gave rise to them. Computing evidence for FROST policies is certainly possible but may not be too easy to understand – given the nested nature of case policies.

But if we think of each case statement as the invocation of a policy operator such as "join", then a policy in FROST is also a tree-like object that may be processed in a manner not too dissimilar as the one that computed evidence for circuits.