Security Policy Languages and Enforcement*

Scott D. Stoller† Yanhong A. Liu‡

October 26, 2004

1 Introduction

As organizations grow larger and more complex, and as cybersecurity becomes an increasingly important concern, there are growing needs for languages that can express complex security policies of organizations and for efficient mechanisms to enforce the policies. An essential function of security policies is to control *authorization*, that is, to determine whether a request to access a resource should be permitted or denied. This paper considers two key aspects of security policy languages—support for decentralized policy administration through "trust management", and support for scalable policy management through roles—and techniques for the efficient implementation of these languages. The implementation techniques provide a basis for enforcing security policies expressed in these languages.

Traditional ways of expressing authorization policies, such as access control lists, were developed to support authorization in centralized systems with a single administrator in control of all aspects of security policy for the entire system. They are generally adequate and widely used in that context, but they have serious deficiencies for enterprise-wide applications [2], which may have many administrative domains and varying trust relationships.

Trust management systems are designed to support authorization in distributed systems [2]. The defining characteristic of trust management systems is support for decentralized security policy administration through *delegation*: an entity can authorize another entity to control specified aspects of security policy. For example, a company's chief executive officer (CEO) might allow each manager to control his subordinates' access to technical data, while the CEO retains complete control over everyone's access to the company's strategic plan.

Role-based policy languages support scalable specification and management of policies in large systems [11, 4]. A *role* is an abstraction that represents a set of permissions, typically the permissions needed to perform the tasks associated with a position in an organization. Role-based authorization policies specify the roles that each user may adopt, and the permissions associated with each role.

*This work was supported in part by NSF under Grants CCR-0306399 and CCR-0205376 and ONR under Grants N00014-02-1-0363 and N00014-04-1-0722.

†Position: Associate Professor. Address: Computer Science Dept.; Stony Brook University; Stony Brook, NY 11794-4400; USA. Phone: 631-632-1627. Email: stoller@cs.sunysb.edu.

‡Position: Associate Professor. Address: Computer Science Dept.; Stony Brook University; Stony Brook, NY 11794-4400; USA. Phone: 631-632-8463. Email: liu@cs.sunysb.edu.

2 Trust Management Policy languages

Datalog (Database Logic) [5], a classic rule-based query language for databases, is an attractive foundation for policy languages in decentralized systems. Datalog allows recursive definitions of relations, so it can express queries not expressible in relational algebra or relational calculus, but it does not allow construction of recursive data structures, so it consumes bounded resources. Trust management systems based on Datalog or extensions of it include Delegation Logic [7], SD3 (Secure Dynamically Distributed Datalog) [6], Binder [3], and RT (Role-based Trust-management) [8]. Datalog is an attractive basis for trust management languages for several reasons: (1) It is declarative and has a simple and well-studied semantics. Datalog statements can easily be translated into declarative English sentences. This helps ensure that users will be able to formulate security policies that accurately capture their intentions. (2) It allows authorization based on all properties of entities and requests, not only their identities. For example, this enables the above policy involving the CEO, manager, and subordinates to be stated directly and clearly in terms of attributes (e.g., is-a-manager) and relations (e.g., is-the-manager-of), instead of by enumerating the permissions of each person individually. (3) Application-specific relations can be defined, and recursive definitions are allowed. Recursive definitions arise naturally when hierarchical structure (e.g., of a computer network, or of an organization) is involved. (4) Queries are decidable in polynomial time. This includes *compliance checking*, which determines whether a given request is permitted by a given policy, and policy analysis problems, such as computing the *meaning* of a given policy (i.e., the set of all operations permitted by the policy).

A significant obstacle to deployment of trust management systems with Datalog-based policy languages is the lack of suitable implementations of such languages. Simple Datalog interpreters are easy to implement but have poor performance, especially for programs containing recursive definitions. Development of an optimized implementation is a significant undertaking, and the result is a heavy-weight system, not easily deployed for trust management [8]. Worse yet, even in the best existing highly-optimized logic programming systems, such as GNU Prolog and XSB (http://xsb.sourceforge.net/), the running time of a program can vary dramatically depending on the order of rules in the program or the order of hypotheses within each rule. It is very difficult for users to determine which order will lead to more efficient execution, because the answer depends on implementation details.

We are addressing these implementation challenges through development of a powerful method, described in [9], for (1) automatic transformation of specifications in Datalog-like languages into specialized algorithms and lightweight implementations, and (2) automated complexity analysis that provides time and space guarantees for the generated algorithms.

The transformation is based on a general method for efficient fixed-point computation. The set of all facts derivable from a Datalog program corresponds to a fixed-point computation that repeatedly derives new facts from

existing facts by using the rules, until no more new facts can be derived. Our transformation exploits three key ideas to compute the fixed point efficiently: (1) perform a minimum update in each iteration, i.e., add one new fact at a time; (2) maintain appropriate auxiliary maps (i.e., indices), based on the structure of the rules, and update them incrementally in each iteration; and (3) use appropriate combinations of indexed and linked data structures.

The generated implementations have guaranteed optimal time complexity and associated space complexity, in the sense that only useful combinations of information that lead to the invocation of a policy rule are considered, and each such combination is considered exactly once. The method computes the worst-case time and space complexities, as formulas. These are independent of the order of rules and hypotheses, in contrast to previous methods.

We are extending the method to handle query-driven (goal-directed) computation, incremental computation with respect to policy changes, and distributed evaluation.

3 Role-Based Policy Languages

Role-based policy languages offer well-established advantages for policy management in large systems. They are widely used in role-based access control (RBAC) [11]. The roots of role-based access control include the notion of groups in UNIX and other operating systems. The growing importance of RBAC motivated the development and recent approval of an ANSI standard for it [4, 1].

The standard specifies sets of users, objects, operations, roles, and sessions, and over a dozen relationships built on top of these sets. The standard then specifies several dozen operations on them.

Producing straightforward implementation of the standard in a high-level language with good support for sets, such as the popular object-oriented scripting language Python (http://www.python.org/), is relatively easy, but the straightforward implementation will have poor performance.

Manual development of a high-performance implementation takes significantly longer and requires writing more complicated and less modular code, which consequently is also more difficult to maintain. To see why, note that it is often much more efficient to incrementally maintain a set (or a relation, which we regard as a set of tuples) than to compute it from scratch each time it is needed. In a straightforward implementation, a set used in one operation may be calculated from scratch in one place, in the code for that operation. To produce an efficient implementation, we must identify all operations that perform updates that may affect that set, and in their implementations insert code to incrementally maintain that set. This breaks the modularity (abstraction) of the straightforward design and implementation. More generally, this problem arises with all expensive computed quantities (not only sets), which we refer to as *queries* of the data from which they are computed.

We are addressing this implementation challenge through development of a general and systematic method that supports the use of abstractions by al-

lowing each component and operation to be specified in a clear and modular fashion and implemented straightforwardly in an object-oriented language [10]. The method then analyzes queries and updates in the straightforward implementation, cutting across the abstractions in the clear and modular specification. Finally, the method breaks through the abstractions and transforms the straightforward implementation into a sophisticated and efficient implementation that incrementally maintains the results of expensive queries with respect to all relevant updates. The main steps are to determine which queries should be incrementally maintained, which updates may affect each query, and, most challengingly, where to store and how to update each incrementally maintained value, based on a cost model. The transformations are expressed declaratively as *incrementalization rules*.

The method can be used automatically, semi-automatically, or manually. We developed conservative analysis and transformations that can be fully automated. A semi-automatic version may use more aggressive analysis and transformations that rely on hints from the user. The method can be followed manually as a design methodology.

We developed a prototype implementation of the method for Python. We applied the method in the development of efficient object-oriented programs in a number of example applications [10]. Here we discuss the application to RBAC.

We created a straightforward implementation of the RBAC specification in Python, as a complete and executable specification. Overall, more than half of the operations involve expensive queries, i.e., queries that take more than constant time.

We then applied our analyses and transformations to our straightforward implementation of core RBAC (core RBAC contains the majority of the sets, relations, and operations in the RBAC standard), to automatically incrementalize the expensive queries with respect to the updates. There are over a dozen expensive queries and over a dozen updates. Our method is able to optimize all expensive queries to take constant time except for a trade-off that lets either `CreateSession` take constant time and `CheckAccess` (and `SessionPermissions`) take $O(|\texttt{Permissions}|)$ time, or *vice versa*. In typical applications, `CheckAccess` is performed much more frequently than `CreateSession`, and thus the latter gives better performance, as confirmed in our experiments. For example, with 900 permissions, the average running time of `CheckAccess`, for the straightforward, the former incrementalized, and the latter incrementalized implementations are 0.342, 0.0345, and 0.000187 seconds, respectively, when measured on a dual-CPU 2.3 GHz Athlon XP computer with Python 2.3.

Future research is needed on improved analysis of costs and frequencies of and dependencies between queries and updates, suitable languages for specifying incrementalization rules, and further optimizations for on-demand and concurrent computations.

We are also investigating the design and implementation of role-based trust management languages.

References

[1] American National Standards Institute, Inc. Role-based access control. ANSI INCITS 359-2004. http://csrc.nist.gov/rbac/.

[2] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer-Verlag, 1999.

[3] John DeTreville. Binder, a logic-based security language. In *Proc. 2002 IEEE Symposium on Security and Privacy*, pages 105–113. IEEE Computer Society Press, 2002.

[4] David F. Ferraiolo, Ravi sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. PROPOSED nist standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[5] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice-Hall, 2002.

[6] Trevor Jim. SD3: A trust management system with certified evaluation. In *Proc. 2001 IEEE Symposium on Security and Privacy*, pages 106–115. IEEE Computer Society Press, 2001.

[7] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security*, 2003.

[8] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proc. 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, 2002.

[9] Yanhong A. Liu and Scott D. Stoller. From Datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 172–183, Uppsala, Sweden, August 2003.

[10] Yanhong A. Liu, Scott D. Stoller, Yanni Ellen Liu, Michael Gorbovitski, and Tom Rothamel. Design by building up and breaking through abstractions. Technical Report DAR-04-15, SUNY at Stony Brook, Computer Science Dept., September 2004.

[11] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.