Towards an Efficient and Language-Agnostic Compliance Checker for Trust Negotiation Systems

Adam J. Lee and Marianne Winslett
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{adamlee, winslett}@cs.uiuc.edu

Abstract

To ensure that a trust negotiation succeeds whenever possible, authorization policy compliance checkers must be able to find all minimal sets of their owners' credentials that can be used to satisfy a given policy. If all of these sets can be found *efficiently* prior to choosing which set should be disclosed, many strategic benefits can also be realized. Unfortunately, solving this problem using existing compliance checkers is too inefficient to be useful in practice. Specifically, the overheads of finding all satisfying sets using existing approaches have been shown to rapidly grow exponentially in the size of the union of all satisfying sets of credentials for the policy, even after optimizations have been made to prune the search space for potential satisfying sets.

In this paper, we describe the CLOUSEAU compliance checker. CLOUSEAU leverages efficient pattern-matching algorithms to find all satisfying sets of credentials for a given policy in time that grows as $O(NA)$, where N is the number of satisfying sets for the policy and A is the average size of each satisfying set. We describe the design and implementation of the CLOUSEAU compliance checker, evaluate its performance as the number and size of satisfying sets for a given policy varies, and show that it vastly outperforms existing approaches to finding all satisfying sets of credentials. We then present a method for automatically compiling *RT* policies into a format suitable for analysis by CLOUSEAU and prove the correctness and completeness of this compilation procedure.

1 Introduction

In trust negotiation approaches to authorization (e.g. [5, 9, 17, 22, 23, 24]), resources are protected by attribute-based access policies, rather than explicit access control lists. Entities use cryptographic credentials issued by third-party attribute certifiers (e.g., professional organizations, employers, or government bodies) to prove various attributes about themselves. Since these attributes might themselves be considered sensitive, they can optionally be protected by release policies placing constraints on the individuals to whom they can be disclosed. As such, a trust negotiation session evolves into a bilateral and iterative exchange of policies and credentials with the end goal of developing new trust relationships on-the-fly. Because these types of systems allow resource administrators to specify the *intention* of a policy, rather than its logical *extension* (i.e., an access control list), authorized entities can gain access to available resources without requiring that their identity be known a priori.

The design of robust and highly-available trust negotiation systems hinges on the availability of efficient policy *compliance checkers*. Given a policy p and a set C of credentials, the compliance checker is responsible for determining one or more minimal subsets of C that satisfy p. We call these minimal subsets *satisfying sets* of credentials. To ensure that trust negotiation protocols establish trust whenever possible, the compliance checkers used by these systems must be capable of finding all satisfying sets of credentials for a given policy. This enables the negotiator to attempt alternate means of establishing trust in the event that the initially-chosen negotiation tactic leads to a deadlock or other dead end.

Trust negotiation is intrinsically a strategy-driven

process in which the participants each attempt to advance the state of the protocol while maximizing their own particular goals [27]. For instance, the so-called *eager* and *parsimonious* negotiation strategies allow negotiation participants to balance a trade-off between negotiation speed and privacy by choosing to disclose either all credentials that a remote party is authorized to view or only those that have been deemed relevant to satisfying the policy at hand, respectively [23]. If a negotiation participant is able to determine *all* satisfying sets of credentials for a given policy a priori, much finer-grained strategies can be employed. For example:

- If entities assign point values to individual credentials that indicate each credential's level of sensitivity (e.g., as in [26]), the negotiation process can respond to a given policy by disclosing the satisfying set with the lowest overall sensitivity.
- In the event that an entity has digital credentials representing memberships in organizations that may lead to various types of discounts or preferential treatment (e.g., AAA, AARP, or frequent flyer credentials), they could employ a negotiation strategy that discloses satisfying sets containing these types of credentials first.
- In some cases, entities might wish to minimize the cumulative number of credentials disclosed over multiple rounds of a given trust negotiation session; a simple greedy algorithm could be used to determine the satisfying set that minimizes the overall number of credentials disclosed.
- A party might wish to steer the negotiation in the direction most likely to minimize its duration. For example, a server may wish to lead the negotiation in the direction that the analysis of logs of past negotiations has shown to be the way that most users gain access.

Existing compliance checkers designed for trust negotiation policy languages find at most one satisfying set of credentials at a time, but can be operated in an iterative manner to discover alternate satisfying sets in the event that the first set found leads to a negotiation path that fails to establish trust. While this iterative approach to discovering satisfying sets is sufficient to ensure the completeness of trust negotiation protocols, it is a very slow way to discover all satisfying sets at once. As a result, it is unrealistic to use this approach as the basis for the types of strategies discussed above. Specifically, the overheads of finding all satisfying sets using such an approach have been shown to grow exponentially in the size of the union of all satisfying sets of credentials for the policy, even after optimizations have been made to prune the search space for potential satisfying sets [21].

In this paper, we describe the design and implementation of Clouseau, a highly-efficient and policy language-agnostic compliance checker for trust negotiation systems. Rather than discovering satisfying sets of credentials using a top-down proof construction system, Clouseau solves the policy compliance checking problem by compiling policies into a format that can be efficiently analyzed using solutions to the many pattern/many object pattern match problem. Given a set of patterns and a set of objects, algorithms for solving this problem find all patterns matched by subsets of the provided objects. Internally, Clouseau represents access control policies as patterns specifying constraints on the credentials, credential chains, and uncertified claims (e.g., phone numbers, addresses, etc.) that must be presented to gain access to a particular resource. The Rete algorithm [13] is then used to find all satisfying sets by efficiently matching objects representing a user's credentials and claims against these patterns. Overall, Clouseau makes several important contributions related to the compliance checker problem for trust negotiation systems:

- Clouseau requires only tens of milliseconds, on average, to determine *every* satisfying set of credentials associated with a reasonably-sized policy; this is comparable to the time required by existing trust negotiation compliance checkers to find *one* satisfying set.
- To the best of our knowledge, Clouseau represents the first trust negotiation compliance checker capable of finding all satisfying sets of credentials for a given policy with time overheads that scale as $O(NA)$, where N is the number of satisfying sets for a policy and A is the average size of each satisfying set. Previous solutions to this problem have running time overheads that grow exponentially in the size of the union of all satisfying sets. As a concrete example, the iterative solution presented in [21] takes over 10 seconds to find two overlapping satisfying sets containing a total of 20 credentials, while Clouseau finds the same satisfying sets in approximately 40 ms.

- In the worst case, the number of satisfying sets for a given policy can be exponential in the size of the policy. However, CLOUSEAU's performance remains reasonable even when policies become inordinately complex. For example, CLOUSEAU can find 512 satisfying sets each of size 18 in approximately one second; we have not found policies of this complexity being used in practice or mentioned elsewhere in the research literature. In Section 6, we show that policies as complex as even the most complicated policies used in Becker's formalization of the security requirements for the UK's electronic health records service [2] can be analyzed by CLOUSEAU in under 100 ms.
- Since it can efficiently find all satisfying sets of credentials for a given policy, CLOUSEAU makes the use of "smarter" trust negotiation strategies practical. In Section 6, we show that CLOUSEAU is very fast at finding the minimum-weight (e.g., least-sensitive) satisfying set of credentials for a given policy.
- The design of a single highly-optimized compliance checker capable of analyzing policies written in *any* policy language would allow entities to write policies without worrying about the costs of analyzing them. CLOUSEAU compiles policies written in high-level policy languages into an intermediate representation that specifies constraints on the actual credentials used to satisfy a given access control policy, which it can then efficiently analyze. We present a process for automatically compiling *RT* [18] policies into a format that can be analyzed by CLOUSEAU and prove the correctness and completeness of this compilation procedure. Since policies written in all existing trust negotiation policy languages are satisfied by the same types of evidence, we conjecture that equivalent compilation mechanisms could be specified for the other languages as well.

The rest of this paper is organized as follows. We begin with a discussion of related work in Section 2. We then formally define the specific instance of the more general policy compliance checking problem solved by CLOUSEAU in Section 3. Section 4 describes the Rete algorithm, presents the design and implementation of the CLOUSEAU compliance checker, and discusses the internal representation of policies and evidence used by CLOUSEAU. We present a procedure for automatically compiling *RT* policies into a format suitable for analysis by CLOUSEAU in Section 5. In Section 6 we conduct a series of experiments to evaluate the performance of CLOUSEAU and compare its benefits and limitations to those of other compliance checking approaches. Lastly, we present our conclusions and directions for future work in Section 7.

2 Related Work

In [20], the authors broadly classify policy compliance checkers for trust management and trust negotiation systems into three categories. They first define *type-1* compliance checkers as functions that return only a Boolean result indicating whether the policy in question was satisfied. Compliance checkers for the PolicyMaker [7, 8] and KeyNote [6] trust management systems are included in this first category, as the non-iterative nature of these systems makes the discovery of *why* a particular access was permitted superfluous; simply knowing that the compliance checker can construct a formal proof of authorization is sufficient. The CPOL compliance checker [10] is a highly-optimized compliance checker designed to enforce access policies on centralized resources in high-throughput environments, such as location-detection systems. CPOL uses aggressive caching and other optimizations to achieve incredible performance, but does not return evidence supporting the binary decisions that it makes. Lastly, the compliance checker for Ponder [12], which is used for policy-based network administration, also falls into this first category.

Type-2 compliance checkers return one satisfying set of credentials in addition to a Boolean value in the case that a policy is found to be satisfied. The compliance checker used by the REFEREE system [11] is capable of returning such justifications, though it need not do so. It is important to note that the ability to associate at least one satisfying set of credentials with a compliance checker decision is *required* by the trust negotiation process, as otherwise individuals could not determine which credentials should be sent to their negotiation partner after they determine that a remote policy can be satisfied. As such, the compliance checkers for the XML-based policy languages $\mathcal{X}$-TNL [4] and the IBM Trust Policy Language [15] fall into this category, as do compliance checkers for existing logic-based trust negotiation policy languages, such as Cassandra [3] and the language presented by Koshutanski and Massacci in [16].

Lastly, *type-3* compliance checkers are defined as functions that return every minimal set of credentials

that can be used to satisfy a particular policy. To date, no trust negotiation compliance checkers have been developed expressly for this purpose, although significant strategic benefits could be recognized by such a compliance checker. In [21], Smith et al. discuss several important uses of this type of compliance checker and describe the Satisfying Set Generation (SSgen) algorithm for discovering all satisfying sets for a given policy using a type-2 policy compliance checker. They show that when policies are expressed in disjunctive normal form (DNF), then a number of clever optimizations can be made to prune the state space that must be searched for satisfying sets of credentials. They then evaluate the performance of an implementation of the SSgen algorithm that used the IBM TE compliance checker [15] as the base type-2 compliance checker.

Figure 1 is a reproduction of the results presented in [21] depicting the running times of the SSGen algorithm. The three most interesting cases in which the SSgen algorithm was evaluated include the cases in which (i) a policy has one satisfying set of size U, (ii) a policy has U satisfying sets of size one, and (iii) a policy has two satisfying sets, each of size $\frac{3U}{4}$. In all cases, U represents the size of the union of all satisfying sets and was varied between 1 and 24. In case (i), the SSgen algorithm scaled linearly with U in the sub-second time range. In cases (ii) and (iii), the SSgen algorithm's running time increased exponentially with U and rapidly became impractical.

Clouseau improves upon this previous work by compiling trust negotiation policies into an intermediate representation that can be analyzed using efficient pattern matching algorithms. This non-traditional approach to theorem proving greatly optimizes the process of finding all possible satisfying sets of credentials for a given policy. Further, because existing trust negotiation policy languages can be compiled into a format that can be analyzed using Clouseau, developers can optimize trust negotiation runtime systems while still allowing policy writers to continue to use existing high-level policy languages.

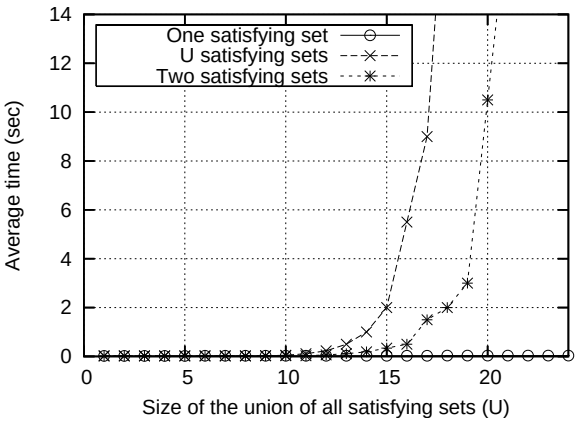

Figure 1: Running time of the SSGen algorithm as a function of the size of the union of all satisfying sets.

3 Problem Definition

At its most basic level, a trust negotiation session is a bilateral and iterative exchange of access policies and evidence conducted to establish mutual trust between two parties. For example, a student that wishes to access a resource connected to a computing grid might be returned a policy stating that only full-time students at accredited universities can access that resource. Prior to proving her enrollment status to the resource operator, the student might first require that the resource operator prove that it is operated by either an NSF-sponsored organization or an organization that is a member of the Better Business Bureau. Digital credentials, such as X.509 certificates, are the most common form of evidence used by these protocols, although uncertified claims (as in [5, 9]), proof fragments (as in [1, 25]), or trust tickets (as in [5]) could also be exchanged. In the remainder of this paper, we will define $\mathcal{E}$ as the set of all such evidence and $\mathcal{P}$ as the set of all policies.

Formally, a compliance checker is defined as a function $cc : 2^{\mathcal{E}} \times \mathcal{P} \rightarrow \mathcal{R}$ that takes some set of evidence and a policy and determines whether (and possibly how) this policy is satisfied by the specified set of evidence. The exact definition of *satisfaction* is specific to the policy language being used. For example, in any language with a model theory, we say that a set E of evidence satisfies a policy p if in every model where E is true, p is also true. In Clouseau, a policy is specified as one or more patterns placing constraints on the credentials and other evidence that must be presented to gain access to a particular resource. We say that such a policy is satisfied if at least one of these patterns can be matched by the set of objects describing the credentials and other evidence possessed by a given entity. As we will see in Section 5, proving the correctness of our RT to Clouseau policy compilation process involves proving an equivalence between these two concepts of satisfaction. That is, we must show that an Clouseau

pattern-match occurs if and only if the *RT* rules of inference draw the same conclusion.

When a compliance checker is invoked to check the satisfaction of some policy protecting a local resource r, it will be given a set of evidence provided by the remote entity wishing to access r. In this case, the compliance checker need only return a Boolean value indicating whether the policy was satisfied (i.e., $\mathcal{R} \equiv \mathbb{B}$). However, if local credentials are used in an attempt to satisfy a remote policy p, then $\mathcal{R} \equiv 2^{2^{\mathcal{E}}}$. That is, the compliance checker must return zero or more sets of local evidence that minimally satisfy p so that the local entity knows which local evidence can be sent to the remote entity to gain access to the resource protected by p. We say that some set E of evidence *minimally satisfies* a policy p if no proper subset of E also satisfies p. A compliance checker capable of recognizing *all* possible subsets of the local evidence that minimally satisfy a given policy is required to ensure that a trust negotiation protocol will establish trust whenever possible and can also afford its user a number of strategic advantages. Therefore, our focus in this paper is to efficiently solve the following specific instance of the more general policy compliance checking problem:

The Type-3 Compliance Checker Problem. *Given a set $E \in \mathcal{E}$ of evidence and a policy $p \in \mathcal{P}$, find all distinct subsets sets $e_1, \ldots, e_n$ of E that minimally satisfy p.*

4 Design of Clouseau

In this section we discuss the design of the Clouseau compliance checker, which we have designed to efficiently solve the type-3 compliance checker problem. We begin by showing that this problem naturally translates into an instance of the many pattern/many object pattern match problem. This is followed by a discussion of the technical details of our implementation of the Clouseau compliance checker, including an overview of the Rete algorithm, which is used by Clouseau. Lastly, we conclude this section with an overview of Clouseau's internal representation of trust negotiation evidence and policies.

4.1 Design Approach

To date, most policy languages for trust negotiation are modeled using logic programming approaches, as the formal semantics of logic programs are well understood. For example, policies in Cassandra [3], PSPL [9], the language used by Koshutanski and Massacci [16], *RT* [18], and PeerAccess [25] are all specified in this manner. Even some XML-based languages, such as TPL, are formally modeled using logic programming approaches [15]. Not surprisingly, the policy compliance checking approaches used for these types of languages have leveraged traditional theorem-proving techniques. When the compliance checker is invoked on a policy p with a set E of evidence, the underlying theorem prover stores the set of evidence $e \subseteq E$ used during the construction of a single proof that p was satisfied. Rather than simply returning the Boolean value `true`, the set e is also returned by the compliance checker to provide support for its decision.

Although this type of theorem-proving approach to the general compliance checker problem is natural given the logical foundations of trust management, it is not the only way in which this problem can be formulated. In fact, using this type of approach to solve the type-3 compliance checker problem is unappealing, as theorem provers in general are designed to find a single proof that a fact is valid (i.e., that a policy is satisfied) and search for alternate proofs only when a given proof attempt fails. As an alternate approach, we have recognized that the the type-3 compliance checker problem is actually an instance of the more general many pattern/many object pattern matching problem [13]:

The Many Pattern/Many Object Pattern Matching Problem. *Given a set of patterns and a set of objects, determine all of the ways in which the set of objects can be used to match any of the specified patterns.*

Clearly, if credentials and other evidence are treated as objects and policy clauses are treated as patterns, an efficient solution to this problem could likely lead to an efficient solution to the type-3 compliance checker problem. This problem has been studied previously by the artificial intelligence community, as it is central to the design of efficient production system interpreters. As a result, efficient algorithms, such as Rete [13] and TREAT [19], have been developed to solve this problem. The Rete algorithm is optimized for instances of the many pattern/many object pattern matching problem in which (i) patterns are compilable, (ii) all objects remain constant once inserted into the Rete engine's working memory, and (iii) the set of objects changes rel-

atively slowly [13]. Note that trust negotiation policies are all compilable, as they are designed to enable automated reasoning, rather than human interpretation. Further, credentials and other evidence remain constant once obtained. For example, modifying or tampering with a digital certificate invalidates its attached issuer signature. Lastly, the set of local evidence changes very infrequently and the set of remote evidence grows monotonically as the protocol proceeds. We therefore use the Rete algorithm as the basis for the CLOUSEAU compliance checker.

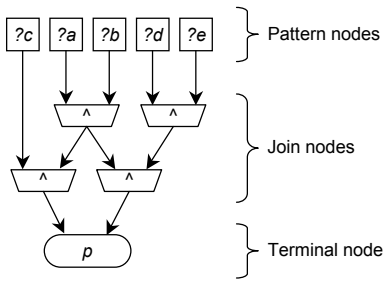

Figure 2: An example Rete network.

4.2 The Rete Algorithm

We now provide an overview of the Rete algorithm and highlight its benefits in solving the type-3 compliance checker problem. Space limitations prevent a full discussion of the specifics of the Rete algorithm; interested readers should consult [13] for more information. At a high level, the Rete algorithm works by forming a network of nodes that represent one or more matching tests found in the specified patterns. *Pattern nodes*, which are also known as *one-input nodes*, are used to match single objects stored in the *working memory* of the Rete engine. In the case of CLOUSEAU, these objects represent constraints on individual pieces of trust negotiation evidence such as digital certificates and uncertified claims. The outputs of these pattern nodes can then be fed into one or more *join nodes*, which are used to build more complex patterns consisting of conjunctions of basic patterns and constraints existing between the objects matched by these patterns. In our formulation of the type-3 compliance checker problem, join nodes are used to specify conjunctions of basic credentials as well as inter-credential constraints (i.e., chains of trust or credential delegations).

A collection of pattern nodes and join nodes forms a directed acyclic graph whose sink nodes are called *terminal nodes*. As matches occur in the Rete network, information describing the match is propagated along the edges of the graph. When a terminal node is reached, an event is triggered that signifies that a complete match has occurred. In CLOUSEAU, this implies that a given policy has been satisfied and enables the compliance checker to extract the set of evidence that led to this particular policy satisfaction. Since information is propagated along all possible edges in the Rete network, all satisfying sets are found by the Rete algorithm.

Figure 2 is an illustration of a Rete network for the policy $p \leftarrow a \wedge b \wedge (c \vee (d \wedge e))$, which is represented internally as the pair of Horn clauses $p \leftarrow a, b, c$ and $p \leftarrow a, b, d, e$. Square boxes represent pattern nodes, the trapezoids are join nodes, and the oval node is a terminal node that represents the satisfaction of the policy p. Note that distinct patterns are matched at most once, despite appearing in multiple Horn clauses. Further, join nodes can be shared between multiple clauses of the policy.

Another benefit of the Rete approach is that the network maintains state between invocations, which greatly minimizes the number of times that the working memory is iterated over as multiple policies are matched. That is, each object in the Rete engine's working memory is matched against each pattern node at most one time and the results of this matching operation are saved. For instance, if the policy $p' \leftarrow a \wedge b \wedge (c \vee f)$ is added to the working memory of the Rete engine in Figure 2, the Rete engine needs only to check for the existence of credential f, as any matches for $a \wedge b$ and $a \wedge b \wedge c$ were found and memoized during the analysis of the policy p. These types of optimizations further help make the Rete algorithm an efficient approach to solving the type-3 compliance checker problem.

4.3 Implementation

We now describe our implementation of CLOUSEAU, a fully-functional compliance checker that leverages the Rete algorithm to efficiently solve the type-3 compliance checker problem. Our goal in designing CLOUSEAU was *not* to propose a new trust negotiation policy language, but rather to explore the design of efficient solutions to the type-3 compliance checker problem. Therefore, rather than designing CLOUSEAU to check the satisfaction of policies specified in one particular policy language (e.g., Cassandra, *RT*, TPL, or $\mathcal{X}$-TNL), we instead focus on designing a more general-purpose compliance checker.

Ultimately, the access control policies used by trust negotiation systems are satisfied by digital certificates or other such evidence presented by participants in the negotiation process. To this end, the policy patterns used to construct the Rete network analyzed by Clouseau specify constraints on the actual evidence (e.g., certificates, certificate chains, and claims) necessary to gain access to a particular resource. This is in contrast to higher-level policy languages, such as *RT*, which have syntactic constructs to represent concepts such as delegation natively. In Section 5, we discuss a process through which *RT* policies can be automatically compiled into the native rule format used by Clouseau for analysis. Since all trust negotiation policies are eventually satisfied by the same types of evidence, we believe that equivalent compilation procedures could be derived for other higher-level policy languages as well.

Our implementation of Clouseau was developed using the Java programming language. At a high level, Clouseau takes a set E of evidence and an access control policy p and uses an implementation of the Rete algorithm provided by the Jess expert system [14] to determine all of the ways in which subsets of E can satisfy p. The running time of this Rete implementation scales, on average, linearly with the size of its working memory [14]. This implies that Clouseau's running time scales as $O(NA)$, where N is the number of satisfying sets for a policy and A is the average size of each satisfying set; we confirm this result experimentally in Section 6. Our implementation consists of a Jess specification defining the internal representations of evidence and several useful functions for reasoning about credential chains, and a larger Java code base responsible for examining various types of evidence, translating evidence into objects that can be instantiated within Clouseau, and creating and querying the Rete network used by Clouseau. We now discuss the internal representation of evidence used by Clouseau as well as the specification of access control policies.

Evidence Representation

Because Jess provides a general-purpose implementation of the Rete algorithm, it has no way of representing or reasoning about trust negotiation evidence natively. We therefore had to define several *object templates* that represent key types of evidence inside of the Rete engine's working memory. The current implementation of Clouseau supports the use of digital certificates, certificate chains, and uncertified claims as forms of evidence; adding support for other types of evidence, such as Trust-$\mathcal{X}$ trust tickets, would be a relatively straightforward process.

```
;; Used to describe digital certificates and
;; other credentials
(deftemplate credential
  (slot id)
  (slot issuer)
  (slot subject)
  (slot fingerprint)
  (slot owned (default FALSE))
  (slot map (default (new java.util.HashMap))))

;; Contains an ordered list of credentials making
;; up a credential chain
(deftemplate credential-chain
  (multislot credentials))

;; Claims are stored as attribute/value pairs
(deftemplate claim
  (slot id)
  (slot type)
  (slot value))
```

Figure 3: Internal evidence representations used by Clouseau.

Clouseau makes use of an extensible credential type hierarchy that allows trust negotiation implementations to add support for new credential types to Clouseau without modifying its code base. Trust negotiation implementations are responsible for validating any proof-of-ownership challenges associated with a given credential and for forming and verifying the credential chains passed into Clouseau. Once a collection of credential chains has been passed into Clouseau, they are translated into instances of the `credential` and `credential-chain` object types described in Figure 3. Uncertified claims provided as evidence to Clouseau are represented internally as instances of the `claim` object type.

Objects of type `credential` are generated by extracting information from a given cryptographic credential using methods of the abstract credential class at the top of Clouseau's credential type hierarchy. Each `credential` structure is assigned a unique identifier and contains fields describing the credential's subject and issuer, a cryptographic fingerprint of the credential, a Boolean value indicating whether proof-of-ownership of the credential was verified, and a map containing key/value pairs describing attributes (e.g., job function, hire date, etc.) or other information (e.g., expiry date) embedded in the credential. Internally, credential chains are represented as ordered lists of unique identifiers satisfying two invariants: (i)

```
;; This policy is satisfied by graduate students at ABET-
;; accredited universities who provide an email address
;; that can be used for future correspondence.
(defrule rule-grad-student
  ;; Find a certificate chain leading from a university
  ;; to a graduate student
  (credential (id ?iuniv) (subject ?suniv))
  (credential (id ?istud) (owned true) (map ?mstud))
  (test (eq "Graduate Student" (?mstud get "Type")))
  (credential-chain (credentials $?cstud))
  (test (is-root ?iuniv ?cstud))
  (test (is-leaf ?istud ?cstud))

  ;; Find a certificate chain leading from ABET to
  ;; the university found above.
  (credential (id ?iabet) (fingerprint
              "38:1A:42:E9:00:7D:19:41:AC:66:F2:EF:12:E6:B4:A1"))
  (credential (id ?icert) (map ?mcert)
              (subject ?scert &: (eq ?scert ?suniv)))
  (test (eq "Accredited University" (?mcert get "Type")))
  (credential-chain (credentials $?ccert)
  (test (is-root ?iabet ?ccert))
  (test (is-nth ?icert 2 ?ccert))

  ;; See if the student provided an email address
  (claim (id ?iemail) (type "Email") (value ?v))
=>
  (assert (satisfaction (resource-name server)
          (credentials ?cstud ?ccert)
          (claims ?iemail))))
```

Figure 4: An example Clouseau policy.

the credential referenced by the identifier at index 0 in the list is the root of the credential chain and (ii) the credential referenced by the identifier at index $i > 0$ was issued by the owner of the credential at index $i - 1$. Uncertified claims are represented as attribute/value pairs associated with a unique identifier field.

Policy Specification

In Clouseau, access control policies are specified as collections of Jess rules that place constraints on the credentials, credential chains, and uncertified claims that must be presented to gain access to a particular resource. In the remainder of this section, we provide a overview of the Clouseau policy syntax by discussing an example access control policy. We note that only a very small subset of the Jess language is needed to specify Clouseau policies. In particular, we use only the language constructs discussed in this section.

Figure 4 is an example access control policy designed to allow graduate students at universities accredited by the Accreditation Board for Engineering and Technology (ABET) to access some resource "server," provided that they disclose an email address that can be used for future correspondence. Because of the relatively simple nature of this policy, it can be specified using a single Jess rule. Rules consist of two parts: a left hand side (LHS) specifying patterns that must be matched by objects in the working memory of the Rete engine and a right hand side (RHS) that specifies some action to be taken if the pattern in the LHS of the rule is completely matched. These two parts of a rule are separated by the `=>` token.

The LHS of the rule in Figure 4 consists of three groups of patterns that must be matched by objects in the working memory of the Rete engine representing trust negotiation evidence. The first group determines whether there exists a certificate chain whose leaf node is a certificate of type "Graduate Student." The first line of this group is a pattern that matches any credential and saves the values of its unique identifier and subject string in the variables `?iuniv` and `?suniv`, respectively. The second line in this grouping is a similar pattern that matches any credential whose ownership was proven during the trust negotiation protocol. The third line in this grouping enforces the constraint that the second credential matched has a "Type" field whose value is "Graduate Student." The last three lines of the first grouping require that the two matched credentials must exist in a credential chain whose authenticity was verified by the trust negotiation implementation. Note that a given pattern need not constrain all fields of the `credential` object type.

The second group of patterns is similar to the first, in that it also establishes the existence of another credential chain. The first two lines of this of this group form a pattern that matches only the certificate whose cryptographic fingerprint is represented by the hexadecimal string `38:1A:42:E9:00:7D:19:41:AC:66:F2:EF:12:E6:B4:A1`, which is the fingerprint of the (fictitious) certificate used by ABET to issue university accreditations. The third, fourth, and fifth lines of this group form a pattern that matches any credential that has a subject field that is the same as that of the root of the first credential chain (i.e., the university), and has a "Type" field whose value is "Accredited University." The last two lines of this pattern place the constraint that the ABET credential must form the root of a credential chain of length two that ends with the university's accreditation certificate. The last group of constraints consists of a single line specifying that the user needs to also disclose an uncertified claim of type "Email" containing his or her email address, which will pre-

sumably be stored for future correspondence.

In general, the RHS of an CLOUSEAU policy can either assert an intermediate result that can be used as input to other rules, or assert a `satisfaction` object describing one way in which a particular policy was satisfied. The former action might be taken if a complicated policy has several paths to satisfaction that each require a common prefix to be matched; we will see examples of this in Section 5.2. The policy in Figure 4 takes the latter action and asserts a `satisfaction` object containing the set of credentials and the single claim used to satisfy the policy.

We note that despite a simple policy specification syntax, CLOUSEAU policies can quickly become large and difficult to understand due to the number of constraints that might exist between elements of a credential chain or fields of credentials in different chains. However, we do not view this as a limitation of CLOUSEAU. We do not expect that users of CLOUSEAU will choose to specify policies using this subset of Jess. Rather, we view the native policy representation used by CLOUSEAU as being akin to assembly language in that it provides a representation of a potentially-complex expression that can be efficiently analyzed. We expect that users will specify policies in higher-level trust negotiation policy languages and that these policies will be automatically compiled into CLOUSEAU's native language, much as programs written in high-level programming languages are compiled into assembly code prior to execution.

5 Analyzing RT Policies

In this section, we discuss a method for automatically compiling *RT* policies into a format suitable for analysis by CLOUSEAU. For ease of exposition, we begin by discussing a compilation process for RT_0 policies, which support the use of unparameterized roles, and prove the correctness and completeness of this process. We then provide an intuition for how this process can be extended to support RT_1 policies allowing the use of parameterized roles. The ability to correctly analyze these types of policies is a necessary step towards establishing CLOUSEAU as a general-purpose solution to the type-3 compliance checker problem. Since all trust negotiation policies are eventually satisfied by the same types of evidence, we conjecture that equivalent compilation procedures could be devised for other higher-level policy languages as well.

5.1 RT_0 Policy Syntax

Recall from [18] that RT_0 is the most basic language in the *RT* family of trust management languages. As in all of the *RT* languages, principals are identified by means of identity certificates. RT_0 roles are defined simply as strings identifying the name of the role and cannot be parameterized. Policy statements in RT_0 are expressed as one or more of these role definitions and are encoded as *role definition credentials* signed by the author of the role definition. There are four basic types of role definition credentials in RT_0:

Simple Member A role definition of the form $K_A.R \leftarrow K_D$ encodes the fact that principal K_A considers principal K_D to be a member of the role $K_A.R$.

Simple Containment A role definition of the form $K_A.R \leftarrow K_B.R_1$ encodes the fact that principal K_A defines the role $K_A.R$ to contain all members of the role $K_B.R_1$, which is defined by principal K_B.

Linking Containment A role definition of the form $K_A.R \leftarrow K_A.R_1.R_2$ is called a *linked role*. This defines the members of $K_A.R$ to contain all members of $K_B.R_2$ for each K_B that is a member of $K_A.R_1$.

Intersection Containment The role definition $K_A.R \leftarrow K_{B_1}.R_1 \cap \cdots \cap K_{B_n}.R_n$ defines $K_A.R$ to contain the principals who are members of each role $K_{B_i}.R_i$ where $1 \leq i \leq n$.

These four basic types of role definitions can be used to define a wide range of access control policies. For example, the following RT_0 role definitions express an access control policy requiring that entities accessing a given resource be employees of a SuperGrid member organization:

$$Provider.service \leftarrow Provider.partner.employee$$
$$Provider.partner \leftarrow SuperGrid.memberOrganization$$

If a principal, Alice, could provide credentials proving the statements $SuperGrid.memberOrganization \leftarrow AliceLabs$ and $AliceLabs.employee \leftarrow Alice$, she could satisfy the policy formed by the above two role definitions and gain access to the protected service.

5.2 Compiling RT_0 Policies

In RT_0, policies are collections of role definition credentials. Therefore, we must preprocess the set of

```
;; Template to store role membership information
(deftemplate is-member
  (slot role)
  (slot roleMgr)
  (slot roleSubj)
  (multislot credentials))

;; Code to detect role memberships via the presence of simple
;; membership policy credentials.  I.e., this can prove that
;; K_A.R <- K_B
(defrule member-of
  ;; Match K_B's identity certificate
  (credential (id ?kb) (fingerprint ?fkb))

  ;; Prove that K_A says that K_B is in role R
  (credential (id ?ka) (fingerprint ?fka))
  (credential (id ?r) (map ?m))
  (test (eq ?fka (?m get "roleMgr")))
  (test (eq ?fkb (?m get "roleSubj")))
  (credential-chain (credentials $?c))
  (test (is-root ?ka ?c))
  (test (is-nth ?r 2 ?c))
=>
  (assert (is-member (role (?m get "role"))
                     (roleMgr (?m get "roleMgr"))
                     (roleSubj (?m get "roleSubj"))
                     (credentials ?ka ?kb ?r))))
```

Figure 5: Base policy enabling CLOUSEAU to determine role membership via the use of simple membership credentials.

credentials provided to CLOUSEAU as input in order to generate the set of policy rules that CLOUSEAU will attempt to satisfy. Since CLOUSEAU examines the actual credentials used to hold RT_0 assertions, rather than these higher-level RT_0 assertions, we must make a few assumptions regarding the format of these credentials.

1. We assume that principals in the system are identified by the fingerprint of their identity certificates. Text strings such as "*ABET.accredited*" will be used during the discussion of abstract policies, although such assertions are actually shorthand for statements such as "`38:1A:42:E9:00:7D:19:41:AC:66:F2:EF:12:E6:B4:A1`.*accredited*." When defining CLOUSEAU policies later in this section, we will use the notation `<K_A>` to denote the fingerprint of K_A's identity certificate.
2. Simple membership role definition credentials of form $K_A.R \leftarrow K_B$ are assumed to have the attributes "roleMgr," "role," and "roleSubj" set to the values `<K_A>`, `R`, and `<K_B>`, respectively.
3. Role definition credentials are valid if and only if they are signed by the principal identified at the head of the credential. For example, the simple membership credential $K_A.R \leftarrow K_B$ is considered valid if and only if it is signed by the principal K_A.

Given the above assumptions regarding credential format, we now describe an algorithm for generating a CLOUSEAU policy p' that is equivalent to an RT_0 policy p consisting of the valid role definition credentials $r_1, \ldots, r_n$ and the set of identity certificates $c_1, \ldots, c_m$.

1. Insert the `is-member` template type and the `member-of` rule defined in Figure 5 into p'. The `is-member` object type holds information regarding a particular principal's membership in a particular role. The `member-of` rule asserts an `is-member` object if a simple membership role definition credential of form $K_A.R \leftarrow K_B$ can be found, along with identity certificates for K_A and K_B.
2. Generate the `credential` objects corresponding to the identity certificates $c_1, \ldots c_m$ and insert these into the working memory of CLOUSEAU.
3. For each valid role definition credential r_i:
   - Generate the `credential` object corresponding to r_i and insert it into the working memory of CLOUSEAU. Save the "id" field of this object as the variable `<id>`.
   - If r_i is a simple containment credential of form $K_A.R \leftarrow K_B.R_1$ then insert the following rule into p':

     ```
     (defrule rule-sc-<id>
       (is-member (role "R_1") (roleMgr <K_B>)
                  (roleSubj ?rs) (credentials $?c))
     =>
       (assert (is-member (role "R") (roleMgr <K_A>)
                          (roleSubj ?rs)
                          (credentials ?c <id>)))
     ```

     This rule asserts that a principal is a member of role $K_A.R$ if he is also a member of $K_B.R_1$.
   - If r_i is a linking containment credential of form $K_A.R \leftarrow K_A.R_1.R_2$ then insert the following rule into p':

     ```
     (defrule rule-lc-<id>
       ;; Find a member of R_2
       (is-member (role "R_2") (roleMgr ?r2mgr)
                  (roleSubj ?r2subj) (credentials $?cr2))
       ;; find a member of K_A.R_1
       (is-member (role "R_1") (roleMgr <K_A>)
                  (roleSubj ?r1subj)
                  (credentials $?cr1))
     ```

```
  (test (eq ?r1subj ?r2mgr))
=>
  (assert (is-member (role "R") (roleMgr <K_A>)
                     (roleSubj ?r2subj)
                     (credentials ?cr1 ?cr2 <id>)))
```

This rule asserts that a principal is a member of the role $K_A.R$ if he is a member of the role R_2 defined by some member of $K_A.R_1$.

- If r_i is an intersection containment credential of form $K_A.R \leftarrow K_{B_1}.R_1 \cap \cdots \cap K_{B_k}.R_k$ then insert the following rule into p':

```
(defrule rule-ic-<id>
  (is-member (role "R_1") (roleMgr <K_B_1>)
             (roleSubj ?rs1) (credentials $?cr1))
  ...
  (is-member (role "R_k") (roleMgr <K_B_k>)
             (roleSubj ?rsk &: (eq ?rs1 ?rsk))
             (credentials $?crk))
=>
  (assert (is-member (role "R") (roleMgr <K_A>)
                     (roleSubj ?rs1)
                     (credentials ?cr1 ... ?crk <id>)))
```

This rule asserts that a principal is a member of the role $K_A.R$ if role memberships for each of the roles $K_{B_1}.R_1, \ldots, K_{B_k}.R_k$ can be found. Note that this rule enforces the constraint that these role membership must all refer to the same subject principal.

4. Given the target role for the negotiation, say $K_A.R_t$, insert the following rule into p':

```
(defrule target
  (is-member (role "R_t") (roleMgr <K_A>) (roleSubj ?rs)
             (credentials $?c))
  (credential (fingerprint ?fp &: (eq ?fp ?rs))
              (owned TRUE))
=>
  (assert (satisfaction (resource-name "target")
                        (credentials ?c))))
```

This rule triggers the insertion of a policy satisfaction object whenever an identity certificate with valid proof-of-ownership can be found for a member of the target role $K_A.R_t$.

Intuitively, this compilation process works in a bottom-up fashion, as follows. The `member-of` rule enables Clouseau to conclude that a principal K_B is a member of the role $K_A.R$ if it finds K_B's identity certificate, K_A's identity certificate, and the simple membership role definition certificate declaring K_B to be a member of $K_A.R$. These three credentials are retained as evidence supporting K_B's membership in $K_A.R$. The rules inserted during step 3 of the compilation process can then combine these basic role membership assertions to prove membership in roles defined by more complex expressions (i.e., simple containment, linking containment, and intersection containment).

As role membership assertions are combined by these rules, the credential identifiers stored in these role membership assertions are combined and stored in the newly-concluded role membership assertions. Finally, the `target` rule inserted into p' at step 4 of the compilation process asserts a `satisfaction` object whenever membership in the target role of the negotiation process can be found for a principal who could demonstrate proof-of-ownership of his or her identity certificate. This allows Clouseau to conclude that the policy in question has been satisfied and to extract the credentials used during the satisfaction process. The Rete algorithm ensures that all paths leading to the creation of a `satisfaction` object are explored during the pattern matching process, which implies that all satisfying sets of evidence are discovered by the Clouseau compliance checker. Further, an increase in the size of the RT policy to be analyzed causes only a linear increase in the running time of Clouseau. This is a result of the fact that the size of Clouseau's working memory is increased linearly for each credential analyzed during the policy compilation process (i.e., we add at most one rule to Clouseau for each credential processed).

Figure 6 illustrates the result of applying the above compilation process to the RT_0 access policy described in Section 5.1. For brevity, the `is-member` template and the `member-of` rule in Figure 5 were omitted from this figure, although they would be included in the generated policy. We now make the following claim regarding the correctness and completeness of this policy compilation process. A full proof of this theorem can be found in Appendix A.

Theorem 1 (Correctness and Completeness)**.** *Let $R = \{r_1, \ldots, r_n\}$ be a set of role definition credentials, $C = \{c_1, \ldots, c_m\}$ be a set of identity certificates, $p = R \cup C$ be an RT_0 policy, and let p' be the result of compiling p using the above process.* Clouseau *finds the satisfying set $S \subseteq (R \cup C)$ of credentials for the policy p' if and only if the RT rules of inference can be used on exactly the set S of credentials to prove membership in the target role.*

5.3 Supporting RT_1 Policies

The only feature that RT_1 adds to RT_0 is the ability to parameterize role definitions. For example, rather than requiring Alice's employee credential to be of the form *AliceLabs.employee* $\leftarrow$ *Alice*, it could instead encode other attributes regarding Alice's employment. For example, we could define Alice as the

```
;; Provider.service <- Provider.partner.employee
(defrule rule-1
  (is-member (role "employee") (roleMgr ?r2mgr)
             (roleSubj ?r2subj) (credentials $?cr2))
  (is-member (role "partner") (roleMgr <Provider>)
             (roleSubj ?r1subj)
             (credentials $?cr1))
  (test (eq ?r1subj ?r2mgr))
=>
  (assert (is-member (role "service") (roleMgr <Provider>)
                     (roleSubj ?r2subj)
                     (credentials ?cr1 ?cr2 <id>)))

;; Provider.partner <- SuperGrid.memberOrganization
(defrule rule-2
  (is-member (role "memberOrganization") (roleMgr <SuperGrid>)
             (roleSubj ?rs) (credentials $?c))
=>
  (assert (is-member (role "partner") (roleMgr <Provider>)
                     (roleSubj ?rs) (credentials ?c <id>)))

;; Provider.service is our target role
(defrule target
  (is-member (role "service") (roleMgr <Provider>)
             (roleSubj ?rs) (credentials $?c))
  (credential (fingerprint ?fp &: (eq ?fp ?rs))
              (owned TRUE))
=>
  (assert (satisfaction (resource-name "target")
                        (credentials ?c))))
```

Figure 6: A compiled version of the RT_0 policy discussed in Section 5.1.

President of AliceLabs by defining the following simple membership credential:

$$\textit{AliceLabs.employee(title=“President”)} \leftarrow \textit{Alice}$$

RT_1 role definition credentials can also constrain role memberships based on the values of role parameters. For example, the following simple containment credential declares that only widgets whose price is over $10 are on sale:

$$\textit{Acme.sale} \leftarrow \textit{Acme.widget(price} > \textit{10)}$$

Adding support for the above types of parameterizations and constraints to the policy compilation process described in Section 5.2 is a relatively straightforward process. In fact, we must only (i) provide support for storing parameters and their values in simple membership role definition credentials and (ii) allow the various containment role definition rules generated during the policy compilation process to place constraints on these parameter values. To address (i), we can store the parameters of a given simple membership credential and their corresponding values in the “map” field of the simple membership's Clouseau `credential` object. Further, the `is-member` object template must be extended to include this mapping of parameter names to values.

Addressing point (ii) is a slightly more complicated process involving the generation of Clouseau rules during step 3 of the compilation process. Rather than explain each case in detail, we will instead provide one example compilation rule and claim that the other cases can be handled in a similar fashion. As an example, consider the following simple containment role definition credential:

$$\textit{AliceLabs.seniorManagement} \leftarrow \textit{AliceLabs.employee(hireYr} < \textit{2000, mgt} = \textit{true)}$$

This credential defines members of the “Senior Management” role at AliceLabs to be managers hired before the year 2000. A policy compiler could parse the above type of role definition credential to form the following rule:

```
(defrule rule-<id>
  (is-member (role "employee") (roleMgr <AliceLabs>) (map ?m)
             (roleSubj ?rs) (credentials $?c))
  (test (eq TRUE (?m get "mgt")))
  (test (> 2000 (?m get "hireYr")))
=>
  (assert (is-member (role "seniorManagement") (roleSubj ?rs)
                     (roleMgr <AliceLabs>)
                     (credentials ?c <id>)))
```

Creating this rule automatically involves an extension to the rule generation logic presented in Section 5.2 that adds one `test` clause to the rule for every constraint placed on an attribute value by the RT_1 role definition credential. Since the Rete engine used by Clouseau supports comparison operators such as $>$, $<$, and $=$, it can support the types of constraints allowed by RT_1. As such, it is possible to define an automated procedure for compiling RT_1 policies into a format suitable for analysis by Clouseau.

6 Evaluation

In this section, we evaluate the performance of the Clouseau compliance checker and then discuss the implications of our findings. In particular, we examine the amount of time required to find all satisfying sets of evidence for a given policy in three sets of experiments; each set of experiments was conducted on a 2.5GHz Pentium 4 with 512MB RAM running Linux. The running times reported include all overheads associated with generating a Rete network based on the policy rules provided to Clouseau, creating `credential` and `credential-chain` objects corresponding to the credentials and credential chains

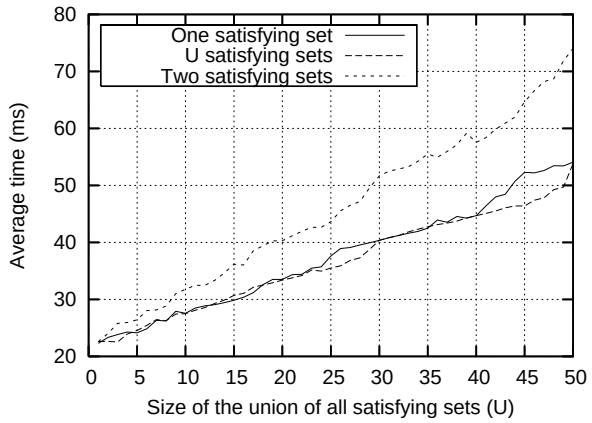

Figure 7: Running time as a function of the size of the union of all satisfying sets.

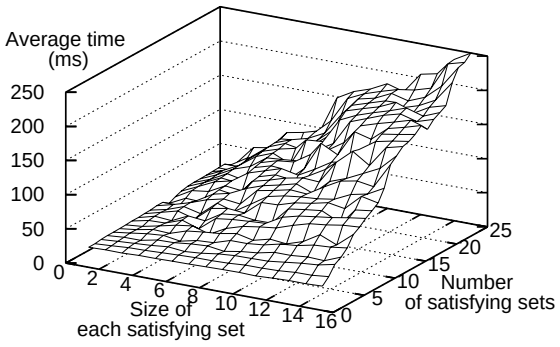

Figure 8: Running time as the number of satisfying sets and the size of each satisfying set varies.

provided to Clouseau, inserting these objects into Clouseau's working memory, and recovering all satisfying sets of credentials. We first repeat the experiments conducted in [21], which explored the overheads associated with using a type-2 compliance checker to solve the type-3 compliance checker problem.

6.1 Experimental Results

The three most interesting cases explored in [21] examined the overheads of using the SSGen algorithm to find all satisfying sets of credentials for a policy in the event that (i) the policy had one satisfying set of size U, (ii) the policy had U satisfying sets of size one, or (iii) the policy had two satisfying sets, each of size $\frac{3U}{4}$. In all cases, U represents the size of the union of all satisfying sets. Recall from Figure 1 in Section 2 that the overheads associated with cases (ii) and (iii) grew exponentially and quickly became impractical. Figure 7 shows the results of running these same tests using the Clouseau compliance checker; note that the y-axis of Figure 1 is labeled in seconds, while the y-axis of Figure 7 is labeled in milliseconds. In our experiments, we varied the size of the union of all satisfying sets (U) between 1 and 50. Each data point in the figure represents the average running time over 100 randomly-generated policies; a new Rete network was constructed for each of these 100 trials as to eliminate any optimizations that might occur as a result of partial network reuse as discussed in Section 4.2. We see that, in all cases, the running time overheads of Clouseau grow linearly with U and never exceed 80 ms to find all satisfying sets. We note also that, unlike the SSgen algorithm, Clouseau does not require that policies be specified in DNF form in order to efficiently find all satisfying sets.

To further explore the running time characteristics of Clouseau, we conducted another experiment designed to more fully examine the types of policies that might be processed by Clouseau in practice. In this experiment, we varied both the number of satisfying sets contained in a particular policy and the size of each satisfying set. For each ⟨number, size⟩ pair, 100 policies were generated at random and examined using Clouseau. The random generation of policies allowed us to explore cases in which satisfying sets overlap with one another to varying degrees. This is important because overlapping satisfying sets will result in shared nodes in the Rete network constructed by Clouseau and, thus, more efficient analysis; examining a random sampling of policies provides us with a more "average case" view of Clouseau's performance. The results of this experiment are shown in Figure 8 and confirm that the performance of Clouseau scales as $O(NA)$, where N is the number of satisfying sets and A is the average size of each satisfying set. We then considered the case in which each credential was assigned a sensitivity value by its owner, as in [26], and ran the above experiments again. Given the satisfying sets detected by Clouseau, it took a trust negotiation strategy, on average, only 0.04 ms to choose the least-sensitive satisfying set to disclose.

We next sought to evaluate the performance of Clouseau in a worst-case scenario. To accomplish

this, we analyzed policies of the form $p \leftarrow (c_1 \vee c_2) \wedge \cdots \wedge (c_{2i-1} \vee c_{2i})$, which can be satisfied in 2^i different ways by a set of $2i$ credentials. Figure 9 shows the results of this experiment, which confirm that the performance of CLOUSEAU continues to scale as $O(NA)$ (note that the x-axis of Figure 9 follows a logarithmic scale). Policies with exponentially-many satisfying sets are unlikely to be used legitimately in practice, but could be formed by attackers wishing to consume inordinate amounts of system resources. Detecting these types of malicious policies in practice is out of the scope of our current paper, and thus we defer that topic to future work. We do note, however, that CLOUSEAU found 512 satisfying sets of credentials in approximately 1 second. This implies that the naive strategy of capping the time spent in the compliance checker might be a reasonable means of detecting these types of attacks in practice, as finding that many satisfying sets of credentials for a non-attack policy seems exceedingly unlikely.

In all of our experiments, a completely new Rete network was created at each invocation of CLOUSEAU. As stated previously, this was done to eliminate the possible benefits of partial network sharing between distinct policies, as network sharing improves the performance of CLOUSEAU. However, some of the primary benefits of the Rete algorithm arise precisely because the state encoded in a particular Rete network can be saved between invocations of the matching algorithm, which reduces the number of times objects (i.e., credentials and other evidence) need to be matched against the pattern nodes (i.e., policy clauses) in the network. If a participant in the trust negotiation process is willing to trade memory for execution time, they could maintain a separate Rete network for each ongoing negotiation constructed using *all* of the policies relevant to that particular negotiation. This would allow them to leverage the statefulness of the Rete algorithm to reduce the number of matching operations required at each invocation of CLOUSEAU and obtain better performance as the negotiation proceeds into later rounds. Further examination of these types of speed versus memory trade offs is an interesting direction for future work.

6.2 Discussion

The experiments discussed above illustrate that our CLOUSEAU implementation performs very quickly, requiring only tens of milliseconds to find all satisfying sets of credentials for policies of a reasonable size. Furthermore, these results experimentally confirm our claim that the running time of CLOUSEAU scales as $O(NA)$, where N is the number of satisfying sets for the policy being analyzed and A is the average size of each satisfying set. As always, the number of satisfying sets for a policy p is in the worst case exponential in the size of p, as was the case in the third set of experiments described above. Even in this case, CLOUSEAU performed admirably, finding 512 satisfying sets of size 18 in just one second; for ordinary policies, CLOUSEAU will find all satisfying sets in about as much time as is needed for a single disk access.

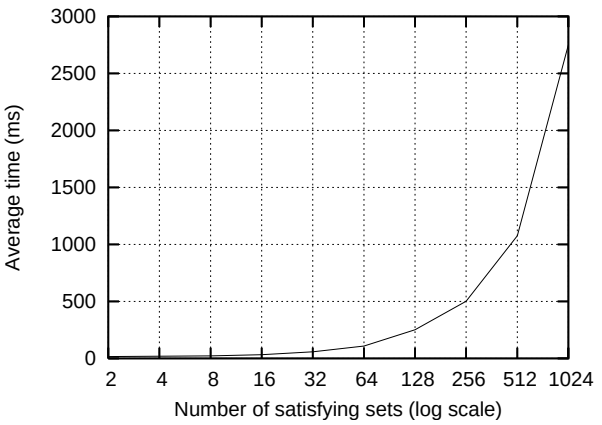

Figure 9: Time required to find all 2^i satisfying sets of size $2i$ for $i = 1, \ldots, 10$.

The largest trust management case study to date is Becker's formalization of the security policies required by the electronic health record service that is being proposed by the United Kingdom's National Health Service [2]. This service, also known as the "Spine," aims to make electronic patient records available to medical personnel, patients, and their designated agents and includes provisions for ensuring the confidentiality of patient records. In [2], Becker completely specifies a collection of Cassandra [3] policies for the NHS Spine and its related services that comply with all available NHS documents describing the requirements for the Spine. The complete Cassandra specification includes definitions of 375 policy rules, 71 roles, and 12 actions that can be taken in the system. Each of the rules is a Horn clause (i.e., a strict conjunction), although often several rules will have the same head. For example, there are several ways in which a clinician can assert that he or she is the "treating clinician" for a particular

patient. We will call a set of rules with the same head a policy, since each such set completely specifies the ways in which a user can accomplish a particular goal.

Even the most complex policies specified in [2] contain less than a dozen rules, and thus can be satisfied in at most this many ways. We see from Figures 7, 8, and 9 that policies of this size can be efficiently analyzed by CLOUSEAU in under 100 ms in all cases. This shows that even when the myriad of requirements concerning patient privacy in the medical domain are considered, the number of unique ways that any given policy can be satisfied remains reasonably low. Thus, CLOUSEAU can efficiently handle the largest and most complex set of realistic policies assembled to date.

7 Conclusions and Future Work

In this paper, we described the design and implementation of the CLOUSEAU compliance checker. CLOUSEAU was designed to efficiently solve the type-3 compliance checker problem: given a set E of evidence and a policy p, determine all subsets of E that can be used to minimally satisfy p. Previous solutions for this problem have had running time overheads that are exponential in the size of the union of all satisfying sets. CLOUSEAU's time overheads scale linearly in the size of the union of all satisfying sets for the tests conducted in [21], and as $O(NA)$ in general, where N is the number of satisfying sets for a policy and A is the average size of each satisfying set. On average, CLOUSEAU requires only tens of milliseconds to find all satisfying sets of credentials for a given policy.

CLOUSEAU achieves this level of performance by taking a non-traditional approach to theorem proving. Rather than directly analyzing access control policies written in high-level languages (such as Cassandra, *RT*, TPL, or $\mathcal{X}$-TNL), CLOUSEAU compiles high-level policies into an intermediate representation that specifies constraints on the actual credentials and other evidence that must be presented to gain access to a particular resource. CLOUSEAU then leverages Rete, an efficient pattern matching algorithm, to enumerate all satisfying sets. In this paper, we provided a process through which access control policies specified in the *RT* language can be automatically compiled into the native rule format analyzed by CLOUSEAU, and proved its completeness and correctness. Since all trust negotiation policies are eventually satisfied by these same types of evidence, we conjecture that equivalent compilation procedures could also be derived for other higher-level policy languages. This allows policy writers to express their policies concisely using high-level policy languages, yet still analyze them efficiently.

Although CLOUSEAU is much more efficient than previous solutions to the type-3 compliance checker problem, further optimization is still an important area of future work. Spending tens of milliseconds during an interaction with their compliance checker is perfectly reasonable for entities in a peer-to-peer environment or clients in a client/server setting. However, servers that must process high volumes of traffic may need several such interactions for each trust negotiation session. An interesting direction of future work is to investigate high-level policy language constructs that can be compiled into CLOUSEAU policies that can be analyzed in a particularly efficient (or inefficient) manner. Better understanding the language constructs that most directly affect compliance checker performance could help lead to the design of yet more efficient compliance checkers.

Acknowledgments

This research was supported by the NSF under grants IIS-0331707, CNS-0325951, and CNS-0524695 and by Sandia National Laboratories under grant number DOE SNL 541065.

References

[1] L. Bauer, S. Garriss, and M. K. Reiter. Distributed proving in access-control systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 81–95, May 2005.

[2] M. Y. Becker. A formal security policy for an NHS electronic health record service. Technical Report UCAM-CL-TR-628, University of Cambridge Computer Laboratory, Mar. 2005.

[3] M. Y. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 159–168, June 2004.

[4] E. Bertino, E. Ferrari, and A. C. Squicciarini. $\mathcal{X}$-TNL: An XML-based language for trust negotiations. In *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 81–84, June 2003.

[5] E. Bertino, E. Ferrari, and A. C. Squicciarini. Trust-$\mathcal{X}$: A peer-to-peer framework for trust establishment. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):827–842, July 2004.

[6] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote trust management system version 2. IETF RFC 2704, Sept. 1999.

[7] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[8] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust management system. In *Proceedings of the Second International Conference on Financial Cryptography*, number 1465 in Lecture Notes in Computer Science, pages 254–274. Springer, Feb. 1998.

[9] P. Bonatti and P. Samarati. Regulating service access and information release on the web. In *Proceedings of the Seventh ACM Conference on Computer and Communications Security (CCS)*, pages 134–143, Nov. 2000.

[10] K. Borders, X. Zhao, and A. Prakash. CPOL: High-performance policy evaluation. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pages 147–157, Nov. 2005.

[11] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for web applications. *World Wide Web Journal*, 2(3):127–139, Summer 1997.

[12] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proceedings of the Second IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*, number 1995 in Lecture Notes in Computer Science, pages 18–39. Springer, Jan. 2001.

[13] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 27(3):219–227, 1985.

[14] E. Friedman-Hill. Jess: The rule engine for the Java platform. Web site, Apr. 2007. ⟨`http://www.jessrules.com`⟩.

[15] A. Herzberg, Y. Mass, J. Michaeli, D. Naor, and Y. Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–14, May 2000.

[16] H. Koshutanski and F. Massacci. An interactive trust management and negotiation scheme. In *Proceedings of the Second International Workshop on Formal Aspects in Security and Trust (FAST)*, pages 139–152, Aug. 2004.

[17] J. Li, N. Li, and W. H. Winsborough. Automated trust negotiation using cryptographic credentials. In *Proceedings of 12th ACM Conference on Computer and Communications Security (CCS)*, pages 46–57, Nov. 2005.

[18] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130, May 2002.

[19] D. P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 42–47, Aug. 1987.

[20] K. E. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation. In *Proceedings of the Third IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 68–79, June 2002.

[21] B. Smith, K. E. Seamons, and M. D. Jones. Responding to policies at runtime in TrustBuilder. In *Proceedings of the Fifth IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 149–158, June 2004.

[22] W. H. Winsborough and N. Li. Towards practical automated trust negotiation. In *Proceedings of the Third IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 92–103, June 2002.

[23] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Automated trust negotiation. In *Proceedings of the DARPA Information Survivability*

Conference and Exposition, pages 88–102, Jan. 2000.

[24] M. Winslett, T. Yu, K. E. Seamons, A. Hess, J. Jacobson, R. Jarvis, B. Smith, and L. Yu. Negotiating trust on the web. *IEEE Internet Computing*, 6(6):30–37, Nov./Dec. 2002.

[25] M. Winslett, C. Zhang, and P. A. Bonatti. PeerAccess: A logic for distributed authorization. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005)*, pages 168–179, Nov. 2005.

[26] D. Yao, K. Frikken, M. Atallah, and R. Tamassia. Point-based trust: Define how much privacy is worth. In *Proceedings of the Eighth International Conference on Information and Communications Security (ICICS '06)*, number 4307 in Lecture Notes in Computer Science, pages 190–209. Springer, 2006.

[27] T. Yu, M. Winslett, and K. E. Seamons. Supporting structured credentials and sensitive policies throu gh interoperable strategies for automated trust negotiation. *ACM Transactions on Information and System Security*, 6(1), Feb. 2003.

A Proof of Theorem 1

A role membership proven using the *RT* rules of inference can be represented as a proof tree in which the root node represents the target role (i.e., $K_A.R$), intermediate nodes represent memberships in intermediate roles, and all leaves of the tree represent identity certificates. All non-leaf nodes of the proof tree are labeled with the identity of the principal whose membership in that particular role has been proven. Figure 10 is an example proof tree proving that Alice is a member of the *Provider.service* role as discussed in Section 5.1. Note that the linking containment rule *Provider.service* ← *Provider.partner.employee* causes the proof tree to branch: the left branch proves that AliceLabs is a a member of the *Provider.partner* role and the right branch proves that Alice is an employee of AliceLabs. Given this representation of a role membership verified using the *RT* rules of inference, we now prove the following.

Lemma 1. *Let $R = \{r_1, \ldots, r_n\}$ be a set of role definition credentials, $C = \{c_1, \ldots, c_m\}$ be a set of identity certificates, $p = R \cup C$ be an RT_0 policy, and let p' be the result of compiling p using the process described in Section 5.2. If a proof tree with root node $K_C : K_A.R$ can be found for p using the RT rules of inference on a set $S \subseteq (R \cup C)$ of credentials, then* Clouseau *finds the satisfying set S of credentials proving membership in $K_A.R$ after analyzing the compiled policy p'.*

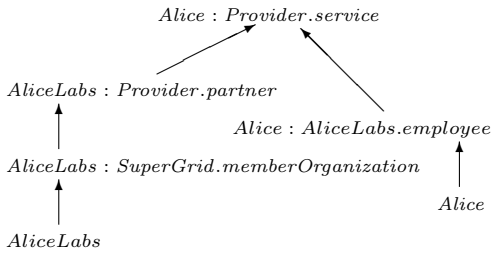

Figure 10: An example RT_0 proof tree.

Proof. We proceed by induction on the depth of the proof tree found using the *RT* rules of inference. The base case occurs when the proof tree extends one level beyond the root node. In this case, the proof tree is the result of a simple membership role definition $K_A.R \leftarrow K_C$. The root node of this proof tree is labeled $K_C : K_A.R$ and the leaf node of the tree signifies that K_C's identity certificate was obtained. The Clouseau compliance checker would determine that the policy p' generated by compiling this base policy was satisfied, as follows. The simple membership credential containing the assertion $K_A.R \leftarrow K_C$, K_A's identity certificate, and K_C's identity certificate would match the LHS of the `member-of` rule inserted during step 1 of the policy compilation process (see Figure 5). This rule would assert an `is-member` object defining K_C to be a member of $K_A.R$ and would store references to the simple membership credential, as well as the identity certificates of K_C and K_A.

Assume that Clouseau can determine membership in any role using the same set of credentials as *RT* in all cases where the depth of the proof tree found using the *RT* rules of inference is at most n. To prove that Clouseau can determine role membership using the same set of credentials as *RT* where the proof tree found using the *RT* rules of inference is of depth $n + 1$, we must consider three cases.

Case 1 (Simple Containment): In this case, the *RT* inference linking the first and second levels of the proof tree will occur as a result of processing some simple containment credential $K_A.R \leftarrow K_B.R_1$.

The root of the resulting proof tree has one child node whose label is $K_C : K_B.R_1$. The subtree of the proof rooted at this child node is of depth n and thus CLOUSEAU finds the same set of credentials proving K_C's membership in $K_B.R_1$ (by the inductive hypothesis) and will assert an `is-member` object containing this information. This will match the LHS of the `rule-sc-<id>` rule inserted after preprocessing the $K_A.R \leftarrow K_B.R_1$ simple containment credential during step 3 of the compilation process, which will then insert an `is-member` object defining K_C to be a member of $K_A.R$.

Case 2 (Linking Containment): In this case, the RT inference linking the first and second levels of the proof tree will occur as a result of processing a linking containment credential $K_A \leftarrow K_A.R_1.R_2$. The root of the resulting proof tree will have two child nodes: one labeled $K_B : K_A.R_1$ asserting that some principal K_B is a member of the role $K_A.R_1$, and one labeled $K_C : K_B.R_2$ which asserts that K_C is a member of $K_B.R_2$. Because the subproofs rooted at these two nodes each have a depth of at most n, CLOUSEAU will have asserted `is-member` objects describing these memberships using the same set of credentials by the inductive hypothesis. These two objects will then match the LHS of the `rule-lc-<id>` rule inserted after preprocessing the $K_A \leftarrow K_A.R_1.R_2$ linking containment credential during step 3 of the compilation process, which will assert K_C's membership in $K_A.R$.

Case 3 (Intersection Containment): In this case, the RT inference linking the first and second levels of the proof tree will occur as a result of processing an intersection containment credential $K_A.R \leftarrow K_{B_1}.R_1 \cap \cdots \cap K_{B_m}.R_m$. The root of the resulting proof tree will have m child nodes, each labeled to assert K_C's membership in some role $K_{B_i}.R_i$. Because the subproofs rooted at these m nodes are of depth at most n, CLOUSEAU can assert `is-member` objects providing evidence of K_C's membership in these roles using the same sets of credentials by the inductive hypothesis. These m objects will then match the LHS of the `rule-ic-<id>` rule inserted after preprocessing the $K_A.R \leftarrow K_{B_1}.R_1 \cap \cdots \cap K_{B_m}.R_m$ intersection containment credential during step 3 of the compilation process, which will assert K_C's membership in $K_A.R$.

Since CLOUSEAU can discover the same satisfying set of credentials as RT in each of these three cases, it can do so for any policy p' resulting from the compilation of an RT_0 policy p. For RT to grant access based on this proof, however, K_C must demonstrate proof of ownership of his or her identity certificate. CLOUSEAU enforces this same constraint by means of the `target` rule, which also requires a demonstration of proof-of-ownership for K_C's identity certificate. □

Lemma 2. *Let $R = \{r_1, \ldots, r_n\}$ be a set of role definition credentials, $C = \{c_1, \ldots, c_m\}$ be a set of identity certificates, $p = R \cup C$ be an RT_0 policy, and let p' be the result of compiling p using the process described in Section 5.2. If* CLOUSEAU *asserts a* `satisfaction` *object containing the set $S \subseteq (R \cup C)$ of credentials that proves K_C's membership in the target role $K_A.R$, then this membership can also be proven using the RT rules of inference on S.*

Proof. We say that a set of CLOUSEAU rules $l_1, l_2, \ldots, l_j$ form a *chain* if an assertion made by the RHS of l_1 matches a pattern on the LHS of l_2, an assertion made by the RHS of l_2 matches a pattern on the LHS of l_3, and so on. Our proof then proceeds by induction on the length of the longest chain of rules invoked by CLOUSEAU prior to invoking the `target` rule. The base case occurs when one rule is invoked. This can only occur when the `member-of` rule is matched by a simple membership credential $K_A.R \leftarrow K_C$, the identity certificates of K_A and K_C. This will assert an `is-member` object attesting to K_C's membership in $K_A.R$. In this case, we can use the RT rules of inference to determine that K_C is a member of $K_A.R$, as we have both the simple membership credential $K_A.R \leftarrow K_B$ and K_C's identity certificate.

Now, assume that the RT rules of inference can be used on the policy p to determine membership in $K_A.R$ as long as the length of the longest chain of rules invoked by CLOUSEAU when analyzing the policy p' is less than or equal to n. To prove that the RT rules of inference can be used to determine membership in $K_A.R$ if the longest chain of rules invoked by CLOUSEAU is $n+1$, we must examine three cases.

Case 1 (`rule-sc-<id>`): Consider the case in which the last rule in a chain to be invoked by CLOUSEAU is an instance of the `rule-sc-<id>` inserted by CLOUSEAU upon examining a simple containment credential $K_A.R \leftarrow K_B.R_1$. The LHS of this rule would be matched by a single `is-member` object asserting K_C's membership in the role $K_B.R_1$. Since the longest chain of CLOUSEAU rules needed to assert this object is at most $n-1$, the RT rules of inference could also be used to prove membership in $K_B.R_1$ using the same set of credentials by the

inductive hypothesis. This membership proof can be combined with the simple containment credential $K_A.R \leftarrow K_B.R_1$ to prove K_C's membership in $K_A.R$ using the RT rules of inference.

Case 2 (`rule-lc-<id>`): Consider the case in which the last rule in a chain to be invoked by CLOUSEAU is an instance of the `rule-lc-<id>` inserted by CLOUSEAU upon examining a linking containment credential $K_A.R \leftarrow K_A.R_1.R_2$. The LHS of this rule would be matched by two `is-member` objects: one proving that some K_B is a member of $K_A.R_1$ and another proving that K_C is a member of $K_B.R_2$. The longest chain of CLOUSEAU rules needed to assert either of these objects is at most $n-1$ and thus the RT rules of inference could also be used to prove $K_A.R_1 \leftarrow K_B$ and $K_B.R_2 \leftarrow K_C$ using the same set of credentials. These membership proofs can be combined with the linking containment credential $K_A.R \leftarrow K_A.R_1.R_2$ to prove K_C's membership in $K_A.R$ using the RT rules of inference.

Case 3 (`rule-ic-<id>`): Consider the case in which the last rule in a chain to be invoked by CLOUSEAU is an instance of the `rule-ic-<id>` inserted by CLOUSEAU upon examining an intersection containment credential $K_A.R \leftarrow K_{B_1}.R_1 \cap \cdots \cap K_{B_m}.R_m$. The LHS of this rule would be matched by m `is-member` objects each proving that K_C is a member of some $K_{B_i}.R_i$. The longest chain of CLOUSEAU rules needed to assert any of these objects is at most $n-1$ and thus the RT rules of inference could also be used to prove $K_{B_i}.R_i \leftarrow K_C$ for $1 \leq i \leq m$ using the same set of credentials. These memberships can be combined with the intersection containment credential $K_A.R \leftarrow K_{B_1}.R_1 \cap \cdots \cap K_{B_m}.R_m$ to prove K_C's membership in $K_A.R$ using the RT rules of inference.

Since the RT rules of inference can be used to find an equivalent proof of membership in the role $K_A.R$ for each of these three cases, we can conclude that the RT rules of inference can be used on the policy p to determine an equivalent proof of membership for the principal K_C in the role $K_A.R$ any time that CLOUSEAU asserts an `is-member` object when analyzing the policy p'. For CLOUSEAU to grant access based on this role membership (i.e., assert a `satisfaction` object), proof-of-ownership of K_C's identity certificate is required by the `target` rule. RT also requires this proof-of-ownership prior to granting access, as the label of the root node of the proof tree is K_C. □

Theorem 1 follows directly from Lemmas 1 and 2.