Adjustable Autonomy for Cross-Domain Entitlement Decisions

Jacob Beal
Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA, USA 02138
jakebeal@bbn.com

Jonathan Webb
Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA, USA 02138
jwebb@bbn.com

Michael Atighetchi
Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA, USA 02138
matighet@bbn.com

ABSTRACT

Cross-domain information exchange is a growing problem, as business and governmental organizations increasingly need to integrate their information systems with those of partially trusted partners. Current identity management and access control technologies operate only within a specific domain and are unable to scale to the asymmetric, heterogeneously administered, and highly restrictive security policies of cross-domain environments. We approach the problem as one of *adjustable autonomy*, in which the human administrator needs to encode policy intent in a way that allows routine decisions about policy interactions to be safely delegated to the machine. In this paper, we present work toward such a system, combining a lattice representation of access control decisions and client attributes with search through a space of cross-domain mapping relations. This combination enables a policy resolution algorithm that resolves routine policy interactions while flagging potential conflicts for attention from a human administrator.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Inference engines*

General Terms

Security, Theory, Algorithms

Keywords

Cross-Domain, Policy Description, Domain-Specific Language

1. INTRODUCTION

In today's increasingly networked world, *cross-domain* information exchange, in which an organization's information system is integrated with the information systems of partially trusted partners, is a problem of rapidly growing importance. Information must be able to flow freely, but only in accordance to the policies of the organizations involved—both their own internal policies and the exchange policies that have been mutually agreed upon. For example:

- A business may need to integrate its systems with partners that it has long-term contracts with, yet avoid exposing sensitive information to competitors that contract with the same partners.
- Ethically and legally protected data, such as personal medical records and financial information, needs to be able to flow through a network of service providers while being protected.
- Government agencies need to be able to safely share information with other agencies and with coalition partners.

Current identity management and access control technologies perform identity validation and enforcement of access rights only within a specific domain and are unable to operate within the asymmetric and highly restrictive security policies of cross-domain environments. In addition to the basic challenges of heterogeneity, multi-party maintenance, and data-sharing constraints, there is a fundamental problem of safety and maintenance. Simply put: the potential interactions of a set of cross-domain policies quickly grow beyond the ability of humans to reliably determine whether the composed policy-set erroneously allows prohibited information flows.

One view in approaching such a problem, which we have adopted, is of adjustable autonomy: the human and machine are viewed as a cooperative team, with the human in ultimate control but delegating authority for routine decisions to the machine. The challenge before us, then, is to create a policy representation that allows a human administrator to succinctly control policy interactions where necessary, to delegate most routine decisions about policy interactions to machine reasoning, and to tractably prove safety properties for composed policy sets.

In this paper, we present foundational work toward such a system of policy representation. We begin by representing access control decisions as a lattice, with the *meet* relation used for autonomous decision combination. The set of applicable policies can then be determined by a search through cross-domain mapping relations, with a lattice representation of client attributes allowing implicit precedence relations between policies to be inferred. Together the lattices

AISec'10, October 8, 2010, Chicago, Illinois, USA.

Identity	The essence of an entity. One's identity is often described by one's characteristics, among which may be any number of identifiers.
Identifier	A data object (for example, a string) mapped to a system entity that uniquely refers to the system entity.
Attribute	A distinct characteristic of an object.
Credential	Data that is transferred to establish a claimed principal identity.
Policy	A set of rules and practices that specify or regulate how a system or organization provides security services to protect resources.
Decision	The result of evaluating a policy.

Table 1: Definitions of security terms used in this paper, taken from [1] and [2].

and relation search enable a policy resolution algorithm that may be applied statically or dynamically to autonomously resolve routine policy interactions while flagging potential conflicts for attention from a human administrator.

1.1 Related Work

A significant amount of work exists in the areas of privilege and identity management, both within and across domains. We divide the space into enforcement substrates, privilege management frameworks, policy analysis solutions, and security models.

Enforcement substrates:.

Enforcement substrates: XACML[2] and SELinux[3, 4] are two widely used technologies for enforcing access control policies. XACML is part of the OASIS standards stack and defines both architectural components, e.g., policy decision points (PDPs) and policy enforcement points (PEPs), as well as an XML-based policy language for expressing access control rules. XACML is widely used in service-oriented architecture based systems to restrict access to critical services. SELinux is available for many standard Linux distributions today, and allows fine-grained restriction of functions processes are allowed to perform on operating system resources, such as opening files or sending packets over the network. SELinux is also used as a main component in military high assurance platforms (HAPs)[5]. Our work uses existing enforcement technologies, in particular XACML PDPs and PEPs, for making online access control decisions.

Privilege management frameworks:.

A number of access control and identity management frameworks exist that provide a federated capability. OpenID[6] and Shibboleth[7] provide single sign on capabilities across a federation domains, OpenPERMIS[8] provides distributed access control in grid environments with advanced support for delegation of authority and normalization between different policy representations. DISA is pursuing the notion of policy-based and risk adaptive policy management that combines information from the enterprise including policies, user attributes, resource meta-data, environmental attributes, and the action performed to provide access control capabilities to dynamic operating environments. These ongoing efforts are facing and addressing many of the same problems described in this paper, but lack formalisms that scale with the number of users, attributes, and resources, and can support information sharing requirements associated with handling of classified information in cross domain environments.

Policy analysis solutions:.

Various analysis methodologies have been used to analyze and deconflict access control policies. The Lobster[9] domain specific language and tool chain enables expression of high-level security policies through graphs, focusing on information flow between resources in a system. Lobster also provides a tool chain for automated assured refinement of those policies into enforceable SELinux policies, and can check policies for consistency through definition and evaluation of policy assertions. Lobster also provides support for automatically constructing high-level policies from manually crafted low-level policies. Lobster does not support obligations on flowing data (e.g. filters or logs) or precedence relationships by which policies can override one another, which is a key feature of our work. KAoS[10] is a policy management framework, based on semantic web technology, that has been used in multi-agent frameworks to uses ontological representations of policies and provides deconfliction analysis through use of the Stanford Java Theorem Prover[11]. For theorem proving, we are currently exploring the use of Alloy[12], a first-order logic modeling language frequently used for program analysis, on our models to find theorems.

Formal security models:.

A number of formal security models were developed in the 70's for cross domain interactions in military contexts. The Bell-La Padula [13] model formalizes interactions that implement a "read-down, write-up" secrecy policy. The BIBA model [14] formally describes constraints on access control rules that prevent violation of data integrity . Our work relates to these models in the sense that rigorous formalisms are used to describe valid and invalid interaction patterns. However, these models are becoming increasingly untenable because of the increasing need to seamlessly handle dynamically unfolding events and the associated need to dynamically change policies. Our approach aims at establishing proof that any policy modification and composition is integrity preserving by analyzing the resulting policies for unauthorized disclosure (leakage) or denial of permissible information access (blockage).

2. A MOTIVATING EXAMPLE

Consider the following example of policy interaction, which we will use to motivate discussion throughout the remainder of the paper: Bob is a purchaser working for a large online retailer "Bacchae.com," where he is typically logged into its System B. In the course of business, he frequently interacts with the manufacturer Acme Inc., one of Bacchae's suppliers, obtaining inventory and shipping data from Acme's System A.

Acme grants purchasers on B access to shipping data only regarding its contracts with Bacchae, since it sells to Bacchae's competitors as well. Acme grants access to inventory information to all of its logistics staff, and grants Bacchae logistics staff the same permission by treating them like Acme logistics staff. However, Acme's logistics staff are allowed unfiltered access to shipping data.

The pseudo-code for these policies (Figure 1) is:

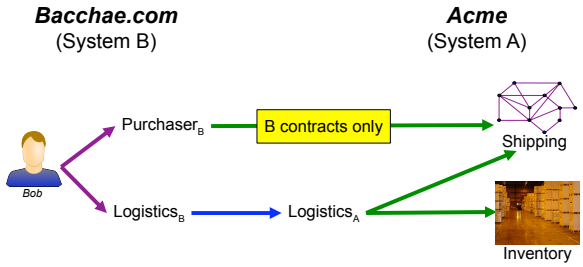


Figure 1: Policy and attributes for a simple case of privilege escalation through cross-domain policy combination: credential/attribute relations are purple arrows, attribute remapping is blue, access privileges are green arrows with modifications in yellow boxes.

1. $Purchaser_B$ reads $Shipping_A$ with filter: B-contracts only
2. $Logistics_A$ reads $Inventory_A$
3. $Logistics_B \rightarrow Logistics_A$
4. $Logistics_A$ reads $Shipping_A$

This is a situation that has the potential for privilege escalation, since Acme does not want System B users to have unfiltered access to shipping data, but the alternate path through $Logistics_A$ does not have a filter. This potential can be induced via any of three locally correct actions:

- Policy 3 is added to give System B users access to inventory data.
- Policy 4 is added to give System A users access to shipping data.
- The filter is added to Policy 1, weakening previously unfettered access.

Let us analyze how a situation of this type can be resolved, considering the four types of access information that can flow across boundaries: **Credentials**, **Attributes**, **Policies**, and **Decisions**. In the use case in Figure 1, System B originates all relevant **Credentials** and **Attributes**, while System A originates all relevant **Policies**.

In addition to **Policies** that declare privileges, we will consider two other types of user-provided policy information: **Precedence** relationships between policies, determining when one policy supersedes another and **Mappings** that translate **Credentials** and **Attributes** from one domain to another.

3. DECISION AUTOMATION LATTICES

At the end of the day, what is most important for any access control system is that an appropriate decision gets made, so we will start by organizing decisions and working our way back from there.

When multiple policies apply, they may specify different decisions about what access a client is entitled to for a resource. In some cases, there may be many variants on a basically similar decision, and in these cases we want the system to be able to autonomously combine the variants into a compound decision compatible with all of them. In other cases, the policies may specify fundamentally incompatible decisions, and in these cases we want the system to log an error for a human operator to debug.

Typical existing combination methods are *ad hoc* and make it impossible to reason about the intent of the human specifying the policy. For example, XACML[2] offers several standard combination methods, such as "first-applicable" or "deny-overrides." These fail to provide safety even in a simple example such as the one we have given: "first-applicable" gives unfettered access if Policy 4 happens to be listed before Policy 1, and "deny-overrides" silently combines the two paths without noticing the fundamental incompatibility that indicates a mistake has been made in policy specification.

We will use the algebraic concept of a *lattice* as a combination method that will better represent user intent. A lattice is a partial order on a set, where the least upper bound ("join") and greatest lower bound ("meet") of any subset contains precisely one element. Any finite lattice thus includes two special elements, *top* and *bottom*, which are the greatest and least elements of the set respectively. By representing decisions as a lattice and interpreting the bottom element as an error, we can automate combination of decisions: any set of decisions whose *meet* is above *bottom* can be safely combined to that *meet*, while any set of decisions with a *meet* of *bottom* is a problem that requires human intervention.

There are three basic **Decisions** in any security system: **Permit**, **Deny**, and **Conflict**, where **Conflict** is a report of an error of some sort (and will usually be implemented as a denial plus some sort of report to the system administrator).

A **Decision** of **Permit**, however, may be modified: "Yes, but..." (in XACML, this is implemented via "obligations"). For example, a redaction filter may be added that allows only client-specific information to cross a domain boundary, or an entry may be added to a log file recording the decision.

We will systematize such modifications into two allowable categories:

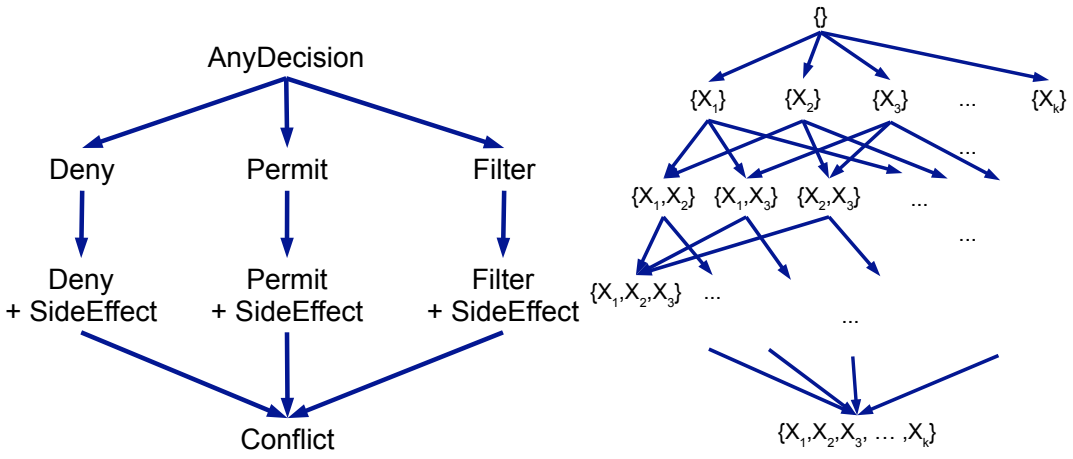
- **Filter** is an operator that modifies the content of the service interaction regulated by the **Decision**.[1]
- **SideEffect** is an operator that takes an action that does not modify policy, does not modify the behavior of either this or future service interactions, and does not send information across domain boundaries.

Figure 2 shows the type-lattice of order relationships between the three basic decisions, these two categories, and a top element **AnyDecision**, of which all decisions are subtypes.

These categories do not cover all of the possible modifications that one could make to a decision. For example, a rate-limiting policy might use a side-effect to count how many times a given client has used a service. However, such modifications appear harder to prove safety properties about so we will begin by excluding them from consideration, reserving the possibility of later expanding the set of allowable categories.

We further require that the set of **Filter** operators for a system must be commutative, associative, and idempotent, as must be the set of **SideEffect** operators. Commutativity and associativity mean that the same effects will happen, no matter what order the operators are arranged in.

[1]Note that some filters may be unsafe or impractical to prove safety properties about; we are assuming that filters and the safety of their combination is vetted elsewhere.



(a) Decision Lattice (b) Power Set Lattice

Figure 2: Type lattice (a) giving ordering relationships between Decisions, where Filter and SideEffect are deployment-specific sub-lattices. The base form of these sub-lattices is subset ordering over the power set of Filter or SideEffect operators (b). Important to note is that there is no ordering between Deny, Permit, and Filter, such that their GCS is always Conflict, but that side-effects can be added or filters combined without causing conflicts.

Idempotent means that two copies of an operator can be consolidated into one without changing its effect.[2] These sorts of compositional properties are not generally considered in current policy design, but many of the obvious classes of **Filter** and **SideEffect** operators can be constructed to satisfy them. For example, is it easy to prove that most information-removing filters are commutative, associative, and idempotent.

With these restrictions, we are guaranteed that the set of **Filter** operators and the set of **SideEffect** operators can each be arranged into a lattice. Given a set of associative, commutative, and idempotent operators, it is always possible to establish a lattice over their power set, taking ordering from subset relations. Adding ordering relationships between operators can compact the lattice, making it more efficient. For example, time delay filters of variable length can be ordered such that longer delays supersede shorter delays. Thus, given any two time delay filters, the one with the shorter delay can be discarded, eliminating all lattice elements containing more than one time delay filter.

It is thus possible to take any set of decisions, including arbitrary side-effects and filters, and determine whether the set is consistent by finding the *greatest common subtype* (GCS, a.k.a *meet*, *greatest lower bound*, or *infimum*). An inconsistent set has a GCS of **Conflict**, an indeterminate set has a GCS of **AnyDecision**, and a consistent set (no matter how large) resolves to a GCS or a single **Decision** that consolidates all of the behaviors implied by the policy into their simplest possible implementation.

The lattice in Figure 2 thus plays a critical role in system design, as it determines when the system will view differing policies as a problem to report, as opposed to when it will view them as an opportunity for automatic resolution to be useful. Both spurious problem reports and incorrect automatic resolution of problems are not acceptable, so we must strike a balance that attempts to minimize these misbehaviors, and given the concerns of security we are biased to prefer spurious reports over incorrect automatic decisions.

Since side-effects do not change the cross-domain information flow, it is safe to set it so that side-effects will never create conflicts (thus their position as subtypes of all other decisions).

Intuitively, it seems reasonable that combining **Permit** and **Deny** should result in a conflict, so they should not be ordered with respect to one another. This may be reinforced by considering example policies in which either should override:

- If attributes are normal, then **Deny**, but when given special permission the decision should instead be **Permit**, e.g. white-listing, common firewall rules.
- If attributes are normal, then **Permit**, but when a security incident flag is set the decision should instead be **Deny**, e.g. black-listing, certificate revocation.

Given that the override can go in either direction, it must be established explicitly in the policy design, rather than being implicit via an ordering of **Permit** and **Deny** in the lattice of **Decisions**.

This leaves filters, which could be argued to be related either to **Permit**, which can be viewed as a null filter, or to **Deny**, which can be viewed as a total filter. For now, however, we choose to leave them unordered (so that combining **Filter** with either **Permit** or **Deny** is a conflict). We do not order **Filter** and **Deny** since we believe that security policies generally make a qualitative distinction between no access and any trickle of access no matter how small. We do not order **Filter** and **Permit** because we believe it likely that cross-domain security policies will often want to carefully separate unfettered in-domain access and filtered cross-domain access.

[2]Note that for this to hold for logging operations, we must define the effect to be "Entry X is contained in the log file", or else the (unimportant) ordering and/or duplication of log entries fails commutativity and idempotence. Redaction filters must be constructed in a similar manner where the effect is "Entry X is not contained in the document."

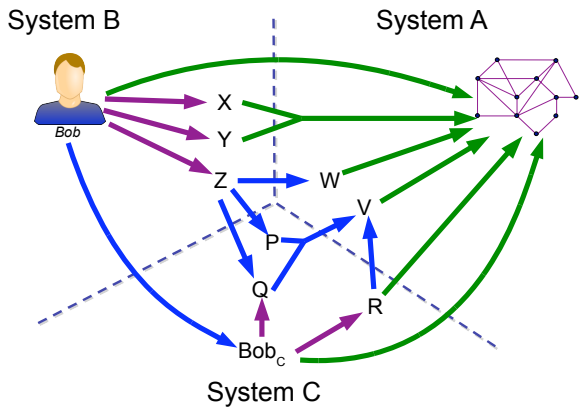
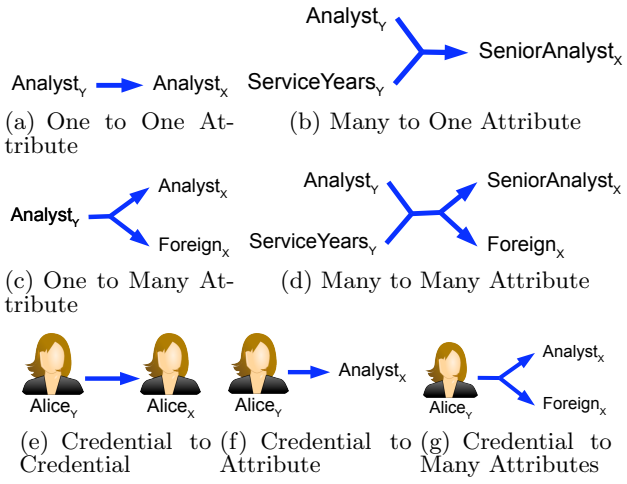


Figure 3: Even when only a small number of domains are involved, there is potentially a significant search task required in order to determine all applicable policies for a service request. In the example scenario above, with Bob on System B requesting access to shipping data on System A, there are 8 different applicable Policy combinations, each involving a different cross-domain path. Attributes are shown as letters, Credential/Attribute relations are purple arrows, cross-domain Mappings for Credentials and Attributes are blue arrows, and access Policies are green arrows.



(a) One to One Attribute (b) Many to One Attribute

(c) One to Many Attribute (d) Many to Many Attribute

(e) Credential to Credential (f) Credential to Attribute (g) Credential to Many Attributes

Figure 4: Examples of the seven possible cases of cross-domain mapping of Attributes and Credentials.

4. DISCOVERING POTENTIAL POLICY CONFLICTS

If only one access policy applies to a service request, then the decision is clear. It is often the case, however, that multiple policies may apply (for this work, we will assume that the resolving party has access to all policy information; the conditions under which it is even possible to detect conflicts with incomplete information are not currently known). Sometimes this is intentional, as when a policy specifies an exception that overrides a more general policy. Sometimes this is a mistake, as in our example when two separately conceived policies interact to create an unintended privilege escalation.

We would like to detect all instances of the latter case, while resolving the former case with as little explicit instruction from the human security administrator as possible. The reason is one of scalability: for n policies, there are n^2 possible statements of precedence between applicable policies. We expect that some explicit precedence statements will always be required—otherwise, all resolution could have been built into the **Decision** lattice, as explained above. The more precedence statements that an administrator is required to make, however, the greater the administrative burden and the more likely it is for mistakes to be introduced.

The likelihood of multiple policies applying to a service request, as well as of unintended policy interaction, is significantly higher in the case of cross-domain interactions: not only are there are more independent decision makers, but credentials and attributes may need to be translated between different standards, and in-domain and cross-domain policies may interact, as in our example from Figure 1.

Figure 3 shows some of the many ways in which in-domain and cross-domain information can interact to produce many different applicable policy combinations. In addition to within-domain policies, cross-domain policies introduce two additional relations:

- A *cross-domain reference* is the inclusion of a **Credential** or **Attribute** from System X in a policy originated by System Y.
- A **Mapping** takes a set of **Credentials** or **Attributes** from System X and transforms them into **Credentials** or **Attributes** on System Y.

Note that a **Mapping** could potentially involve multiple systems (i.e. Acme grants a certain access only to Bacchae staff who are also dealing with a mutual contract with FedEx). We believe this introduces much additional complexity for little gain, and so consider only pairwise **Mappings** for now.

4.1 Cross-Domain Mappings

Because the set of **Attributes** and **Policies** many vary greatly across domains, mappings are extremely general. We will assume, however, that mappings are monotonic: that is, mappings can only add applicable **Credentials** and **Attributes**, not delete them (though policy can prevent them from crossing a boundary). This is an important assumption because otherwise it is possible to construct a set of mappings that does not resolve, but deadlocks in an infinite loop of addition and deletion.

The cardinality of an **Attribute** mapping may be any of several cases (illustrated in Figure 4):

- **One-to-one:** e.g. System X treats all analysts from System Y as though they were System X analysts.
- **Many-to-one:** e.g. System X distinguishes between junior and senior analysts, while System Y only has one type of analyst. System X combines the System Y analyst **Attribute** with the number of years of service to determine whether to treat a System Y analyst as a junior or senior System X analyst.

- **One-to-many:** e.g. System X treats all analysts from System Y as though they were System X analysts, except that they are also given a "foreign" attribute that limits some accesses. This case is distinct from multiple one-to-one mappings because it encodes the intent that these attributes *must* be linked together (i.e. tagging an outsider with the "foreign" flag as their attributes are translated to the local domain).

- **Many-to-many:** e.g. a combination of the previous two cases.

Moreover, there may be multiple mappings that can produce the same output attributes, as when the many-to-one example of senior and junior analysts is reversed, such that two types of System Y analyst both map to the same type of System X analyst. Note that although all cases might be viewed as special cases of many-to-many mappings, we begin by distinguishing these cases because we expect that there may be stronger inferences or more efficient heuristics that can be applied in the narrower cases and also that a large fraction of mappings in real systems will be instances of the narrower cases.

Mappings may involve **Credentials** as well as **Attributes**:

- A **Credential** in one domain can map to a **Credential** in another, e.g. Alice has accounts on both System X and System Y.

- A **Credential** in one domain can map to **Attributes** in another domain, e.g. Alice's account on System Y is given the "visiting analyst" attribute by System X.

Although there is no theoretical bar to arbitrary mappings involving **Credentials**, we believe that practically it is reasonable to assume that attributes do not map to **Credentials**, that **Credential** to **Attribute** mappings never involve multiple **Credentials**, and that **Credential** to **Credential** mappings are always one-to-one. It might also be possible to map other elements from one domain to another, but doing so is not central to the problem so we do not consider these cases at present.

In order to determine the applicable **Policies** for a service request, it is necessary to find paths between the service and the client, where a path is a chain from **Policy** to (local or cross-domain) **Attributes** or **Credentials** referenced by the **Policy**, and thence on through cross-domain mappings and local **Credential**/**Attribute** relations to the original client **Credential**. Much of this search for paths might be cached via dynamic programming, of course—for example, the paths connecting Bob to $Logistics_A$, discovered when he accesses inventory data on A can be reused when he attempts to access shipping data on A. In general, it is necessary to find only those paths capable of producing a **Conflict** in the **Decision**, but in many cases we expect it will be more expedient simply to find all paths, since there may be many ways to produce a **Conflict**.

It is easy to see that, even when only a small number of domains are involved, there is potentially a significant search task required in order to determine all applicable policies for a service request. In the example scenario in Figure 3, with Bob on System B requesting access to shipping data on System A, there are 8 different applicable policy combinations, each involving a different cross-domain path.

5. EXPLICIT AND IMPLICIT RULE PRECEDENCE

When multiple rules apply and the **Decisions** which they specify are not compatible, there must be some way of determining which rule takes precedence. In some cases, this must be declared explicitly by a human administrator. If **Attribute** information is given a lattice structure, however, similar to the lattice structure of **Decisions**, then implicit precedence relations for **Policies** may be inferred in many circumstances.

In general, the description language for **Policies** is too complex for general inference. In XACML, for example, even if the full complexity of possible rule specifications were not too complex to analyze, the ability to invoke arbitrary external code as part of a policy would make it impossible. A large subset of pragmatically useful policies, however, can be described as a conjunction of tests on **Attribute** values and external circumstances. Moreover, human thinking about policies often takes the form of defaults and exceptions, such as: "Normal employees don't get access to the financial database, but contracts and accounting do." or "Nobody outside the company gets access to the financial database, except for the auditors."

In such cases, implicit precedence can be inferred for pairs of rules where the tests for one specify a strictly stronger set of **Attributes** values and conditions. For example, these examples of default and exception policies might be specified as:

5. **Deny** $Employee_B$ read of $Financials_B$
6. ($Employee_B$ and $Contracts_B$) reads $Financials_B$
7. ($Employee_B$ and $Accounting_B$) reads $Financials_B$
8. **Deny** $Foreign_B$ read of $Financials_B$
9. $Auditor_B$ reads $Financials_B$

Policies 6 and 7 have implicit precedence over Policy 5 because they each specify a narrower class of employee, e.g. those employees who are also in accounting. Policies 6 and 7 have no implicit precedence with respect to one another, however, since $Contracts_B$ and $Accounting_B$ are independent attributes.

The more information is encoded in the **Attribute** lattice, the stronger an inference that can be made. For example, if it is declared that statements about auditors always take precedence over other statements, Policy 9 will have implicit precedence over Policy 8 and an external auditor will be given access to the company financials.

As with the **Decision** lattice, structure in the **Attribute** lattice need not be declared unless it is valuable to the human administrator. This allows a pragmatic trade-off to be made where the human can opportunistically determine whether a given family of relationships is simpler and more intuitive to declare in terms of policy precedence or attribute relationships.

6. RESOLUTION ALGORITHM

Given these mathematical foundations, we can now specify an algorithm for adjustably-autonomous policy resolution. Assuming that a Policy Decision Point (PDP) has access to the full set of **Policies**, it can determine the **Decision** for a given request via the following procedure:

1. Search to find all applicable **Policies** and evaluate to collect their **Decisions**.
2. Use explicit and implicit precedence to create a partial order on the set of applicable **Policies**.
3. Find the set of maximal elements in the set of applicable **Policies** (those where there is no other element with precedence over them).
4. The overall **Decision** is the GCS of the **Decisions** associated with the set maximal elements (or **Conflict** if an error in policy-writing has caused there to be no maximal element to exist). If the GCS is **Conflict** then human intervention is requested; otherwise the policy combination is autonomously resolved to the GCS value.

Applied dynamically in a PDP, this algorithm considers particular client requests for resources. Thus, in our privilege escalation example, Bob's request for shipping data on System A leads to two **Policy/Decision** pairs being found: {Policy 1, **Filter** (B-contracts only)} and {Policy 4, **Permit**}. No explicit precedence has been declared, and no implicit precedence exists, since $Purchaser_B$ and $Logistics_A$ are independent attributes, so both are members of the set of maximal elements. The overall decision is thus the GCS of **Filter** (B-contracts only) and **Permit**, which is **Conflict**. The possibility of privilege escalation is thus caught and prevented, though the machine must defer to a human to resolve the conflict by adjusting the **Policies**. For example, the administrator might declare that Policy 1 has precedence over Policy 4. Now if Bob asks for shipping data, both pairs are found as before, but Policy 1 take precedence. The GCS of that single **Decision** is **Filter** (B-contracts only), giving him appropriate partial access to the shipping data.

Potential conflicts may also be detected statically by considering a generic client. This client gives all possible answers to policy questions, unless there are explicit declarations that certain attributes have exclusive value sets. For example, if there are policies that Managers are permitted access to a resource and Technicians are denied access, then this will appear to be a conflict unless there is attribute type information specifying that the job category **Attribute** has precisely one value per individual or an individual can never be both a Manager and a Technician.

If no PDP has access to sufficient information about the full set of **Policies** (both internal and cross-domain), however, it is not possible to guarantee safety for the 3+ domain case, and is only possible to guarantee safety for a subset of the two domain case. It may thus be desirable for a group of interacting domains to share their **Policies**, either with each other or with a hub that has been entrusted with the job of determining applicable cross-domain policy. It may also be possible to compose analyses of policies, such that although **Policies** themselves are not shared, sufficient summary type information is shared to ensure safety.

7. CONTRIBUTIONS

We have presented a lattice-based approach to policy representation that allows some of the policy intent of a human administrator to be inferred or explicitly captured. Combining this representation with search through a set of cross-domain mapping policies enables a policy resolution algorithm that resolves routine policy interactions autonomously while flagging potential conflicts for attention from a human administrator.

To move from this foundation to practical deployment in real cross-domain scenarios, some key challenges must be addressed: a succinct means of specifying and maintaining policies must be developed, the framework must be proved to sufficiently capture administrator intent, and it must be instantiated in a form where its safety is certifiable. Finally, since organizations may not wish to share their entire policies with one another, it is an open question how to identify the minimum information must be shared in order to safely resolve policy interactions.

Acknowledgements

The authors would like to acknowledge the support and collaboration of the US Air Force Research Laboratory (AFRL) Information Directorate. This material is based upon work supported by the Air Force Research Laboratory under Contract No. FA8750-10-C-0131.

8. REFERENCES

[1] J. Hodges et al. (March, 2005) "Glossary for the OASIS Security Assertion Markup Language (SAML) V2.0."
[2] Tim Moses. (1 Feb 2005) eXtensible Access Control Markup Language (XACML) Version 2.0.
[3] Bill McCarty, SELinux: NSA's Open Source Security Enhanced Linux.: O'Reilly, 2005.
[4] Karl MacMillan, and David Caplan. Frank Mayer, "SELinux by Example.," Open Source Software Development Series, vol. Prentice Hall, 2007.
[5] NSA Web Site. [Online]. http://www.nsa.gov/ia/programs/h_a_p/releases/index.shtml
[6] OpenID Web Site. [Online]. http://openid.net
[7] Shibboleth Web Site. [Online]. http://shibboleth.internet2.edu/, August 2009
[8] David Chadwick and Gansen Zhao, PERMIS: a modular authorization infrastructure, Concurrency and Computation: Practice and Experience, vol. 20, no. 11, pp. 1341-1357, 2008.
[9] Peter White, Security Configuration Domain Specific Language (DSL), 2008 SELinux developer summit, [Online], http://selinuxproject.org/files/2008_selinux_developer_summit/2008_summit_white.pdf
[10] Tonti, G., Bradshaw, J. M., Jeffers, R., Montanari, R., Suri, N., & Uszok, A. (2003). Semantic Web languages for policy representation and reasoning: A comparison of KAoS, Rei, and Ponder. International Semantic Web Conference (ISWC 03). Sanibel Island, Florida.
[11] Java Theorem Prover (JTP), [Online]. http://www-ksl.stanford.edu/software/jtp/
[12] The Alloy analyzer, [Online], http://alloy.mit.edu/community/
[13] David Elliott Bell, "Looking Back at the Bell-La Padula Model," Washington, DC, USA, 2005.
[14] Biba, K. J. "Integrity Considerations for Secure Computer Systems", MTR-3153, The Mitre Corporation, April 1977.