H. Roßnagel, C.H. Schunck, S. Mödersheim, D. Hühnlein (Hrsg.): Open Identity Summit 2020,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2020

Accountable Trust Decisions: A Semantic Approach

Anders Schlichtkrull[1] Sebastian Mödersheim[2]

Abstract: This paper is concerned with the question of how to obtain the highest possible assurance on trust policy decisions: when accepting an electronic transaction of substantial value or significant implications, we want to be sure that this did not happen because of a bug in a policy checker. Potential bugs include bugs in parsing documents, in signature checking, in checking trust lists, and in the logical evaluation of the policy. This paper focuses on the latter kind of problems and our idea is to validate the logical steps of the trust decision by another, complementary method. We have implemented this for the Trust Policy Language of the LIGHTest project and we use the completely independently developed FOL theorem prover RP_X as a complementary method.

Keywords: Trust policies, Accountability, Security, Logic, Theorem Prover, Isabelle, eIDAS

1 Introduction

When an organization engages in an electronic transaction of substantial value or with significant implications for the organization, it should have *policies* in place to protect themselves or mitigate risks. For instance, before starting a costly production the business wants to be sure that the apparent customer that ordered the production is really the entity who initiated the order and that the electronic signature on the order is indeed legally binding (so it is reasonable to assume that it can be enforced by the legal system in the applicable jurisdiction). This may also include limits on the total value of the order so that even if a legal dispute fails or takes time, the company can stay operational.

The project LIGHTest [BL16] offers an infrastructure for formulating trust policies for this kind of purpose, e. g., a company may define the following policy: we accept every order up to a specified amount, if it was signed with an eIDAS qualified signature. They may additionally allow for trust schemes outside the European Union, but rely on trust translation recommendations: suppose the European Commission defines a translation from a foreign scheme to eIDAS, say level 3 in the foreign scheme is regarded as equivalent to level advanced in eIDAS, then the company may accept those signatures as well, but may choose to set a lower cap on the value of accepted orders. The reason for such a lower cap is that a legal dispute outside the European Union may be much more difficult for this company. On the other hand, they may have other business policies, e. g. if the customer

[1] Technical university of Denmark, DTU Compute, Richard Petersens Plads, Bygning 324, 2800 Kongens Lyngby, Denmark andschl@dtu.dk
[2] Technical university of Denmark, DTU Compute, Richard Petersens Plads, Bygning 324, 2800 Kongens Lyngby, Denmark samo@dtu.dk

is well-known or has a good reputation from other partners. Finally, We may also have delegation, i. e., if the customer is itself a company, it may be an employee signing on behalf of the company. For all these purposes, LIGHTest offers a Trust Policy Language (TPL) and has an automated trust verifier (ATV) to evaluate a given transaction against a given policy, possibly looking up trust list entries as needed [Mö19].

This paper is concerned with the question: how can we trust the trust decision made by the ATV, or more generally, how can we be sure about the result of an automated policy evaluation? The problem would be if we accidentally accept a transaction that actually does not meet the requirements of the policy – due to a bug in policy evaluation tool. We see at least the following aspects relevant to this question:

Cryptography Are for instance the electronic signature algorithms sufficiently secure (until the time we rely on them) and implemented correctly?

Parsing When extracting information from a document, is this parsing done correctly? Potential problems include ambiguous document formats, parts erroneously not included in the signature, vulnerabilities to injection/overflow attacks.

History Can we later prove to a third party what was the state of a trust list at the time of the policy decision? There can be modifications of the trust list by the hosting organization, as well as the problem of revocation.

Semantics Assume the previous points are all correct: does the transaction then indeed logically satisfy the policy? Potential problems include that the ATV has some logical bugs such as instantiating variables inconsistently.

Real world Does the policy actually make sense to the business such as limiting damages and providing sufficient legal assurances?

The main contribution of this paper is to propose a solution for the point "Semantics" by an "independent set of eyes". The idea comes from the area of proof assistants like Isabelle/HOL [NPW02] which provide a way to formalize mathematical claims and proofs for them in a language similar to programming languages, and that the proof assistant can check. This gives an overwhelming assurance that proofs are indeed correct, because it rules out the problem of holes, false conclusions, or imprecisions in proofs that often occur in standard proofs that are written in a mixture of mathematical terms and natural language. While Isabelle/HOL offers some automation to find proofs, the human prover still has to provide at least the main idea of a proof. The prover RP_X [SBT19], in contrast, is an automatic prover for First-Order Logic (FOL) that was proven correct in Isabelle. It is based on a handbook chapter by Bachmair and Ganzinger [BG01] and thus proves that their approach is correct. The formalization, however, revealed several non-trivial mistakes in the chapter, all of which were then rectified. With RP_X we thus have a theorem prover where we have the same overwhelming assurance as in Isabelle when RP_X accepts a FOL-statement as logically valid. Since LIGHTest's TPL is inspired by Prolog, we can cast policy decisions as FOL theorem proving problems and thus use RP_X for double checking them.

One may wonder why not to use RP_X as the policy decision tool in the first place. The reason is that the ATV of LIGHTest evaluates policies in a different way than a theorem prover: it processes a transaction (parsing, signature checking, comparing fields) and interacts with different servers maintaining trust lists; also it has to process policy rules and their elements in a given order. In contrast, a theorem prover deals only with logical formulae and needs the freedom to "process" them in an arbitrary order in order to be most effective. We thus propose the combination of ATV and RP_X – or more generally: the combination of a policy decision tool with a verifier – as they benefit from complementary strengths.

The main contribution of this paper is the integration of the ATV and RP_X. This includes defining a good interface for the first three aspects **Cryptography**, **Parsing** and **History** that are beyond what RP_X can check. The implementation of our approach is part of the LIGHTest distribution and first experiments have indeed revealed a few mistakes in a preliminary implementation of the ATV that are now all corrected; thus our verification has, if anything, already practically contributed to improving the ATV.

2 Preliminaries

Our work is based on the LIGHTest Trust Policy Language TPL [Mö19] which we first briefly introduce by way of an example TPL policy (adapted from [Mö19]). This example is that an auction house receives bids for auction lots over the Internet in a custom format (defined by the auction house themselves) and it accept all bids up to 1500 Euro as long as the bidding form is signed by an eIDAS qualified signature. In TPL this looks as follows:

```
accept(Transaction) :-
  extract(Transaction, format, theAuctionHouse2020format),
  extract(Transaction, bid, Bid),
  Bid <= 1500,
  extract(Transaction, certificate, Certificate),
  extract(Certificate, format, x509),
  extract(Certificate, pubKey, PK),
  verify_signature(Transaction, PK),
  check_eIDAS_qualified(Certificate).

check_eIDAS_qualified(Certificate) :-
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, issuer, IssuerCertificate),
  extract(IssuerCertificate, trustScheme, TrustSchemeClaim),
  trustscheme(TrustSchemeClaim, eIDAS_qualified),
16.   trustlist(TrustSchemeClaim, IssuerCertificate, TrustListEntry),
17.   extract(TrustListEntry, format, trustlist_entry),
18.   extract(TrustListEntry, pubKey, PkIss),
19.   verify_signature(Certificate, PkIss).
```

Lines 1 to 9 define a *predicate* `accept`. Such a definition is called a *clause*. The predicate `accept` specifies when a transaction is accepted. The transaction is here represented as a parameter variable `Transaction` to the predicate (on line 1). As a convention, variables always start with a capital letter, while identifiers that start with a lower-case letter are constants or predicate symbols. The following lines give constraints on the transaction that need to be all true in order to satisfy `accept(Transaction)`.

TPL supports the use of arbitrary data formats as long as a parser for that format exists. Consider the constraint on line 2 that uses the `extract` predicate. The `extract` predicate connects TPL with the parsers of the various data formats where the first two parameters are the input and the third parameter is the output. In the concrete example we ask what the `format` of `Transaction` is and we expect a particular result defined by the constant `theAuctionHouse2020format`. In the example, this is a custom format defined by the auction house themselves, containing a number of fields that they require to be filled in in order to make a bid at their current auction; assume these fields include at least the fields `bid` and `certificate` that we formulate constraints on below. The policy decision would be negative if at this point the parser for this format does not successfully parse the given transaction.

In line 3 the `extract` predicate is given as parameters `Transaction`, `bid` and `Bid`. As said before, the `theAuctionHouse2020format` contains a field called `bid`, and the constraint here is to simply bind the concrete value in the transaction to the variable `Bid` (in fact this constraint cannot fail). Line 4 specifies the constraint that whatever is now the value of `Bid` should be at most 1500. Again, if the concrete bid in the transaction is above 1500, then at this point the policy decision stops with a negative result.

In lines 5-7 we first extract a certificate from the transaction, now represented as variable `Certificate`, then we put the constraint that it should be of the x509 format, and lastly we extract a public key from it, binding it to a new variable `PK`. Here we assume a similar interface to the x509 format as for the auction house format. In line 8 we use the `verify_signature` predicate to require that the signature on the transaction `Transaction` can be verified with the `PK` public key. In fact, this requires that `theAuctionHouse2020format` is a format with a notion of a signature.

Each constraint of the clause so far (lines 2-8) uses *built-in* predicates of TPL (`extract`, <=, and `verify_signature`). The predicate in line 9, however, is not built-in: it is defined by the clause in lines 11-19. Lines 11-14 constrain what format the certificate must have and also extracting from it an issuer certificate and from that a trust scheme claim. A trust scheme claim is a URL to a trust list that the issuer certificate claims to be represented on. In line 15 we use the built-in `trustscheme` predicate with second parameter `eIDAS_qualified` which checks if the URL points to the eIDAS trust list. In line 16 we use the built-in predicate `trustlist` actually perform the trustlist lookup, to check if it indeed contains the required `IssuerCertificate`. As a result of a successful lookup, we obtain a `TrustListEntry` that we check in lines 17 to 19: we check the entry format, extract the public key stored in the entry, now `PkIss`, and verify `Certificate`'s signature with respect to `PkIss`.

In this example we have defined both the predicates `accept` and `check_eIDAS_qualified` by one clause each. In general, we can define any number of clauses, and they represent alternative ways to satisfy a predicate. For instance, in the example a bid above 1500 Euro would not fulfill the above clause, but if another `accept` clause is specified (e. g. on known customers) then the ATV tries that next.

3 Transcripts

The goal of this paper is to verify the logical aspect of the trust decisions of the ATV – and we leave out the aspects that are "outside" the logical realm, namely the verification of signatures, parsing, and server lookups. We thus need an appropriate interface between the logical and the extra-logical side. To this end we introduce the notion of a *transcript* as a triple (P, Q, E) where P is the TPL policy, Q is a *query* and E is an *event log*. The query will be simply of the form `accept(transaction)` where `transaction` is a constant representing the transaction document in question. In fact, for all elements that we talk about in the transcript, we use such symbolic constants. The event log contains a recording of all built-in predicates that were successfully evaluated by the ATV during the policy decision.

Since we need to work later with symbolic constants, but the ATV works on quite different data-structures, we need to take an intermediate step to arrive at a transcript. As a policy is being checked, the ATV will build what is basically a tree representation of the various objects that it stores. Running the above policy on with a concrete transaction could result in the following tree representation:

```
(root)
+--transaction
|  +-- format = "the_auction_house_2020"
|  +-- bid = "600"
|  +-- certificate
|      +-- pubKey
|      +-- issuer
|          +-- trustScheme = "trust.eidas.eu"
+--trustlistentry1
   +-- pubKey
```

This tree represents a state of the ATV where it contains a transaction and a trust list entry which each are represented as subtrees, namely the subtrees rooted in "(root).transaction" and "(root).trustlistentry1" respectively. We shall from now on ignore the root node in paths and thus the mentioned paths of the example start with "transaction" or "trustlistentry1". The tree has internal nodes like "transaction.certificate". The tree also has leaves such as "transaction.bid". This leaf contains the value "600". Additionally, the ATV keeps track of the concrete data that some parts of the tree represent, for instance "transaction.certificate" and "trustlistentry1.pubKey" represent a certificate and a public key, respectively, and the ATV needs to keep track of the certificates signature.

We have thus augmented the ATV so that it can generate the event log as a side effect during its normal work. We use again the example policy from section 2 and as a transaction the concrete objects shown in the above tree (fulfilling the policy):

```
extract(transaction, format, theAuctionHouse2020format).
extract(transaction, bid, 600).
600 <= 1500.
extract(transaction, certificate, transaction_certificate).
extract(transaction_certificate, format, x509).
extract(transaction_certificate, pubKey, transaction_certificate_pubKey).
verify_signature(transaction, transaction_certificate_pubKey).
extract(transaction_certificate, format, eIDAS_qualified_certificate).
extract(transaction_certificate, issuer, transaction_certificate_issuer).
extract(transaction_certificate_issuer, trustScheme,
            transaction_certificate_issuer_trustScheme).
trustscheme(transaction_certificate_issuer_trustScheme, eIDAS_qualified).
trustlist(transaction_certificate_issuer_trustScheme,
            transaction_certificate_issuer, trustlistentry1).
extract(trustlistentry1, format, trustlist_entry).
extract(trustlistentry1, pubKey, trustlistentry1_pubKey).
verify_signature(transaction_certificate, trustlistentry1_pubKey).
```

This log contains an encoding of the concrete instance of all built-in predicates that occurred during evaluation. The representation includes constants representing the paths into the tree that were used during the execution; for example, `transaction_certificate` represents the path "transaction.certificate". We take care that when such constants are introduced they do not clash with the constants already present in the policy.

4 Translating Transcripts to Logical Formulae

We now translate a transcript to logical formulae. Our running example policy will be translated to the following logical formula:

$(\forall \mathit{Transaction}, \mathit{Bid}, \mathit{Certificate}, \mathit{PK}.$
$\quad \mathit{accept}(\mathit{Transaction}) \leftarrow ($

$$
\begin{array}{l}
\quad\quad \mathit{extract}(\mathit{Transaction}, \mathit{format}, \mathit{theAuctionHouse2020format}) \wedge \\
\quad\quad \mathit{extract}(\mathit{Transaction}, \mathit{bid}, \mathit{Bid}) \wedge \\
\quad\quad \mathit{less\_or\_eq}(\mathit{Bid}, \mathit{i1500}) \wedge \\
\quad\quad \mathit{extract}(\mathit{Transaction}, \mathit{certificate}, \mathit{Certificate}) \wedge \\
\quad\quad \mathit{extract}(\mathit{Certificate}, \mathit{format}, \mathit{x509}) \wedge \\
\quad\quad \mathit{extract}(\mathit{Certificate}, \mathit{pubKey}, \mathit{PK}) \wedge \\
\quad\quad \mathit{verify\_signature}(\mathit{Transaction}, \mathit{PK}) \wedge \\
\quad\quad \mathit{check\_eIDAS\_qualified}(\mathit{Certificate}) \\
\quad) \\
) \\
\wedge \\
(\forall \mathit{Certificate}, \mathit{IssuerCertificate}, \mathit{PkIss}, \mathit{TrustListEntry}, \mathit{TrustSchemeClaim}. \\
\quad \mathit{check\_eIDAS\_qualified}(\mathit{Certificate}) \leftarrow (\\
\quad\quad \mathit{extract}(\mathit{Certificate}, \mathit{format}, \mathit{eIDAS\_qualified\_certificate}) \wedge \\
\quad\quad \mathit{extract}(\mathit{Certificate}, \mathit{issuer}, \mathit{IssuerCertificate}),\ \wedge \\
\quad\quad \mathit{extract}(\mathit{IssuerCertificate}, \mathit{trustScheme}, \mathit{TrustSchemeClaim}) \wedge \\
\quad\quad \mathit{trustscheme}(\mathit{TrustSchemeClaim}, \mathit{eIDAS\_qualified}) \wedge \\
\quad\quad \mathit{trustlist}(\mathit{TrustSchemeClaim}, \mathit{IssuerCertificate}, \mathit{TrustListEntry}) \wedge \\
\quad\quad \mathit{extract}(\mathit{TrustListEntry}, \mathit{format}, \mathit{trustlist\_entry}) \wedge \\
\quad\quad \mathit{extract}(\mathit{TrustListEntry}, \mathit{pubKey}, \mathit{PkIss}) \wedge \\
\quad\quad \mathit{verify\_signature}(\mathit{Certificate}, \mathit{PkIss}) \\
\quad) \\
)
\end{array}
$$

The translation for each clause replaces the commas (`,`) with conjunction symbols ($\wedge$), and the colon dash (`:-`) is replaced with an implication symbol ($\leftarrow$). Lastly all variables in the clause are universally quantified (using the $\forall$ symbol). The translation of the policy is then simply the conjunction (using $\wedge$) of the translated clauses. Note that numbers such as 1500 are translated to symbolic constants *i1500* and the <= operator becomes the predicate *less_or_eq*. Our implementation makes sure that these names do not clash with names of other logical constants or predicates as to avoid an ambiguity in their meaning. Let us call the above example formula P^{ex}.

The event log is translated as a conjunction as well, in our example E^{ex} is the formula:

$$
\begin{array}{l}
\mathit{extract}(\mathit{transaction}, \mathit{format}, \mathit{theAuctionHouse2020format}) \wedge \\
\mathit{extract}(\mathit{transaction}, \mathit{bid}, \mathit{i600}) \wedge \\
\mathit{less\_or\_eq}(\mathit{i600}, \mathit{i1500}) \wedge \\
\ldots \\
\mathit{verify\_signature}(\mathit{transaction\_certificate}, \mathit{trustlistentry1\_pubKey})
\end{array}
$$

With the query $Q^{ex} = \mathit{accept}(\mathit{transaction})$ we have the translated transcript (P^{ex}, Q^{ex}, E^{ex}).

5 Semantics: From Logic Programming to FOL

Check the Transcripts Essentially the idea is now that after a successful run of the ATV we have three formulae (P, Q, E): the policy P, the query Q and the event log E. We essentially want to double check with RP_X whether $P \wedge E$ logically implies Q. If so, then the positive decision of the ATV is *verified*. However, as always, the devil is in the details and we describe now how to make the connection to RP_X semantically precise.

RP_X [SBT19] is an automatic theorem prover for First-Order Logic (FOL). Given a FOL formula in TPTP format [Su17], RP_X attempts to prove that the negation of the formula is unsatisfiable. While RP_X can run into non-termination (since validity is undecidable for FOL), we have a strong guarantee when it does terminate. The reason is that the inference engine of RP_X has been proved to be *sound* and *complete* using the proof assistant Isabelle/HOL [SBT19]. *Soundness* gives us strong mathematical guarantees that if RP_X claims that the formula is unsatisfiable then indeed it is unsatisfiable. *Completeness* gives us strong mathematical guarantees that if the formula is unsatisfiable then RP_X will be able to prove that, when given enough time.

Since TPL is inspired by Prolog, unsurprisingly its semantics is based on logic programming as well, and in fact this work exploits how close TPL is to the semantics of FOL for which RP_X implements an automated theorem prover. There are some differences however, and we now highlight these details carefully and discuss how we can handle them.

Free Algebra Logic programming generally works on so-called *free models*, for an overview see for instance [EFT94, Chapter 11] and since there is sometimes confusion about the semantics, Hinrichs and Genesereth suggested the formal definition of *Herbrand Logic* [HG06] to contrast it more precisely with FOL. One of the key differences is that in Herbrand logic and logic programming in general, function symbols behave like in a free term algebra. For instance, if we have a binary function symbol f, then $f(t_1, t_2) = f(s_1, s_2)$ holds iff both $t_1 = s_1$ and $t_2 = s_2$. In other words: two terms are equal iff they are syntactically equal. This means in particular that for any two distinct constants c and d, it holds that $c \neq d$, because constants are just functions of arity 0. In contrast, standard FOL allows to model function symbols with algebraic properties such as commutativity.

Universes The model-theoretic definition of a logic is based on the concept of a *universe*, i. e. a non-empty set U of objects. For standard FOL, every function f of arity n is interpreted as a function from U^n to U, and every relation symbol of arity n is interpreted as a subset of U^n. For instance, the universe may be $U = \{0, 1\}$ and $+$ is interpreted as disjunction, and $\cdot$ is interpreted as conjunction, and the binary relation $<$ is interpreted to be true only for the pair $(0, 1)$. The difference for Herbrand logic is that the universe is determined by the set of function symbols we employ: we take as universe the set of terms that can be built from the terms. For instance if we have just two function symbols 0 of arity 0 and s of arity 1, then $U = \{0, s(0), s(s(0)), \ldots\}$ which can be regarded as the set of natural numbers. The "interpretation" of function symbols is then as expected.

Arithmetics In fact, this makes Herbrand logic even more expressive than standard FOL: we cannot formalize arithmetics in first-order logic because we lack the expressive power to formalize that the universe U is the natural numbers (we would have to formalize well-foundedness or the induction principle which are higher-order concepts), while Herbrand logic fixes the universe and we can thus get the natural numbers "for free". This even allows formalizing arithmetics (addition, multiplication, and comparison on natural numbers) as Hinrichs and Genesereth show [HG06]: even though we cannot for instance define addition as a binary function directly, we can use a ternary relation like $add(x, y, z)$ to represent that the addition of x and y gives z, and based on this axiomatize arithmetics completely. While this allows a semantically unambiguous integration of arithmetic into our policy language, what TPL (or logic programming approaches for that matter) can actually support is the direct evaluation of ground arithmetic statements like $5 + 3 < 100$, but they cannot solve equations. Therefore, in TPL we have to require that when the ATV reaches a condition like $X < 100$ that X is instantiated with a concrete integer through some other condition. In fact, note that while validity of formulae in FOL is semi-decidable, for Herbrand logic neither validity nor its complement is semi-decidable.

Least Models A last important difference to FOL is that most logic programming approaches do also fix the interpretation of the relationship symbols to be the *least* interpretation that satisfies all clauses. For instance, consider the policy that consists only of the single clause $p() : -q()$, and no more information is given. Then in normal FOL, there are three interpretations satisfying this clause: $q()$ can be false and $p()$ can be either true or false, or both $q()$ and $p()$ are true. The least interpretation, i. e. the interpretation chosen by Prolog-style semantics, is that both $p()$ and $q()$ are false. This is sometimes also called *negation by failure*: since we fail to prove $q()$, it counts as false and thus we also fail to prove $p()$ which is therefore also false.

While this behavior can cause confusion in logic programming (e. g. when using *not* and *cuts* as in Prolog), for policies it can make specifications actually quite intuitive: a policy is described always in a positive way, i. e. by sufficient conditions to satisfy the policy (or a particular concept of the policy), and everything else is outside the policy, i. e. a default-deny behavior which also means that forgetting to describe a case leads to erring on the safe side.

Mind the Gap Now that we have highlighted all the differences, let us consider how they need to be taken into account so that the verification we perform in the FOL-prover RP_X indeed agrees precisely with the semantics of TPL. Recall that we already have predicates like `extract` that are interfaces between the logical side and the real-world documents and servers. We handle them axiomatically in the logic, i. e. whatever checks and lookups the ATV does are recorded and supplied as facts, e. g. that $\mathit{extract}(\mathit{transaction}, \mathit{format}, \mathit{theAuctionHouse2020format})$ holds.

The first idea is now that, since we cannot formalize arithmetic in FOL, we handle all arithmetic checks axiomatically as well, e. g., if the ATV encountered the check $500 < 1000$, then this is also added as an axiom to the library. Indeed, it is easy to check such statements outside the FOL, so there was no sense in trying to integrate them in RP_X (which would only be possible in some approximation, anyway).

For the other issues – free algebra, universes and least models – we make use that all policy decisions have a particular form, namely evaluating a goal predicate with respect to a policy. More in detail, let H be a conjunction of the policy rules and all other information we have (statements about extractions, signature verifications, server lookups, and arithmetic checks), and let q be a query q (a ground predication). The *least model* of H is the least interpretation that satisfies all conditions we described above: the universe is the set of ground terms that can be built using the function symbols, every function symbol is freely interpreted in that universe, and all relationship symbols are interpreted as the least relation that satisfies all clauses in H. The policy decision, written $H \models_{TPL} q$, is now the question whether q holds in the least model of H.

The corresponding question that we ask RP_X is whether the formula $H \wedge \neg q$ is *satisfiable* in FOL. Satisfiable means that there is at least one satisfying FOL interpretation for this formula. If RP_X answers "no", then there is no way to interpret the universe and the function and relation symbols such that both H and $\neg q$ are satisfied – and this includes the least TPL model of H and thus $H \models_{TPL} q$. Thus if RP_X finds $H \wedge \neg q$ unsatisfiable, we know that q was correctly accepted as satisfied by the Automated Trust Verifier and we have successfully verified the trust decision.

In fact, the converse statement also holds: if RP_X answers "yes" to the question whether $H \wedge \neg q$ is satisfiable, then actually $H \not\models_{TPL} q$. In other words, a correct positive trust decision from the ATV is never refuted by RP_X, only when the ATV erroneously marks a query q as fulfilled, will RP_X complain. The proof of this is trickier though and we only give a brief sketch. Suppose we have some satisfying FOL interpretation $\mathcal{I}$ for $H \wedge \neg q$ and compare it to the least TPL model of H, then the TPL model will be "at least as fine": equality on terms is the finest possible relation in the least TPL model, and thus the least TPL model of the predicates contains at most as much as $\mathcal{I}$. Thus since q is not true in $\mathcal{I}$, it is also not true in the least TPL model of H, thus $H \not\models_{TPL} q$.

6 Experiments and Conclusion

The approach described in this paper is completely implemented and part of the LIGHTest distribution. In fact, we have already started testing it when the implementation of the ATV was still in a preliminary state. We found a couple of examples where the ATV and RP_X did not agree on policy decisions. For example a predicate like $p(X, X)$ is actually true for an arbitrary value as the first parameter – only the second parameter must be identical. This *unification* between parameters was not correctly implemented in the first version of the ATV. The error was of type *wrong reject*, i. e. the decision was negative when it should be positive, which is erring on the safe side, but undesirable nonetheless. All mistakes have been corrected and extensively tested using our RP_X connection.

To our knowledge, the only other work that double checks policy decision through a connection to an independent verifier is by Jim [Ji01]. There, the accountability argument hinges on the proof checker being a relatively simple program; in contrast, we use with RP_X a verifier that itself is verified in Isabelle. We believe that such works are in principle feasible and worthwhile for other policy languages where a formal semantics is defined that can be verified by means other the main policy decision tools. This exploits the fact that the design of such a semantics is often simpler than the procedure to obtain decisions that also integrates the extra-logical aspects like server lookups.

This work has focused exclusively on verifying the logical aspects of the decision. Let us at least briefly discuss the other aspects. For the parsing of documents, we have proposed a notion akin to the formats of TPL that in a heterogenous eco-system of formats prevent confusion about the meaning of messages [MK14]. For the cryptography and transmission channels there are first works on verifying implementations [Bh16]. This is crucial for server lookups, but not sufficient. The problem that trust lists may change over time implies the problem to later prove to a third party that the policy was satisfied at the time of checking. In fact, it is a reasonable requirement that trust lists maintain historical records, but especially in a volatile environment like servers for delegation that also try to protect the contents of the delegation lists against monitoring, this may be non-trivial.

This is actually related to the more "high-level" problem, namely whether a policy makes even sense for a business in the first place: that all conditions we check and all information we have gathered in a policy decision are sufficient to prove to a third party – e. g. in a legal dispute – what has happened. While many legal aspects are outside a technical view, we plan as future work to verify accountability properties [KTV10] of logging mechanisms and their relation to the policies we put in place.

Acknowledgments We thank Andreas Viktor Hess, Georg Wagner, Stefan More and Lukas Alber for helpful discussions and support in the adaption of the ATV. This work was supported by the Sapere-Aude project "Composec: Secure Composition of Distributed Systems", grant 4184-00334B of the Danish Council for Independent Research, by the EU H2020 project no. 700321 "LIGHTest: Lightweight Infrastructure for Global Heterogeneous Trust management in support of an open Ecosystem of Trust schemes" (lightest.eu) and by the "CyberSec4Europe" European Union's Horizon 2020 research and innovation programme under grant agreement No 830929.

References

[BG01] Bachmair, L.; Ganzinger, H.: Resolution Theorem Proving. In (Robinson, A.; Voronkov, A., eds.): Handbook of Automated Reasoning. Vol. I, Elsevier and MIT Press, pp. 19–99, 2001.

[Bh16] Bhargavan, K.; Delignat-Lavaud, A.; Fournet, C.; Kohlweiss, M.; Pan, J.; Protzenko, J.; Rastogi, A.; Swamy, N.; Béguelin, S. Z.; Zinzindohoue, J. K.: Implementing and Proving the TLS 1.3 Record Layer. IACR Cryptology ePrint Archive 2016/, p. 1178, 2016.

[BL16] Bruegger, B. P.; Lipp, P.: LIGHT$^{\mathrm{est}}$ - A Lightweight Infrastructure for Global Heterogeneous Trust Management. In (Hühnlein, D.; Roßnagel, H.; Schunck, C. H.; Talamo, M., eds.): Open Identity Summit 2016, 13.-14. October 2016, Rome, Italy. Vol. P-264. LNI, GI, pp. 15–26, 2016.

[EFT94] Ebbinghaus, H.; Flum, J.; Thomas, W.: Mathematical logic (2. ed.) Springer, 1994, ISBN: 978-3-540-94258-0.

[HG06] Hinrichs, T.; Genesereth, M.: Herbrand Logic, tech. rep. LG-2006-02, http://logic.stanford.edu/reports/LG-2006-02.pdf, CA, USA: Stanford University, 2006.

[Ji01] Jim, T.: SD3: A Trust Management System with Certified Evaluation. In: 2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001. IEEE Computer Society, pp. 106–115, 2001.

[KTV10] Küsters, R.; Truderung, T.; Vogt, A.: Accountability: definition and relationship to verifiability. In (Al-Shaer, E.; Keromytis, A. D.; Shmatikov, V., eds.): CCS 2010. ACM, pp. 526–535, 2010.

[MK14] Mödersheim, S.; Katsoris, G.: A Sound Abstraction of the Parsing Problem. In: CSF 2014. IEEE Computer Society, pp. 259–273, 2014.

[Mö19] Mödersheim, S.; Schlichtkrull, A.; Wagner, G.; More, S.; Alber, L.: TPL: A Trust Policy Language. In (Meng, W.; Cofta, P.; Jensen, C. D.; Grandison, T., eds.): IFIPTM 2019. Vol. 563. IFIP Advances in Information and Communication Technology, Springer, pp. 209–223, 2019.

[NPW02] Nipkow, T.; Paulson, L. C.; Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Springer, 2002.

[SBT19] Schlichtkrull, A.; Blanchette, J. C.; Traytel, D.: A verified prover based on ordered resolution. In (Mahboubi, A.; Myreen, M. O., eds.): CPP 2019. ACM, pp. 152–165, 2019.

[Su17] Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning 59/4, pp. 483–502, 2017.