Policy Verification Paper

May 2019

1 Introduction

Access-control policies are security-critical code. If such code contains inadvertent or malicious errors, then its execution may lead to security breaches with potentially grave consequences.

Errors in such policies may also have implications on the safety or other aspects of the system that such policies control. Although this is a valid concern, we will focus on security considerations of policies and policy writing. We do this since it gives this chapter a proper scope but also since in many systems vital aspects such as safety have additional safeguards that our access-control system could not corrupt – try as it might. For example, if a vehicle were to move at a speed of 40 miles per hour, then a policy that grants access to open the trunk might be granted in FROST. However, the Policy Enforcement Point (PEP) would only be able to pass on this decision to the internal vehicle system that then has its own way of dealing with such requests. In this case, it would refuse to enact the opening of the trunk due to its model of functional safety. FROST therefore is another abstraction layer that can be placed on top of existing system.

Our focus on policy *verification* pertains to both language levels of FROST: the actual policy language as well as the compiled Boolean circuits. The need for verification at both levels should be clear enough. We want to verify policies before they get compiled in that we want assurance that the declared policies do not contain any unintended behavior. And we mean to formally verify the Boolean circuits as these are the structures that actually get executed. For example, it is important to be able to verify that the two circuits that are the claimed representations of a policy do capture that policy's behavior fully and faithfully.

The policy verification we will develop in this chapter is a *parametric* framework: we offer a generic analysis tool, based on Satisfiability Modulo Theories [?] and individual analyses combine logical expressions that are determine by policy arguments and the intended question of the analysis. This combined logical formula is then the input to an SMT solver and the response is then interpretable as an answer to the intended question, e.g. whether a modified policy is not more permissive than an original one. It turns out that the Boolean circuits we developed in Chapter **??** for policy implementation and execution are also key building blocks for constructing the formulas that are subject to SMT analysis.

2 Policy Verification

It is thus helpful to think of these Boolean circuits as predicates when using them for policy verification. This is also reflected in implementations. Boolean circuits on embedded devices will have a JSON or similar data format whereas for verification they will be captured as assertions in the input language of the chosen SMT solver.

With these predicates at hand, it is now easy to express important policy analyses as satisfiability problems over the logic and theories that interpret atomic conditions, and the attributes that they contain. For example, a policy *pol* from the FROST language is gap free iff the predicate $\neg\mathsf{GoC}(pol)\ \&\&\ \neg\mathsf{DoC}(pol)$ is unsatisfiable. Similarly, a policy *pol* from the FROST language is conflict free iff $\mathsf{GoC}(pol)\ \&\&\ \mathsf{DoC}(pol)$ is unsatisfiable. In particular, if a policy *pol* is gap free and conflict free, then we may use $\mathsf{GoC}(pol)$ as an intermediate representation of this policy. This is a Boolean "circuit" that evaluates to true if the policy grants an access request, and to false if the policy denies an access request.

Since these policies make use of typed attributes, the satisfiability of conditions used in verification depends on the combination of theories for these typed expressions. For example, an attribute *agent.reputation* may have an axiom

$$\forall\, agents\colon 0 \leqslant agent.reputation \leqslant 1 \tag{1}$$

saying that the reputation of any agent is a real number between 0 and 1. A theory will have a number of such axioms, and we typically deal with a combination of theories, e.g. we may want to combine the axiom in (1) with the usual theory of real-valued arithmetic, and with theories that capture domain-specific knowledge.

Definition 1. *Throughout this Yellow Paper, we write $\mathcal{T}$ to denote a finite set of axioms that capture a combination of theories of interest.*

We then say that a formula ϕ, for example one generated by syntactic category cond, is satisfiable modulo $\mathcal{T}$, *iff the formula*

$$\phi \wedge \bigwedge_{\psi\in\mathcal{T}} \psi$$

is satisfiable. We write $\mathsf{SAT}_{\mathcal{T}}(\phi)$ to denote this test for satisfiability.

In other words, a formula ϕ is satisfiable modulo $\mathcal{T}$ if we can make ϕ true in a setting that also makes all formulas in $\mathcal{T}$ true. For example, $0 < agent.reputation < 1$ is satisfiable whereas $(0 < agent.reputation < 1)\ \&\&\ (agent.reputation \leqslant (agent.reputation)^2)$ is unsatisfiable modulo any theory $\mathcal{T}$ that contains the axiom in (1).

For example, when we state in the next example below that $\neg\mathsf{GoC}(pol)$ && $\neg\mathsf{DoC}(pol)$ is unsatisfiable the formal meaning of this is that $\mathsf{SAT}_{\mathcal{T}}(\neg\mathsf{GoC}(pol)$ && $\neg\mathsf{DoC}(pol))$ does not hold. Subsequently, we will often use this informal wording with its intended formal meaning.

This technology, which performs policy analysis by reducing it to satisfiability modulo theories, can also be used to verify the integrity of a representation of a policy *pol* from our FROST language.

Example 1 (Verification)**.** *Suppose, e.g., that φ is a Boolean Circuit that is claimed to faithfully represent pol in that truth of the circuit means* `grant` *and falsity means* `deny`*. Anyone can then verify this claim by*

1. *generating the conditions $\mathsf{GoC}(pol)$ and $\mathsf{DoC}(pol)$ as specified in Figure* **??***,*
2. *using a formal verification tool to show that*
   (a) *$\neg\mathsf{GoC}(pol)$ && $\neg\mathsf{DoC}(pol)$ is unsatisfiable, and so pol is indeed gap free,*
   (b) *$\mathsf{GoC}(pol)$ && $\mathsf{DoC}(pol)$ is unsatisfiable, and so pol is indeed conflict-free,*
3. *verifying that $\mathsf{GoC}(pol)$ and φ are logically equivalent.*

If any of the verification tasks in items 2 or 3 fail, this means that the integrity of φ has been disproved; otherwise, we have proof that φ indeed faithfully represents the policy pol.

A similar set of tasks can show that φ_{GoC} faithfully represents $\mathsf{GoC}(pol)$ and φ_{DoC} faithfully represents $\mathsf{DoC}(pol)$ for a policy *pol* that may neither be free of gaps nor free of conflicts. This would then verify the integrity of these two Boolean Circuits, which essentially capture the join normal form of that policy *pol*.

Example 2 (Change Management)**.** *We may also verify other aspects of policy administration. In change management, e.g. we may need assurance that an updated policy pol′ is not more permissive than some currently used policy pol. For example, a proof that*

$$\mathsf{T}(pol' \texttt{ eval grant}) \;\&\&\; (\mathsf{T}(pol \texttt{ eval undef}) \;||\; \mathsf{T}(pol \texttt{ eval deny}))$$

is unsatisfiable would show that policy pol′ can never grant whenever policy pol either denies or has a gap.

The verification tools one would use for these and other validation tasks may vary. Rewrite rules may capture equational theories of operators for transforming policies into equivalent ones and tools could perform such rewrite logic. This can also be helpful for simplifying policies in a manner that is meaningful to users.

Deeper semantic analyses may be obtained by the use of solvers for SMT ("Satisfiability Modulo Theories" [3]) – which can be used to compute answers to $\mathsf{SAT}_{\mathcal{T}}(\phi)$ – such as the OpenSMT solver [2], or through the use of theorem provers such as Isabelle [1, 7]. And these powerful and mature tools from formal methods may also be used to compress Boolean conditions into provably equivalent ones. Such compression of policies will be particularly beneficial if policies need to be stored and executed on resource-constrained devices.

A PDP may need to process a policy that contains gaps or conflicts. For example, a policy may be submitted off-line through a cybersecurity protocol and so the PDP cannot make any assumptions about the semantic behavior of that policy. A PDP may therefore need to *wrap* this policy, at the top-level, into an idiom that makes the composed policy enforceable for a PEP.

Example 3 (Deny-by-default `case`-policy)**.** *A policy wrapper that forces all conflicts and gaps of a policy pol to be denials would then be*

```
case {
   [pol eval undef: deny]
   [pol eval conflict: deny]
   [true: pol]
}
```

This deny-by-default *composition pattern can not only be expressed in our policy language but may also be hard-coded in a PDP. For example, let a PDP execute two Boolean circuits* φ_{GoC} *and* φ_{DoC}*, where these circuits capture the meaning of* GoC(*pol*) *and* DoC(*pol*)*, respectively. The PDP may then combine the outputs of these circuits as seen in the truth table in Figure 1.*

GoC(*pol*)	DoC(*pol*)	output of PDP
true	true	`deny`
true	false	`grant`
false	true	`deny`
false	false	`deny`

Figure 1: Combining outputs of circuits GoC(*pol*) and DoC(*pol*) so that the combination honors all grants and denials of *pol* but overrides all gaps and conflicts of *pol* into denial.

3 Dead-Code Analysis

We now describe a source-to-source transformation of FROST policies to that *all* intended execution paths of the transformed policy are realizable by some input. These transformations can be seen both as code optimization but also as a means of flagging up potential oversights or misunderstandings of programmers. An execution path that is not realizable is caused by some control structure not allowing for the execution of all its branches. In the FROST language, we have two such control structures: rules and `case`-policy. We refer to branches that can never be executed as *dead code*, the familiar term for this phenomenon in program analysis (see e.g. [4]). Dead code is hardly ever intended to be written by any programmer, therefore there is great benefit in detecting and removing it before a FROST policy is compiled into Boolean circuits for execution.

Execution of FROST Policies Cannot Get Stuck Let us first note that a policy written in FROST, and where the atomic predicates are well typed, can never get "stuck" in its execution upon a complete input – a vector of values for the attributes occurring in that policy. This is so since constant policies immediately return a decision, and rules can evaluate their condition (due to well typedness of the condition) and then immediately return `undef` or the decision value of the rule (`grant` or `deny`). For `case`-policies, execution cannot stop either: either one of the first $n-1$ guards g_i evaluates to `true` or the default case applies. The evaluation of guards g_i cannot get stuck, by an argument that uses structural induction, since its policies do not get stuck (again, by structural induction) and the equality test of policy decisions and evaluation of logical conjunctions cannot get stuck either due to well typedness. Therefore, a continuation policy p_i of that `case`-policy, with $1 \leqslant i \leqslant n$, gets executed. By structural induction, that policy p_i also does not get stuck.

****!! it would be good to formalize this claim, to state the operational semantics of the FROST policy language and to then show this claim by structural induction; in this case, it may suffice to do induction over the parse tree of the expression, where we also have to cover the condition evaluation that this does not get stuck either !!****

Detecting and Eliminating Dead Code The code transformations that we propose on FROST policies are source to source: a FROST policy p is transformed to a semantically equivalent FROST policy p' such that each of the execution paths of p' does actually occur for some suitable input to policy p'. This is true whenever the satisfiability checker we employ for expressions of syntactic clause *cond* are complete, meaning that all calls to it with conditions *cond* return either "satisfiable" or "unsatisfiable". Should some calls to this SAT checker return "don't know" instead (the underlying theory may be undecidable in general or the input may be computationally too demanding), we will interpret this answer soundly so that we only ever remove dead code; this is then an *under-approximation* as it may leave some dead code within a policy. However, assuming that no SAT call returns "don't know", then we are guaranteed that all dead code will have been removed by our algorithm below from the policy p.

The algorithm performs a recursive descent over the abstract syntax tree of policy p, processing `case`-policy higher up in that tree first, before processing `case`-policies further below in that tree. The reason for that is that this processing may remove cases and therefore some continuation policies p_i, which may themselves be `case`-policies and so subject to this analysis in principal. But the algorithm thus has no need of processing such p_i as they would be removed in any event in the `case`-policy in which its case gets removed. Note that the same policy p_i, as syntactic object, may occur elsewhere within p where it may not be removed, but then this is a different subtree of the abstract syntax tree of p.

Before we present that algorithm, let us understand the computational issues at hand for each of the syntactic categories of FROST policies.

- **Constant policies:** there are no execution paths here, a constant policy denotes the end of an execution path and so no dead code can originate from constant policies. Therefore, the algorithm will return them unchanged.
- **Rules:** a rule has two branches, depending on the truth value of its condition. In programming terms, a rule is akin to an if-statement. We want to ensure that both branches of a rule can be executed by some inputs. The algorithm therefore needs to be able to decide whether that is possible. We will see below that any failure to reach a branch will make the algorithm transform the rule into a semantically equivalent constant policy.
- **`case`-policies:** a general `case`-policy has $n > 1$ branches, the first $n-1$ guarded by guards g_i with continuation policies p_i, and the n-th one with the default guard true and continuation policy p_n. We want to make sure that each of these n cases can indeed be reached by some input to the policy. Otherwise, cases can be removed and may need to be flagged up to the programmer as a potential coding error.

We won't discuss how and when to flag up such issues to programmers here, as this is an orthogonal usability issue of the policy analysis and its implementation. We may address the above semantic considerations with predicates, the first one we already defined above and recall here for sake of convenience:

$\mathsf{SAT}_{\mathcal{T}}\,(\mathit{cond})$ = formula *cond* is satisfiable

$\mathsf{NonTr}\,(\mathit{cond})$ = formula *cond* is neither logically valid nor unsatisfiable

We note that $\mathsf{NonTr}\,(\mathit{cond})$ – standing for "Non-Trivial" – can be derived from $\mathsf{SAT}_{\mathcal{T}}\,(\cdot)$ as

$$\mathsf{NonTr}\,(\mathit{cond}) \quad \text{iff} \quad \mathsf{SAT}_{\mathcal{T}}\,(\neg\mathit{cond})\ \&\&\ \mathsf{SAT}_{\mathcal{T}}\,(\mathit{cond}) \tag{2}$$

It is easy to see that we can detect any dead code in rules with calls to the predicate $\mathsf{NonTr}\,(\cdot)$. For example, consider a rule `grant if` *cond*. This rule does not have dead code if both of its branches can be executed, in other words, if both `grant` and `undef` can be returned by this rule for appropriate inputs. We now show that the latter is equivalent to $\mathsf{NonTr}\,(\mathit{cond})$ being true.

First, let $\mathsf{NonTr}\,(\mathit{cond})$ hold, then:

- a witness for the satisfiability of *cond* exists by (2) as *cond* is satisfiable, and this witness represents an input on which this rule returns `grant`,
- a witness to the satisfiability of $\neg\mathit{cond}$ exists by (2) since *cond* is not valid, and this witness represents an input on which this rule returns `undef`.

Second and conversely, two different inputs to that rule that make the rule return `grant` and `undef`, respectively, function as witnesses to the satisfiability of *cond* and $\neg$*cond*, respectively; and this implies that $\mathsf{NonTr}\,(\mathit{cond})$ holds by (2).

From the above, we see that if $\mathsf{NonTr}\,(\mathit{cond})$ holds for a rule with condition *cond*, the dead code transformation leaves that rule unchanged. What if $\mathsf{NonTr}\,(\mathit{cond})$ does not hold? We then need to make a distinction. By definition of $\mathsf{NonTr}\,(\mathit{cond})$ and given that it is false in this instance, we have two cases by appeal to (2):

- *cond* is unsatisfiable; then the rule always returns `undef` and so the dead-code transformation turns this rule into the constant policy `undef`.
- $\neg$*cond* is unsatisfiable; then the rule always returns `grant` and so the dead-code transformation turns this rule into the constant policy `grant` – if the rule is of form `deny if` *cond*, the transformation returns the constant policy `deny` in this case for a similar reason.

The remaining clause for policies is the `case`-policy with $n > 1$ cases, $n - 1$ guards g_i with continuation policies p_i, and a default n-th case with continuation policy p_n. The predicate $\mathsf{T}(g)$ is defined as in the Figure **??**, it spells out the logical condition for guard g to be true, and $\mathsf{T}(g)$ is an expression of syntactic clause *cond*. As defined in Figure **??**, the predicate $\mathsf{R}(g_i)$ spells out the exact conditions on the input needed to execute continuation policy p_i – for each $1 \leqslant i < n$. Similarly, $\mathsf{R}(\mathsf{true})$ spells out the exact conditions for reaching the execution of the default continuation policy p_n.

We should note that these formulas $\mathsf{R}(g_i)$ and $\mathsf{R}(\mathsf{true})$ are dependent on the concrete `case`-policy. For example, for $n = 8$ the removal of the 4-th case as dead code would require us to recompute $\mathsf{R}(g_i)$ for $5 \leqslant i \leqslant 8$ – by removing the conjunct $\neg\mathsf{T}(g_4)$ from these formulas.

It turns out that the treatment of `case`-policies is relatively simple in that we can process its guards in order as specified in the algorithm given in Figure 2. We point out that the first and last if-statement of that algorithm capture the aforementioned under-approximation: whenever a SAT call returns “don't know”, that if-statement won't execute its sole branch and so the considered case will not be removed from the `case`-policy. The explanation of the algorithm is given in the caption of Figure 2.

Let us consider some examples of dead-code analysis. The first one illustrates how the instantiation of a composition pattern gives rise to code optimization opportunities that our dead code analysis can pick up.

Example 4. *Re-consider the* `case`*-policy in Figure* **??** *which encodes the meaning of the* `join` *composition operator. Let us assume that we now instantiate that* `case`*-policy with a policy P that can only evaluate to* `grant` *or* `undef`*, and a policy Q that cannot evaluate to* `undef` *but to all other three values.*

Then our algorithm will remove the second, third, and fifth case due to resulting unsatisfiabilities from the possible outputs of P and Q. It will therefore

```
$REM = \emptyset$;
for ($i = 1$ to $n - 1$) {
  if ($\mathsf{SAT}_{\mathcal{T}}\,(\mathsf{R}(g_i))$ returns “unsatisfiable”) {
    remove case $g_i\colon p_i$ from case-policy;
    recompute $\mathsf{R}(g_j)$ for all $i < j \leqslant n$;
    $REM = REM \cup \{i\}$;
  }
  if ($REM = \{1, \ldots, n-1\}$) {
    return $p_n$;
  }
  // $\mathsf{R}(g_i)$ is “satisfiable” for all $i \in \{1, \ldots, n-1\} \setminus REM$
  if ($\mathsf{SAT}_{\mathcal{T}}\,(\mathsf{R}(\mathsf{true}))$ returns “unsatisfiable”) {
    remove case true: $p_n$;
    $m = \max(\{1, \ldots, n-1\} \setminus REM)$;
    if ($|\{1, \ldots, n-1\} \setminus REM| == 1$) {
      return $p_m$;
    } else {
      $g_m = \mathsf{true}$;
      return computed case-policy;
    }
  }
}
```

Figure 2: Pseudo-code for detecting and removing dead code in a case-policy. For case-policies, dead code consists of entire cases $g_i\colon p_i$ or true: p_n. The algorithm iterates through all non-default cases in their declared order. If $\mathsf{R}(g_i)$ is unsatisfiable, the case $g_i\colon p_i$ gets removed from the case-policy and all predicates $\mathsf{R}(g_j)$ with $i < j \leqslant n$ get recomputed to reflect that removal. Set REM records the indices of those case that get removed.
If this iterative process happens to remove all non-default cases, then the remaining case-policy really consists only of the default case true: p_n and so the entire case-policy is being replaced with policy p_n. Otherwise, a non-empty list of non-default cases $g_i\colon p_i$ with $i \in \{1, \ldots, n-1\} \setminus REM$ from the input case-policy remains.
For the reachability predicate $\mathsf{R}(\mathsf{true})$ of this remaining case-policy, we check whether that formula is unsatisfiable. If so, the previous case can function as default case and we remove default case true: p_n. Then we need to make a case distinction: if there is only one case remaining, which will be $g_m\colon p_m$, then policy p_m is returned and so the case-policy is replaced with p_m by our dead-code analysis; otherwise more than one case remains and we set g_m to be true making true: p_m the new default case and return that case-policy.
This achieves that, whenever a case-policy is being returned, all remaining $\mathsf{R}(g_i)$ are satisfiable, including the one for the default case. Since these predicates capture reachability of these clauses, this demonstrates that the returned case-policy is free of dead cases. Of course, the same dead code analysis needs to be performed next on each remaining continuation policy p_i, including in the case in which only p_n or p_m is returned by the above algorithm.

compute the optimized `case`*-policy*

```
case {
   [(P eval undef): Q]
   [(Q eval conflict): conflict]
   [((P eval grant) && (Q eval deny)): conflict]
   [true: P]
}
```

Then we have that

$$\mathsf{R}(\mathsf{true}) = \neg(P \texttt{ eval undef}) \,\&\&\, \neg(Q \texttt{ eval conflict}) \,\&\&\, \neg((P \texttt{ eval grant}) \,\&\&\, (Q \texttt{ eval deny}))$$

This formula is satisfiable when P and Q evaluate to `grant`*. Therefore, the default case remains as is (the last if-statement in Figure 2 is not executed), and the above* `case`*-policy is the output of our algorithm.*

Let us know illustrate how our dead-code analysis can also flag up potential programmer issues or errors that developers should be alerted to:

Example 5. *Consider the policy*

$$P = \texttt{grant if } ((\mathit{user.reputation} > 1.5) \,\&\&\, \mathit{user.insured})$$

and some other policy Q within a `case`*-policy*

```
case {
   [P eval undef: deny]
   [P eval grant: grant]
   [true: Q]
}
```

Let the theory $\mathcal{T}$ contain the axiom in (1)*. Then*

$$\mathsf{SAT}_{\mathcal{T}}\,(\neg((\mathit{user.reputation} > 1.5) \,\&\&\, \mathit{user.insured}))$$

is satisfiable, e.g. by making user.insured false and setting user.reputation to 0.2*. Therefore, P* `eval undef` *is also satisfiable. However,* $\mathsf{SAT}_{\mathcal{T}}\,((\mathit{user.reputation} > 1.5) \,\&\&\, \mathit{user.insured})$ *is unsatisfiable, since no reputation score can be above* 1.0*. The* 1.5 *in policy P is presumably a typo made by the programmer. Therefore, the dead-code analysis will remove only the second case from this* `case`*-policy and turn this into*

```
case {
   [P eval undef: deny]
   [true: Q]
}
```

The programmer might then reflect on this removal and its subsequent analysis would help with identifying the programming error.

!! I think we also need a theorem here saying that the algorithm in Figure 2 is correct; and also that the transformations for rules and constant policies are correct; ideally, we should also prove that the nesting of case-policies will be handled correctly, which may require writing up pseudo-code that explains how the algorithm in Figure 2 gets called repeatedly in such nestings !!

4 Policy Verifier Prototype

**!! description of a prototype implementation for policy analysis using an SMT solver such as Z3; this should not work on a nice UI ir UX, but focus on functionality and perhaps exclude the dead-code analysis; initial ideas:

- perhaps one can adapt the Haskell compiler to generate assertions in Z3 that capture the two Circuits DoC and GoC of a policy
- Then one can use variable names in Z3 to bind such circuits to names, which will also be handy for analyses that involve more than one policy as arguments
- the analysis itself is then simple a push-pop frame that may add some additional assertions (specific to the analysis), and then aims to generate a model
- push-pop frames allow us to run more than one analysis in sequence on the same global definitions or pertinent DoC and GoC assertions
- it suffices to report the models that Z3 outputs, then one can discuss what these variable values mean in terms of the intended policy analysis question

!!**

**!! A key question is whether you should bother with writing any front end for this or not. A front end would allow one to declare policies, conditions, and analyses and a compiler would then generate the resulting SMT code for this. Jim did this for the PEALT language, see e.g. https://link.springer.com/chapter/10.1007%2F978-3-642-54862-8_8 . But this is quite a software engineering effort and more "routine", so for the paper it may be better to just describe how such a front end might look like, how a compiler might work, and to just implement something that generates SMT assertions that bind instances of DOC, GOC, T(.), and Eval(.,.) to SMT variables.

The SMT code that is executed could then be hand-edited by adding some boilerplate SMT code around these definition. You could then instantiate some of the analysis examples from the Yellowpaper with concrete policies and/or replicate some of its examples to run them with your generated SMT assertions, notably Examples 4 and 5 in the FROST Yellowpaper.

Let us consider Example 5 of the Yellow paper. You would have two policies, pol and pol' for which you would have to generate three conditions:

- a Boolean variable c1 for which you would add `(assert (= c1 <compiledExpression>))` where `<compiledExpression` is the logical condition $T(pol'\ eval\ grant)$ as compiled in the Yellowpaper, but now represented in SMT syntax
- similarly, you would use Boolean variables c2 and c3 for the other two $T(.)$ expressions of that example
- then you would add a push/pop block in which you would add `(push) (assert (and c1 (or c2 c3)))` followed by `(check-sat) (get-model) (pop)`

Note that there is no need for pushs and pops if you only ever run one analysis over the same set of policies or conditions. But it seems prudent to have that.

Also note that for this example the output would be either `unknown`, `unsat` or `sat`. In the latter case, it would also output values for declared variables (the directive `get-model` sees to that), notably truth values for c1-c3. Of course, it would be of interest to also have truth values for atomic conditions in pol and pol'. Also, these atomic conditions would be specified with the use of terms (e.g. arithmetic expressions). So the model would also return values for those non-Boolean variables. But this presupposes that the compilation of the declarations c1-c3 includes declarations of non-Boolean variables and assertions that bind such variables to the meanings of terms occurring in pol or pol'.

In the paper one could then consider a few of these analysis examples, whose code is generated by a mix of compilation and hand-editing (or completely automatic if a front-end is worked on as well). For each analysis, you could then inspect such a model (if one is found) and map back its values onto the terms, policies, and conditions considered for this FROST example to see why it does witness such a violation of change management. If UNSAT is being returned, you could then state what that means for the analysis in question.

And one could then vary that example by using a different pol' so that the same analysis would then return UNSAT (meaning that the policy management is then giving us the intended effect).

Further note that there may be domain-specific knowledge that has to be added as additional assertions that would be in the scope

of all analyses (and so would be asserted prior to push/pop blocks). For example, that the reputation of all agents is in between 0 and 1. Note that some domain-specific knowledge, e.g. that of linearly ordered real-valued arithmetic, is already implicitly represented in SMT (the "MT" part of that).
!!**

The above ideas have been implemented into a working prototype, interfacing with the SMT solver Z3. We give some of the highlights of the prototype implementation, together with some examples of policy analysis.

4.1 A FROST DSL

First let us describe FROST by way of a domain-specific language (DSL) embedded in Haskell. As with any endeavour into language design and implementation, one faces various design choices each with its own benefits and trade-offs. But since the focus here is predominantly on the *policy analysis* part rather than the language per se, we keep the presentation of FROST itself relatively simple. In particular, although we implement a deep embedding to allow for the flexibility of staged compilation, we mention only a useful subset of the full FROST language. We use Haskell as the host language for its strong type system, syntax overloading capabilities and mature library support for several SMT solvers including Z3.

The abstract syntax of FROST policies is given by the following algebraic data type (ADT) *Pol*

```
data Pol = Konst Dec
         | Case [(Guard, Pol)] Pol
         | Filter Rule
```

Policies comes in one of three forms. The simplest of these is *Konst* comprising a single access control *decision*, a value of type *Dec*

```
data Dec = Grant | Deny | Gap | Conflict
```

The values *Grant* and *Deny* have the expected meaning with respect to access control, and we call these *definite* decisions. On the other hand, the decision *Gap* represents a lack of information to conclude either of the two, while *Conflict* represents an excess of such information.

The four values of *Dec* correspond to those of the Belnap bilattice: *Grant* and *Deny* are the respective top and bottom elements of the lattice in the *truth* ordering, while *Conflict* and *Gap* are the respective top and bottom elements in the *knowledge* ordering.

A *Case* combines one or more policies into a composite. Intuitively, it is a list of sub-policies each tagged with a *guard*, appended with a final (unguarded) "default" policy. Guards themselves are expressed as follows

```
data Guard = Eval Pol Dec
           | TrueG | AndG Guard Guard | NotG Guard
```

A singleton guard *Eval p d* can be read as a relation that the policy *p* evaluates to the decision *d*. Guards can also be empty (*TrueG*), conjunctions (*AndG*) and negations (*NotG*).

The third form of policy is a *Filter*, comprising one of two forms of *Rule*

```
data Rule = GrantIf Cond | DenyIf Cond
```

Rules specify the condition under which an access request is granted (*GrantIf*) or denied (*DenyIf*). Such conditions are built from boolean-like operators of the type *Cond* which we express in GADT (*generalised* algebraic data type) syntax

```
data Cond where
  TrueC :: Cond
  NotC  :: Cond → Cond
  AndC  :: Cond → Cond → Cond
  OrC   :: Cond → Cond → Cond
  (:≡:) :: Term a       → Term a       → Cond
  (:<:) :: Term Integer → Term Integer → Cond
  (:⩽:) :: Term Integer → Term Integer → Cond
```

Conditions are constructed from usual logical operators (*TrueC*, *NotC*, *AndC* and *OrC*) as well as relational operators for equality (:≡:) and inequality (:<:, :⩽:) over *terms* – values of a polymorphic type *Term*

```
data Term :: ∗ → ∗ where
  LitI :: Integer → Term Integer
  LitS :: String  → Term String
  VarI :: String  → Term Integer
  VarS :: String  → Term String
  Add  :: Term Integer → Term Integer → Term Integer
  Mul  :: Term Integer → Term Integer → Term Integer
```

The *Term* type constructor makes proper use of GADT generality. A value of type *Term a* represents a FROST term of type *a*. For example, *LitI* 23 represents the integer literal 23, a term of type *Term Integer*. Integer variables are also integer terms e.g. *VarI* "age". The same applies for string literals (*LitS*) and variables (*VarS*). We also allow sum (*Add*) and product (*Mul*) of integer terms.

At this stage we have the main ingredients for a small DSL of FROST policies. Let us now add to this just a little syntactic sugar. First we use a type class *Boolean* that generalises the standard *Bool* type, unifying the treatment of all Boolean-like values

```
class Boolean b where
  true, false :: b            -- logical true, false
  bnot        :: b → b        -- complement
  (∧), (∨)    :: b → b → b    -- and, or
```

```
(⟹), (⟺) :: b → b → b   -- implies, equivalence
...                        -- nand, xor, etc.
```

Instances of *Boolean* include not only *Bool* but also *symbolic* Boolean values (see later Section 4.3)[1]. Now since conditions also have a similar structure, it is convenient to overload these methods further

```
instance Boolean Cond where { true = TrueC; bnot = NotC
                            ; (∧)  = AndC ; (∨)  = OrC }
```

And similarly for guards, taking *true* = *TrueG*, (∧) = *AndG*, *bnot* = *NotG*. From these minimal instance definitions, all other *Boolean* methods are derived.

In addition, we add an instance of the class *IsString* to conveniently recognise string literals as terms and similarly *Num* for integer literals, as well as the standard numeric operators for addition and multiplication

```
instance IsString (Term String)  where fromString = LitS
instance Num      (Term Integer) where fromInteger = LitI
                                       (+) = Add
                                       (*) = Mul
                                       ...
```

Furthermore, assume we have a helper function *grantIf* = *Filter* ∘ *GrantIf*. Then a simple example of a policy in our DSL is as follows

```
drivingTest :: Pol
drivingTest = grantIf $
  VarS "subject" :≡: "Learner" ∧
  bnot ((VarI "theory" + VarI "practical")
        :⩽: 70)
```

drivingTest expresses a grant decision when the following sub-conditions hold; the string variable `subject` has the value `"Learner"`, and the sum of the integer variables `theory` and `practical` is not $\leqslant 70$.

4.2 Compiling to Circuits

Conditions are not just relevant for *Filter* policies – in fact, they form a lower-level language that *any* policy can be translated into. Policies in this more verbose form lend themselves to analysis and verification. In the following, we give some details of the translation process.

Any FROST policy p can be represented equivalently in a *join normal form* consisting of a pair of conditions, which in this context we refer to as Boolean circuits. One of these circuits *goc* p expresses the condition that holds exactly when p grants or conflicts. The other circuit *doc* p expresses the condition that holds exactly when p denies or conflicts.

[1]See the *Boolean* type class defined in the SBV package for more details.

```
goc, doc :: Pol → Cond   -- given by 2 columns below
goc (Konst Grant)        = true      doc (Konst Deny)         = true
goc (Konst Conflict)     = true      doc (Konst Conflict)     = true
goc (Konst _)            = false     doc (Konst _)            = false
goc (Filter (GrantIf c)) = c         doc (Filter (GrantIf _)) = false
goc (Filter (DenyIf _))  = false     doc (Filter (DenyIf c))  = c
goc (Case [] p)          = goc p     doc (Case [] p)          = doc p
```

While the definitions of *goc* and *doc* are straightforward for *Konst* and *Filter* policies, nonempty *Case* policies are a little more involved. We describe this translation for *goc* only (the *doc*-translation is similar).

Let *gps* be a nonempty list of guarded policies of length $n-1$. The circuit

```
goc (Case gps p) = ...
```

is a composite disjunction of the form

$$\left(\bigvee_{i=1}^{n-1} d_i\right) \vee d_n$$

where each disjunct d_i operates on an *initial segment* of *gps*. As such, it is helpful to use a function *inits* from *Data.List* where *inits xs* returns all initial segments of a list *xs* in ascending order of length [2]. This motivates the definition as a fold

```
goc (Case gps p) = foldr ((∨) ∘ d_i) (d_n gs p) gpsInits
  where gpsInits = tail (inits gps)
        gs       = map fst gps
```

We have no use for the first segment (the empty list $[\,]$), which is discarded with *tail*. Thus each disjunct d_i will operate on the guarded policies in the initial segment of length i. The last disjunct d_n however is a special case requiring just the guards *gs* of the maximal segment (i.e. *gps* itself) and the default policy p.

Let us look at the definitions of the disjuncts closely. In particular, d_i takes the guarded policies $(g_1, p_1), \ldots, (g_i, p_i)$ to construct a conjunction of the form

$$d_i := \left(\bigwedge_{j=1}^{i-1} \neg g_j\right) \wedge g_i \wedge goc\ p_i$$

consisting of negated conjuncts g_j and a positive g_i. But notice that if we interpret these naively as guards, the formula does not quite type-check since *goc* p_i is a *condition.* To make this more precise, we require a function specifying the condition for when a guard holds

```
guard2Cond :: Guard → Cond
guard2Cond TrueG          = true
```

[2]e.g. *inits* $[1,2,3]$ evaluates to $[[\,],[1],[1,2],[1,2,3]]$.

$$\begin{aligned}
&\mathit{guard2Cond}\ (\mathit{Eval}\ p\ \mathit{Conflict}) &&= \mathit{goc}\ p \wedge \mathit{doc}\ p\\
&\mathit{guard2Cond}\ (\mathit{Eval}\ p\ \mathit{Gap}) &&= \mathit{bnot}\ (\mathit{goc}\ p) \wedge \mathit{bnot}\ (\mathit{doc}\ p)\\
&\mathit{guard2Cond}\ (\mathit{Eval}\ p\ \mathit{Grant}) &&= \mathit{goc}\ p \wedge \mathit{bnot}\ (\mathit{doc}\ p)\\
&\mathit{guard2Cond}\ (\mathit{Eval}\ p\ \mathit{Deny}) &&= \mathit{bnot}\ (\mathit{goc}\ p) \wedge \mathit{doc}\ p\\
&\mathit{guard2Cond}\ (\mathit{AndG}\ g_1\ g_2) &&= \mathit{guard2Cond}\ g_1 \wedge \mathit{guard2Cond}\ g_2\\
&\mathit{guard2Cond}\ (\mathit{NotG}\ g) &&= \mathit{bnot}\ (\mathit{guard2Cond}\ g)
\end{aligned}$$

This leads to the definition of d_i where each guard is converted to a condition via *guard2Cond*

```
$d_i\ \mathit{gpsInit} = \mathit{foldr}\ ((\wedge) \circ \mathit{bnot} \circ \mathit{guard2Cond} \circ \mathit{fst})$
                    $(\mathit{guard2Cond}\ g_i)$
                    $\mathit{gpsInit}'$
                $\wedge\ \mathit{goc}\ p_i$
  where
    $(\mathit{gpsInit}', [(g_i, p_i)]) = \mathit{splitAt}\ (\mathit{length}\ \mathit{gpsInit} - 1)\ \mathit{gpsInit}$
```

A key observation is the isolation of the last guarded policy (g_i, p_i) of the segment *gpsInit*. By splitting *gpsInit* in this way at $i-1$, we obtain simultaneously the sole positive guard g_i, its corresponding policy p_i and all negatively-guarded policies $\mathit{gpsInit}'$ before it. Notice that such a splitting must exist since *gps* (and hence *gpsInit*) is nonempty.

The last disjunct d_n follows a similar structure. As above, we give a formula for intuition followed by a precise definition capturing the intuition.

$$d_n := \left(\bigwedge_{k=1}^{n-1} \neg g_k\right) \wedge \mathit{goc}\ p$$

$$d_n\ \mathit{gs}\ p = \mathit{foldr}\ ((\wedge) \circ \mathit{bnot} \circ \mathit{guard2Cond})\ (\mathit{goc}\ p)\ \mathit{gs}$$

This completes the definition of *goc*.

4.3 Generating SMT Assertions

A FROST policy in Boolean circuit form (as explained in Section 4.2) can in turn be interpreted as assertions that a SMT solver can process. This essentially boils down to an interpretation of FROST conditions as symbolic predicates. Since conditions are in turn built from FROST terms, let us begin by describing this interpreter for terms while introducing helpful types and functions of *Data.SBV* from the SBV (SMT-based verification) library[3].

****!! NOTE kept this todo here just for reference, and will be removed later for it has been addressed in the content that immediately follows it.**

[3]http://leventerkok.github.io/sbv/

```
term2Sym :: Term a → Symbolic (SBV a)
term2Sym (LitI n)   = pure (literal n)
term2Sym (LitS s)   = pure (literal s)
term2Sym (VarI x)   = sInteger x
term2Sym (VarS x)   = sString x
term2Sym (Add n m) = (+) ≺$≻ term2Sym n ⊛ term2Sym m
term2Sym (Mul n m) = (∗)  ≺$≻ term2Sym n ⊛ term2Sym m
```

```
cond2Pred :: Cond → Predicate
cond2Pred TrueC          = pure true
cond2Pred (NotC c)       = bnot ≺$≻ cond2Pred c
cond2Pred (AndC c₁ c₂) = (∧)  ≺$≻ cond2Pred c₁ ⊛ cond2Pred c₂
cond2Pred (OrC c₁ c₂)  = (∨)  ≺$≻ cond2Pred c₁ ⊛ cond2Pred c₂
cond2Pred (t₁ :≡: t₂)    = (.≡) ≺$≻ term2Sym t₁ ⊛ term2Sym t₂
cond2Pred (t₁ :<: t₂)    = (.<) ≺$≻ term2Sym t₁ ⊛ term2Sym t₂
cond2Pred (t₁ :⩽: t₂)    = (.⩽) ≺$≻ term2Sym t₁ ⊛ term2Sym t₂
```

While writing up this section, I realised there's a slight problem with the predicate generator above – specifically with the interpretation of FROST variables i.e. *VarI* **and** *VarS*. **It turns out that the SBV functions** *sInteger* **and** *sString* **create fresh variables which is not what I want. While this is not necessarily a problem in a lot of cases, it is when the condition in question contains repeated occurrences of the same variable. For example, the condition**

```
bnot (VarI "x" :<: 0) ∧ VarI "x" :<: 10
```

when converted to a predicate (via *cond2Pred***) will not work as expected when you do a** *sat* **check on it, since the occurrences of** x **will be treated as completely separate symbolic values i.e.**

$$\neg(x_1 < 0) \wedge x_2 < 10$$

This should be fixable however, roughly as follows.

- **Do a first pass of the condition, collecting all the free variables.**
- **From this, create the named symbolic variables but record these along with the names to create an "environment" that you'll pass around in the interpreter.**
- **Interpret variables by looking up this environment for the corresponding symbolic value.**

!!**

While we will see that most FROST terms can be interpreted reasonably straightforwardly, variables require some care. Recall that variables in our DSL appear freely in conditions and in particular, multiple occurrences of variables e.g. *VarI* "x" in

```
bnot (VarI "x" :<: 0) ∧ VarI "x" :<: 10
```

is understood to refer to the same x as one would expect. Thus both occurrences of x should duly be interpreted as the same *symbolic value* to the solver. In SBV, symbolic values are of type *SBV a*. For example, symbolic integers are of type *SBV Integer* and symbolic strings of type *SBV String*. We will use their respective type synonyms *SInteger* and *SString* in the following data type of environments

```
data Env = Env { envI :: Map String SInteger
               , envS :: Map String SString }
```

An environment consists of two maps[4] associating FROST variable names with symbolic values, one for integers and the other for strings. For convenience let us use a type synonym *Vars* for pairs of (respectively, integer and string) variable names:

```
type Vars = ([String], [String])   -- integer var names, string var names
```

The idea is that an environment is allocated with fresh symbolic values tagged according to such names given. The task of declaring the values themselves in the solver can be provided by SBV functions

```
sIntegers :: [String] → Symbolic [SInteger]
sStrings  :: [String] → Symbolic [SString]
```

which are effectful computations in SBV's *Symbolic* monad. Then allocation of an environment is an application of these functions before packaging up into an *Env* as follows.

```
allocEnv :: Vars → Symbolic Env
allocEnv (vi, vs) = do
  vi' ← sIntegers vi
  vs' ← sStrings vs
  let ei = Map.fromList (zip vi vi')
  let es = Map.fromList (zip vs vs')
  return (Env ei es)
```

Having allocated an environment γ :: *Env*, we can thread this through an interpreter that translates FROST terms into symbolic values. Accordingly, we make use of an environment (or *reader*) monad[5] equipped with a "getter" operation *ask*:

[4]e.g. from *Data.Map* in the CONTAINERS package.

[5]See e.g. *Control.Monad.Reader* in MTL for a standard library implementation, and [5] for a reference.

```
newtype Reader e a =
  Reader {runReader :: e → a} deriving (Functor, Applicative, Monad)
ask :: Reader e e   -- retrieves the environment
ask = Reader id
```

where in particular we specialise the type parameter $e = Env$, to abstract away the repetitive detail of passing γ around in the following.

```
interpT :: Term a → Reader Env (SBV a)
interpT (LitI n)    = pure (literal n)
interpT (LitS s)    = pure (literal s)
interpT (VarI x)    = ask ≫= λγ → case Map.lookup x (envI γ) of
  Just i    → return i
  Nothing → error $ "integer variable not in environment: " ++ x
interpT (VarS x)    = ask ≫= λγ → case Map.lookup x (envS γ) of
  Just s    → return s
  Nothing → error $ "string variable not in environment: " ++ x
interpT (Add n m) = (+) ≺$≻ interpT n ⊛ interpT m
interpT (Mul n m) = (∗)  ≺$≻ interpT n ⊛ interpT m
```

To interpret *VarI x*, we retrieve γ with *ask* and lookup[6] the map *envI* γ for the symbolic integer named x. Similarly for *VarS*. The other cases use the standard *applicative functor* [6] methods *pure*, ⊛ and ≺$≻ to lift symbolic values into the *Reader Env* context. Literals are interpreted symbolically with the SBV function *literal* :: $a \rightarrow SBV\ a$. Sums and products are also interpreted as their symbolic counterparts $+$ and $*$[7].

Interpretation of conditions then follows a (pleasantly) predictable pattern:

```
interpC :: Cond → Reader Env SBool
interpC TrueC          = pure true
interpC (NotC c)       = bnot ≺$≻ interpC c
interpC (AndC c₁ c₂) = (∧)  ≺$≻ interpC c₁ ⊛ interpC c₂
interpC (OrC c₁ c₂)  = (∨)  ≺$≻ interpC c₁ ⊛ interpC c₂
interpC (t₁ :≡: t₂)    = (.≡) ≺$≻ interpT t₁ ⊛ interpT t₂
interpC (t₁ :<: t₂)    = (.<) ≺$≻ interpT t₁ ⊛ interpT t₂
interpC (t₁ :⩽: t₂)    = (.⩽) ≺$≻ interpT t₁ ⊛ interpT t₂
```

Note that *SBool* is simply a type synonym for *SBV Bool* which (like *Cond* and *Guard*) is an instance of the type class *Boolean*. Hence the methods *true*, *bnot*, $\wedge$, $\vee$ here all refer to *symbolic* logical connectives. Equality and inequality of terms are also interpreted as their SBV counterparts $.\equiv$, $.<$ and $.\leqslant$.

Finally, we have a helper function *fvs* to determine the free variables in a given condition:

[6]*Map.lookup* :: $k \rightarrow Map\ k\ a \rightarrow Maybe\ a$ where *Map.lookup x m* returns the value in m corresponding to the key x (as a *Just* value) or *Nothing* if x is not a key in m.

[7]Note that *SInteger* is an instance of *Num*.

```
fvs :: Cond → Vars
```

The definition of *fvs* is an entirely routine recursion over the structure of conditions, and is omitted here. With this last piece in place, everything is brought together in *cond2Pred* which translates a condition into a *Predicate* (which is just a type synonym for *Symbolic SBool*):

```
cond2Pred :: Cond → Predicate
cond2Pred c = runReader (interpC c) ≺$≻ allocEnv (fvs c)
```

cond2Pred uses *fvs* for a first-pass over the condition c to obtain its free variables. An environment of symbolic values is then allocated according to these variable names, and threaded through to initialise the *Reader* computation *interpC c*.

Let us look at an example. For policies p and p', the derived predicate *lessPermissive p p'* can be understood to mean that p is "less permissive" than p', in the particular sense that a model exists witnessing p' granting while p denies or has no opinion.

!! Supplement the following with small concrete example. *represents* can be improved to better reflect the essence of the procedure. !!

```
lessPermissive :: Pol → Pol → Predicate
lessPermissive p p′ = cond2Pred
                      $ guard2Cond
                      $ Eval p′ Grant ∧ (Eval p Gap ∨
                                         Eval p Deny)
```

The following example is more involved since it depends on the results of multiple calls to the solver. The idea is that given a condition c, we want to ascertain whether it is in fact a Boolean circuit that faithfully represents a policy p.

```
represents :: SatModel b ⇒ Cond → Pol → IO (Maybe b)
represents c p = do
  gap ← sat ((∧) ≺$≻ denyOrGap ⊛ grantOrGap)
  if modelExists gap
  then return (extractModel gap) else do
    conflict ← sat ((∧) ≺$≻ grantOrConf ⊛ denyOrConf)
    if modelExists conflict
    then return (extractModel conflict)
    else extractModel ≺$≻ prove ((⟺) ≺$≻ grantOrConf ⊛ cond2Pred c)
  where
    grantOrConf = cond2Pred (goc p)
    denyOrConf  = cond2Pred (doc p)
    denyOrGap   = bnot ≺$≻ grantOrConf
    grantOrGap  = bnot ≺$≻ denyOrConf
```

The procedure begins by obtaining the circuit-pair predicates *gocp*, *docp* for p to determine whether it is gap-free. If so, then a similar check occurs for determining whether p is also conflict-free. If both of these hold, it remains to check that the *goc*-circuit is logically equivalent to c. If in any of these stages we have a negative result, a counter-example is returned.

5 Conclusion

References

[1] Ballarin, C., Homann, K., and Calmet, J. Theorems and Algorithms: An Interface between Isabelle and Maple. In *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation, ISSAC '95, Montreal, Canada, July 10-12, 1995* (1995), pp. 150–157.

[2] Bruttomesso, R., Pek, E., Sharygina, N., and Tsitovich, A. The OpenSMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings* (2010), pp. 150–153.

[3] de Moura, L. M., and Bjørner, N. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (2008), pp. 337–340.

[4] Debray, S. K., Evans, W. S., Muth, R., and Sutter, B. D. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst. 22*, 2 (2000), 378–415.

[5] Jones, M. P. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text* (Berlin, Heidelberg, 1995), Springer-Verlag, pp. 97–136.

[6] Mcbride, C., and Paterson, R. Applicative programming with effects. *J. Funct. Program. 18*, 1 (Jan. 2008), 1–13.

[7] Paulson, L. C. Isabelle: The Next Seven Hundred Theorem Provers. In *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings* (1988), pp. 772–773.