A MATCHMAKING APPROACH FOR DISTRIBUTED POLICY SPECIFICATION AND INTERPRETATION

By

Nicholas Coleman

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCES)

at the

UNIVERSITY OF WISCONSIN – MADISON

2007

Abstract

In a distributed system, the separation of policy and mechanism is a vital principle. This separation can be achieved by devising a language for specifying policy and an engine for interpreting policy. In the Condor [52] high throughput distributed system the ClassAd language [57] is used to specify resource selection policy and matchmaking is used to interpret that policy. ClassAds and matchmaking are not currently used for authorization policies in Condor. SPKI/SDSI [21] is a public key infrastructure for authorization policy. This dissertation shows that ClassAds and matchmaking can implement SPKI/SDSI, thereby complementing the resource selection policy capabilities of Condor with the authorization policy capabilities of SPKI/SDSI. Techniques for policy analysis in the context of resource selection and authorization are also presented.

The ClassAd language is based on the concept of classified advertisements. Entities in Condor are represented by classified advertisements or ClassAds. Each job submitted by a condor user has a corresponding ClassAd as does each compute machine. The matchmaking process pairs jobs with machines based on the policies expressed in their ClassAds. Since the bilateral matchmaking framework is not sufficient for assembling three or more parties a multilateral matchmaking framework, *gangmatching*, is required in such cases. A collection of three or more ClassAds that satisfy each others requirements is called a *gang*.

SPKI/SDSI is an infrastructure for expressing authorization policy using public key encryption. Two kinds of certificates can be issued by a principal. An *authorization certificate* grants another principal a set of access rights for a resource as well as the permission to delegate these rights to other principals. A *name certificate* creates a name for another principal or set of principals. A combination of several certificates that authorize a principal to access a resource is called a *certificate chain*. The problem of assembling a suitable *certificate chain* for a given authorization is called the *certificate chain discovery problem* [12].

In the case of bilateral matchmaking, this dissertation presents analysis techniques for solving two key problems: *I Don't Like Anyone* and *Nobody Likes Me*. The *I Don't Like Anyone* problem occurs when the resource selection policy for a resource request rejects all available resources. The *Nobody Likes me* problem occurs when the resource selection policies for resource offers reject a given resource request.

Furthermore, this dissertation presents a representation of SPKI/SDSI certificates using the ClassAd language in a gangmatching context. A ClassAd representing a certificate is composed of several nested ClassAds called *ports*. One of these ports offers the certificate for use in a chain. If needed, additional ports request other certificates to resolve a SPKI/SDSI name or delegate an authorization. A gang of such ClassAds corresponds to a chain of certificates.

In order to support the capability to reuse a certificate indefinitely in a chain while avoiding infinite loops, a modified algorithm for gangmatching is presented. Formal definitions of the static and dynamic structures used by this algorithm are stated along with formal definitions of the concepts of *equivalence*, *evaluation*, and *validity*. The algorithm itself is then presented, its correctness is proved, and its complexity is analyzed.

In the case of multilateral matching, two more matchmaking analysis problems are presented along with their solutions: *Break the Chain* and *Missing Link*. The *Break the chain* problem occurs when an authorization policy grants an access that needs to be revoked. To revoke an authorization, a set of certificates must be invalidated such that no chain can be constructed granting the authorization. A new algorithm using the results of the gangmatching algorithm identifies a set of ClassAds representing such certificates. The *Missing Link* problem occurs when a desired authorization is not granted by any certificate chain. A modified version of the gangmatching algorithm identifies the additional certificate ClassAds needed to complete a gang representing a chain of certificates granting the desired authorization.

The algorithms and strategies presented in this dissertation comprise a rich framework for policy specification and interpretation suitable for resource selection and authorization. The use of the ClassAd language for policy specification and matchmaking for policy interpretation achieves a separation of policy and mechanism. The combination of supporting resource selection policies and authorization policies provides robust support for resource allocation in a distributed environment. The techniques for matchmaking analysis provide a means for understanding why existing policies do not produce desired results.

Acknowledgments

First and foremost I would like to thank Dr. Miron Livny and Dr. Marvin Solomon for giving me the opportunity to work on the Condor project for seven years. Many thanks to the ever expanding Condor team, both students and staff. The experience and wisdom I gained working on Condor has been significant, though the most significant piece of wisdom is that I still have so much more to learn. In particular I would like to thank Pete Keller, Peter Couvares, Todd Tannenbaum, Derek Wright, Dr. John Bent, Dr. Doug Thain, Ian Alderman, Nick LeRoy, and Dr. Alain Roy for guiding me through the maze that is Condor. I would also like to thank Dr. Marvin Solomon for his patience, frank advice, stories from the good old days of Computer Science, and most particularly for believing in me when it seemed that no one else would.

Several people helped with editing earlier versions of the material in this dissertation including Dr. Hao Wang, Dr. Stefan Schwoon, Dr. John Bent, Dr. Somesh Jha, Dr. Rajesh Raman, and Dr. Marvin Solomon. My research would not have been possible without the strong shoulders of Dr. Rajesh Raman, the inventor of the ClassAd language and gangmatching, to stand on. Though it is cited many times in this work, I must make special mention of the work by Dr. Somesh Jha and Dr. Tom Reps on SPKI/SDSI and Pushdown systems, which served as a starting point for my exploration of gangmatching and SPKI/SDSI. My sincere gratitude goes to Dr.

James Cercone and the Department of Computer Science at West Virginia University Institute of Technology for providing me with an opportunity to work at Tech as a Visiting Professor, without which the completion of my degree would not have been possible. I might have never applied to graduate school if it hadn't been for the encouragement of Dr. James Lipton at Wesleyan University, who also first introduced me to programming language theory and mathematical logic.

Last, but certainly not least, I would like to thank my family and especially my fiancée Heather James for a seemingly infinite amount of patience and support. Heather has stuck with me through thick and thin, and has continued to have faith in me even when I had little faith in myself.

Contents

Chapter 1

Introduction

One of the challenges of distributed computing environments is the specification and interpretation of policy. The separation of policy and mechanism has long been one of the key principles in systems design. This principle simplifies the specification of policies and keeps them independent of implementation changes. One way of achieving separation is to provide a policy framework consisting of a language for specifying policies and an engine for interpreting these policies in the context of a given set of system conditions. The flexibility of such a framework is particularly suitable for resource allocation policy in a distributed system.

Distributed systems are dynamic in that principals and resources may join or leave the federation at any time. Allocation of resources in a decentralized environment requires policy for resource selection and access control. Resource selection is the process of finding resources that satisfy a principal's requests. Access control policies determine whether the principal is permitted to access the resources. Currently there is no single language or framework that deals with authorization and resource selection policies.

Advertising languages such as the ClassAd language used by Condor [52], a widely used production-quality distributed computing system, provide a means for expressing resource selection policies. Offers and requests for resources are represented by classified advertisements (ClassAds). These ClassAds are then subjected to a matchmaking process that attempts to find compatible offers for a given request and pick the "best" offer as determined by the requester's preferences and system wide policies. Access control policy in Condor is not specified in this fashion; instead, policy is expressed in a configuration file. Furthermore the allowed or denied accesses are at the granularity of a single machine, not individual files or devices on that machine [1].

Trust management systems [9] like SPKI/SDSI [21] define formal languages for expressing access control policies in distributed environments and provide algorithms to determine if a principal is authorized to access a resource. SPKI/SDSI *name certificates* define a name space that allows a principal to refer to other principals indirectly. SPKI/SDSI *authorization certificates* grant a principal access to a resource, and may allow the principal to delegate that access. A principal must present a set of these certificates, called a *certificate chain*, to gain access to a resource. The problem of marshaling such a set is known as *certificate chain discovery*. Solutions to this problem based on formal language techniques can be found in [12, 29].

An implicit assumption in these systems is that policies for resource selection and access control are independent of one another. This assumption fails when a resource that satisfies a principal's requirements is selected, but cannot be accessed by the principal without proper credentials. One solution to this problem is use a multilateral matchmaking or *gangmatching* [53] paradigm to select resources and assemble the credentials necessary to access those resources. Gangmatching, described more thoroughly in [51], does not address the situation where an unspecified quantity of credentials is required to access a resource, as is often the case in a trust management system like SPKI/SDSI.

The major contributions of this dissertation are as follows:

- A framework for the bilateral matchmaking analysis used by Condor to identify problems with resource selection policies. This framework is applied to the problem in Condor of identifying why a job can not be matched with any available machines.

- A framework for multilateral matchmaking analysis with applications for identifying problems with authorization policies. The specific authorization problems are the failure of a desired authorization and the success of an undesired authorization.

- A ClassAd representation for SPKI/SDSI certificates that allows certificate chains to be assembled using a gangmatching algorithm. The construction of certificate chains from these ClassAds correctly implements SPKI/SDSI semantics for certificate compatibility and composition.

- An enhanced gangmatching algorithm capable of assembling certificate chains comprised of an unspecified quantity of certificates.

- Formal definitions of structures and concepts employed in the enhanced gangmatching algorithm. The primary concepts involved are equivalence of intermediate structures and partial evaluation of ClassAd expressions.

A working definition of policy is needed before discussing the details of the framework for policy specification and interpretation. The IETF Networking Group defines policy in two ways [66]:

1. A definite goal, course or method of action to guide and determine present and future decisions.

2. A set of rules to administer, manage, and control access to network resources.

More informally, Arpaci-Dusseau et al [4] define policy as the scheme for deciding what should be done.

When discussing policy it is also important to distinguish between *specification*, *interpretation* and *enforcement*. In order to specify policies robustly one needs a language that can describe the principals in a system, the state of the system or parts thereof, and conditions that must be satisfied. Once one is able to specify policies a facility is needed to interpret them in a given context. The interpretation stage is where the policy decisions indicated by the above definitions are made. Finally, once a given decision is made actions must be taken to enforce the decision. This dissertation lays out a matchmaking framework using the ClassAd language to provide policy specification and interpretation for distributed systems.

Chapter 2 provides background information on ClassAds, gangmatching, and

SPKI/SDSI. Chapter 3 presents an analysis framework for bilateral matchmaking with the ClassAd language. Chapter 4 describes a ClassAd language representation of SPKI/SDSI certificates, shows that a matching set of these ClassAds is equivalent to a corresponding certificate chain, and recasts the certificate chain discovery problem as a gangmatching problem. Chapter 5 presents a gangmatching algorithm that can handle the reuse of ClassAds, necessary for SPKI/SDSI certificate chain discovery, and shows that the application of the algorithm to ClassAds representing SPKI/SDSI certificates has the same worst case time complexity as the *post** algorithm for certificate chain discovery presented in [29]. Chapter 6 extends the bilateral matchmaking analysis framework to gangmatching. Chapter 7 explores related work, Chapter 8 presents directions for future work and concludes the dissertation.

Chapter 2

Background

2.1 The ClassAd Language and Gangmatching

The ClassAd language is used by Condor primarily to advertise resources and requests for those resources in a distributed environment. An advertisement, called a *ClassAd*, represents an offer of or request for a resource and consists of named descriptive attributes, constraints and preferences. The constraints are expressed by an attribute named `Requirements`, and the expression of the preferences is named `Rank`. [1] A matchmaking process is used to discover offers and requests that satisfy one another's constraints and best suit one another's preferences. If more than two parties are involved – such as a job, a machine, and a license – a bilateral matchmaking scheme is insufficient and a multilateral framework, called *gangmatching* [53], must be used.

In the gangmatching framework a multilateral match is broken down into several bilateral matches. A set of ClassAds that satisfy one another's constraints is called a *gang*. Each ClassAd contains a list of nested ClassAds called *ports*, each of which

[1] To simplify matters this dissertation deals only with `Requirements` expressions and omits `Rank` expressions from example ClassAds.

represents a single bilateral match. A gang is *complete* if all ports of all ClassAds in the gang have been successfully matched to ports of other ClassAds in the gang. A port that has not been matched is an *open* port. Given a port P of a ClassAd and a potentially matching port P' of another ClassAd, a reference in P to an attribute `attr` defined in P' is represented as `other.attr` to distinguish it from a reference to an attribute in P. In addition, P has a label that is used by subsequent ports in the same ClassAd to reference attributes defined in P'. If P's label is `label`, a reference in a subsequent port to an attribute `attr` defined in P' is represented as `label.attr`. The attribute `attr` is *imported* from P' and is called an *imported attribute*.

Figure 1 shows a gangmatching ClassAd representing a job. The ClassAd has two ports: the first requests a machine to run the job, and the second requests a license to run a particular application on that machine. In the `Requirements` expression of the first port of the job ClassAd, a reference to the attribute `Memory`, imported from a matching ClassAd representing a machine, is expressed as `other.Memory`. The port is labeled `cpu`, and the subsequent port contains a reference to the `Name` attribute imported from the ClassAd matching the first port expressed as `cpu.Name`. In contrast, a locally defined attribute like `ImageSize` is referenced locally without using a prefix.

A gang is tree-structured, which means that some ClassAds may not express constraints on other ClassAds directly. For example, in Figure 1 the job ClassAd contains a port requesting a machine and another port requesting a license. The license

```
[
  Ports = {
     [ // request a workstation
       other = cpu;
       Type = "cpu_request";
       ImageSize = 28M;
       Requirements =
         other.Type == "Machine" &&
         other.Arch == "INTEL" &&
         other.OpSys == "LINUX" &&
         other.Memory >= ImageSize
     ],
     [ // request a license
       other = license;
       Type = "license_request";
       CPUName = cpu.Name;
       Cmd = "run_sim";
       Requirements =
         other.Type == "License" &&
         other.App == Cmd
     ]
   }
]
```

Figure 1: A gangmatching ClassAd for a job

and machine ClassAds that match may each contain a port expressing constraints on the job, but may not have ports expressing constraints on one another. This restriction can be circumvented if the job exports attributes imported from the machine ClassAd in the license port. In Figure 1 the `Name` attribute of the `cpu` ad is exposed in the license port by the definition `CPUName = cpu.Name`. The matching license ClassAd can indirectly reference the `Name` attribute of the machine ClassAd as `other.CPUName`. Circular dependencies are avoided by the restriction that a port may only use imported attributes from previous ports.

2.2 SPKI/SDSI

SPKI/SDSI is a trust management system that specifies access control policies using certificates. A SPKI/SDSI certificate is a declaration by a principal, the *issuer* of the certificate, about the naming of another principal, the *subject* of the certificate, or the authorization for the subject to access a resource.

Principals are represented by a unique public key. They may also be referred to indirectly by a *SPKI/SDSI name*. A SPKI/SDSI name consists of a public key followed by zero or more identifiers. The identifiers navigate a hierarchical name space, similar to a hierarchical directory structure. For example, if K_A represents the principal named Alice, then the SPKI/SDSI name "K_A Bob Carol" can be resolved by looking up the identifier "Bob" in Alice's namespace. Assuming that K_A Bob resolves to K_B, Bob's public key, the identifier "Carol" must now be looked up in Bob's namespace. If Bob has defined the identifier "Carol" to resolve to K_C, Carol's public key, then "K_A Bob Carol" is equivalent to the SPKI/SDSI names "K_B Carol" and "K_C."

A name certificate (*name cert*) defines a name in the issuer's local name space by assigning an identifier to a SPKI/SDSI name that represents the subject of the certificate. An authorization certificate (*auth cert*) indicates that the issuer (represented by a public key) authorizes the subject (represented by a SPKI/SDSI name) to access a resource. Both the resource and the permission being granted are specified in an auth cert. For the purposes of this dissertation we are only concerned with a single anonymous resource and a generic operation on that resource. An auth cert also indicates whether or not the authorization may be delegated.

In this dissertation we shall adopt the representation of certificates as rewrite rules with the issuer on the left and the subject on the right as introduced in [12]. Four examples of this rewrite rule representation are shown in Figure 2.

$$\begin{gathered}
(1)\ K_R \,\square \rightarrow K_A \text{ Bob } \square \\
(2)\ K_A \text{ Bob} \rightarrow K_B \\
(3)\ K_B \,\square \rightarrow K_B \text{ Carol } \blacksquare \\
(4)\ K_B \text{ Carol} \rightarrow K_C
\end{gathered}$$

Figure 2: SPKI/SDSI certificates as rewrite rules

There are four principals involved in the example certificates in Figure 2: the administrator of resource R (identified by the public key K_R), Alice, Bob, and Carol (identified by their public keys K_A, K_B, and K_C). Certs (2) and (4) are name certs that indicate that the identifier "Bob" in Alice's name space represents Bob's key, and the identifier "Carol" in Bob's name space represents Carol's key. Certs (1) and (3) are auth certs, denoted by the $\square$ after the subject. In cert (1), the subject "K_A Bob" is granted access to the resource R. The $\square$ at the end indicates that the subject may delegate this access right. Similarly, cert (3) grants the subject "K_B Carol" access to whatever K_B has access to. The $\blacksquare$ at the end of this cert indicates that the subject may not delegate this access right.

The use of delegation and an indirect naming scheme means that more than one certificate may be necessary for a principal to access a resource. Such a set of one or more certificates is called a *certificate chain*. A certificate chain may also be represented by a rewrite rule, derived from the composition of compatible certificates.

As defined in [12], certs $C_1 = K_1\ A_1 \rightarrow S_1$ and $C_2 = K_2\ A_2 \rightarrow S_2$ are *compatible* if $S_1 = K_2\ A_2\ X$ for some sequence of zero or more identifiers X (that is $K_2\ A_2$ is a prefix of S_1). The *composition* of C_1 and C_2, written as $C_1 \circ C_2$ is defined by replacing the prefix of S_1 with S_2. Using the term rewriting notation:

$$C_1 = K_1\ A_1 \rightarrow K_2\ A_2\ \mathrm{X}$$

$$C_2 = K_2\ A_2 \rightarrow S_2$$

$$C_1 \circ C_2 = K_1\ A_1 \rightarrow S_2\ X$$

Certificate chains are built by repeated use of composition.

Returning to the examples in Figure 2, we can form cert chains by composing compatible certificates. (1) ∘ (2) = $K_R\ \square \rightarrow K_B\ \square$ authorizes K_B to access resource R and to delegate that access right; (3) ∘ (4) = $K_B\ \square \rightarrow K_C\ \blacksquare$ grants K_C access to whatever K_B has access to. Putting these two chains together we get the chain ((1) ∘ (2)) ∘ ((3) ∘ (4)) = $K_R\ \square \rightarrow K_C\ \blacksquare$ that authorizes K_C to access resource R, but not to delegate that access right. The problem of assembling such a chain is called the certificate chain discovery problem. Solutions based on formal language techniques can be found in [12, 29].

Chapter 3

Matchmaking Analysis

Occasionally in Condor a submitted job's ClassAd does not match with any machine ClassAds. This situation occurs when none of the machines meet the submitted job's requirements, when the job does not meet the requirements of the machine candidates, or a combination of these two circumstances. In this chapter we will treat the first two problems separately and assume that they are not related. Using a dating service analogy we refer to the first case as *I Don't Like Anyone* and the second as *Nobody Likes Me*.

Requirements expressions are most commonly in *disjunctive normal form* (DNF), a disjunction of conjunctions of atomic Boolean propositions. In most requirements expressions the atoms are in the form of predicates that relate an attribute to a literal value or constant by means of a comparison relation. An example of such a predicate is

```
other.OpSys == "LINUX".
```

We shall often refer to *clauses* that are conjunctions of predicates in this form. An example of such a clause is the following:

```
(other.OpSys == "LINUX") &&
(other.Arch == "INTEL") &&
(other.Memory >= 512M)
```

In the majority of ClassAds a requirements expression consists of a single clause but sometimes they are disjunctions of such clauses. We shall assume all requirements expressions are in DNF. There are well known algorithms for transforming arbitrary arbitrary expressions into this form, but we find that in practice, most requirements are already in DNF. Additionally any atom that is not in the form of a predicate as we have described may be treated as an atom that cannot be modified.

We may look at the matchmaking process geometrically, where the collection of attributes with literal values in a ClassAd is represented by a point in n dimensional space where each dimension corresponds to a single attribute and n is the total number of attributes in the ClassAd. Clauses are represented in this space by n dimensional rectangles, or *hyper-rectangles*. The *I Don't Like Anyone* case consists of a single hyper-rectangle (the job requirements expression) and many points (the machine ClassAds) that lie outside the hyper-rectangle. The goal is to expand the hyper-rectangle to enclose at least one point. The *Nobody Likes Me* case is the reverse with many hyper-rectangles (the machine requirements expressions) and one point (the job ClassAd) that lies outside their union. Here we wish to relocate the point so that it lies within some hyper-rectangle.

In both cases we need a measure of distance between a point and a hyper-rectangle so that we can make the smallest adjustment possible. A number of factors come into play here: the number of predicates or attributes we have to modify, how much we have to modify a given predicate or attribute, how many matches will we get for a given set of modifications, and what kind of machines will we match with. A simple algorithm would be to focus exclusively on the first factor, but there may be situations when several minor modifications are more attractive than one major one. A more complex algorithm would give weights to different job or machine attributes, and in the case of attributes that take non-numerical values identify which values are "closer" than others. This would require collecting a substantial amount of information from the user.

The distance metric we shall use lies somewhere between the simple and complex approaches. First, we calculate the distance in each dimension separately. For numerical values the projection of the hyper-rectangle is an interval or (infrequently) a set of intervals. We begin by computing the absolute value of the difference between the base point and the closest point in the interval. We then divide this difference by the difference between the maximum and minimum values taken by the given attribute. For non-numerical values the projection of the hyper-rectangle is a point (or set of points) and the distance between two points is zero if they are equivalent and one otherwise. This definition assures that regardless of type, one dimensional distance will always be between zero and one inclusive. Finally we sum the one dimensional distances to get a composite distance. Taking the sum (sometimes known as the "taxicab norm") is preferable to using the Euclidean norm as it favors modifying a smaller number of predicates or attributes.

In the *I Don't Like Anyone* case, we can also detect *conflicts* within the requirements expression, that is identify which predicates in a requirements expression clause are incompatible with one another. This situation may arise out of user error, such as misspelling an attribute or a string value. Alternately there may simply be no machines that satisfy a certain combination of predicates. If this is the case it would be useful to identify the smallest subset of predicates that cannot be satisfied.

3.1 I Don't Like Anyone

First, we shall examine the case where no available machines match a submitted job's requirements expression. Depending on ones point of view, the problem is either with the requirements expression of the job, or with the attributes of the various machine ClassAds that are referenced in the job's requirements. As ClassAd analysis is primarily concerned with aiding the user who has submitted the job we shall focus on the job's requirements expression. First we must indicate which predicate or combination of predicates in a given clause is causing the problem. Once the offenders have been identified we may use the machine ClassAds to suggest possible modifications to the expression. It may well be that the job requirements are non-negotiable, and may not be relaxed or modified. In this case the analysis is still pertinent as it provides useful information about the current pool of available machines.

On the assumption that the job requirements expression may be modified, we shall examine how to form useful suggestions to the user in this regard. Our goal in this end is to find the least drastic modification to the expression that results in a successful match. In order to achieve this algorithmically we need a precise metric for the degree to which an expression is modified. We shall use the metric described in the beginning of this section.

Additionally we need to define exactly what constitutes a modification to a predicate. For our purposes we will allow either a modification to the value part of a predicate, or the complete removal of the predicate. If the predicate has an equality operator the value may be changed to anything as long as it has the same type as the original value. In the case of an inequality the value should only be modified so as to relax the predicate, as a stricter predicate will get us nowhere. If the operator in question is a not-equals operator, the only sensible modification is to remove the predicate altogether. Removal is also the best strategy if the attribute is not defined in any machine ClassAd.

As an example consider the following clause of a requirements expression:

```
(other.Arch == "ALPHA") &&
(other.OpSys == "SOLARIS") &&
(other.Memory >= 512M) &&
(other.Disk >= 14M)
```

In this example there are only four machine attributes we care about: `Arch`, `OpSys`, `Memory`, and `Disk`. From this clause we construct a table whose columns correspond to attributes referenced in the predicates of the clause and whose rows correspond to the machine ClassAds. Continuing with our example, here is such a table:

Machine ClassAd	Arch	OpSys	Memory	Disk
1	"ALPHA"	"LINUX"	256M	10G
2	"INTEL"	"LINUX"	256M	20G
3	"SPARC"	"SOLARIS"	1024M	10G
4	"INTEL"	"LINUX"	512M	10G
5	"ALPHA"	"LINUX"	512M	10G
6	"SPARC"	"SOLARIS"	1024M	20G
7	"INTEL"	"LINUX"	256M	20G
8	"SPARC"	"SOLARIS"	256M	10G

Computing the distances for each attribute we get:

Machine ClassAd	Arch	OpSys	Memory	Disk	Total Distance
1	0	1	0.333	0	1.333
2	1	1	0.333	0	2.333
3	**1**	**0**	**0**	**0**	**1**
4	1	1	0	0	2
5	**0**	**1**	**0**	**0**	**1**
6	**1**	**0**	**0**	**0**	**1**
7	1	1	0.333	0	2.333
8	1	0	0.333	0	1.333

The rows in boldface are the machines with the shortest composite distance to our clause. We can now suggest that the user should either change `(other.Arch == "ALPHA")` to `(other.Arch == "SPARC")` or change `(other.OpSys == "SOLARIS")` to `(other.OpSys == "LINUX")`. We give preference to the former as it will net the most machines, and thus give the user a better chance of getting a successful match in the future.

3.1.1 Detecting Conflicts

Another way of looking at the *I Don't Like Anyone* situation is to find predicates that conflict with one another, that is, predicates that may be satisfied by machines on their own, but are not satisfied in conjunction. In our example we have many machines running Solaris and several machines with Alpha processors, but no Alpha machines running Solaris. In this case the expression

```
(other.OpSys == "SOLARIS") &&
(other.Arch == "ALPHA")
```

represents two conflicting predicates, each of which evaluate to **true** in the context of some machine ClassAds, but in conjunction will always evaluate to **false**. Alternately, an expression may contain a conflict that will evaluate to **false** regardless of the context it is evaluated in. An example of such a conflict is the expression

```
(other.Arch == "ALPHA") &&
(other.Arch == "INTEL")
```

In this case we have two predicates that may be satisfied on their own, but together they will never be satisfies as the `Arch` attribute can only have one value. In other words, the first example happens to be false for a given set of machines while the second example is logically inconsistent and therefore unsatisfiable.

Detecting the former kind of conflict requires the evaluation of the individual expressions in the context of machine ClassAds, whereas the latter kind may be identified in isolation. To detect the latter we must separate the predicates in a clause by attribute reference. For each attribute referenced we convert the predicates to points or intervals depending on the type of the values. If the intersection of these intervals is empty (as is the case in our second example) we have identified a conflict. The remainder of this section is devoted to conflicts that are dependent on the values of the machine attributes.

To better understand the problem of conflict detection it is helpful to think of a clause as set of predicates and to construct a subset lattice, with the full clause on the top and an empty clause (semantically equivalent to **true**) on the bottom. In Figure 3 we see a lattice representation of the sub-expressions of the clause from our first example where:

p_1 is `(other.Arch == "ALPHA")`

p_2 is `(other.OpSys == "SOLARIS")`

p_3 is `(other.Memory >= 512M)`

p_4 is `(other.Disk >= 14M)`

Each subset corresponds to sub expression of the clause generated by removing certain predicates.

A given subset *succeeds* (marked with a **T**) if the corresponding expression evaluates to **true** in the context of some machine ClassAd and *fails* (marked with an **F**) otherwise. Any set in the lattice that fails and has no failing subsets is a Minimal Failing Sub-expression (MFS), enclosed by a dashed oval. Any set that succeeds and has no succeeding superset is a Maximal Succeeding Sub-expression (MSS), marked by a solid oval. This terminology comes from work in database query analysis [26].

The conflicts we are looking for are those that correspond to MFSs, that is expressions that always evaluate to **false**, but whose sub-expressions evaluate to true in some context.

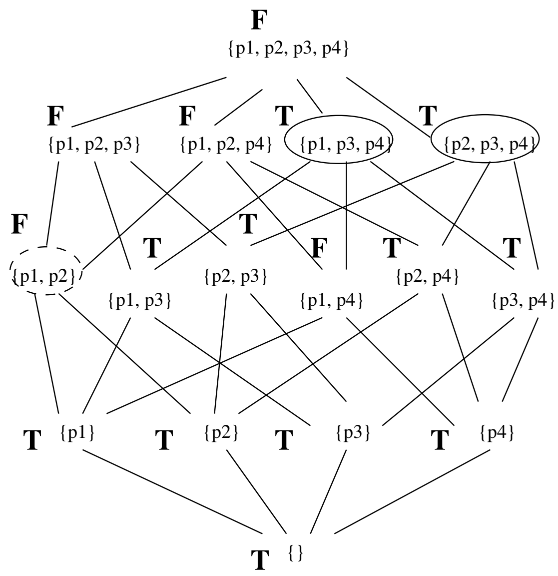

Figure 3: A subset lattice representing sub-expressions of a clause. The solid ovals are MSSs and the dashed ovals are MFSs.

Godfrey [26] shows that in the general case finding all MFSs is NP-Hard, but proposes a linear time algorithm for finding one MFS and a polynomial time algorithm for finding a fixed number k of MFSs. However, these algorithms are measured in terms of the number of database queries needed to produce the desired information. If we have a table of evaluations (generated in $m \times n$ time where m is the length of a clause and n is the number of contexts) of each of the predicates in the context of each of the machines we do not need to make a series of queries as we have all of the information we need.

The table of evaluations for our example is the following:

Machine ClassAd	(p_1) Arch == "ALPHA"	(p_2) OpSys == "SOLARIS"	(p_3) Memory >= 512M	(p_4) Disk >= 14M
1	T	F	F	T
2	F	F	F	T
3	F	T	T	T
4	F	F	T	T
5	T	F	T	T
6	F	T	T	T
7	F	F	F	T
8	F	T	F	T

The columns of this table correspond to the predicates in the clause and the rows correspond to machine ClassAds as in the previous table.

To generate the set of all MFSs we must first generate the set of all MSSs. To get the set of all MSSs we simply collect all of the unique rows of the Boolean table and prune out any rows that do not correspond to MSSs. Using the set of MSSs we derive a formula in DNF that is equivalent to the set of all succeeding sub-expressions. We then negate this formula to get a formula equivalent to all failing sub-expressions. If we convert the negated formulas to DNF and prune out any logically redundant sub-formulas we have a formula that contains all of the MFSs (and nothing but the MFSs) as clauses.

Based on the entries in the Boolean table the set of succeeding sub-expressions may be represented by the formula:

$$(\neg p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge \neg p_2 \wedge \neg p_3) \vee (\neg p_1) \vee (\neg p_1 \wedge \neg p_2) \vee (\neg p_2) \vee (\neg p_1) \vee$$
$$(\neg p_1 \wedge \neg p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge \neg p_3)$$

pruning out redundancies we get:

$(\neg p_1) \vee (\neg p_2)$

Its negation is:

$(p_1 \wedge p_2)$

This result is already in DNF, and corresponds exactly to the MFS we seek.

The main drawback to this process is that converting the negated formula from CNF to DNF may result in an exponential blow up in the size of the formula. This is not a grave concern, as in practice these requirements expressions are not very long, at least with respect to the number of machine ClassAds.

3.2 Nobody Likes Me

The converse of the *I Don't Like Anyone* situation is *Nobody Likes Me*. Instead of the job ClassAd requirements expression rejecting all machines, all of the machine ClassAd's requirements expressions reject the job. Therefore we must examine multiple expressions in DNF in the context of a single job ClassAd. Our focus is providing information for the user who submits the job, so we must look at this in terms of the attributes of the job ClassAd. Just as we sought to suggest modifications to the job requirements expression in the previous section, we shall endeavor to find potential modifications to the job ClassAd's attributes. It is even possible that crucial attributes may be missing from the job ClassAd entirely.

In the semantics of the ClassAd language a reference to a nonexistent attribute evaluates to *undefined*. For the purpose of matchmaking, *undefined* is equivalent to *false*. Since we wish to distinguish between attributes that are not defined in a job ClassAd and attributes with values that cause machine requirements expressions to evaluate to false, this distinction is crucial.

The algorithm for *I Don't Like Anyone* in effect finds the closest point to the hyper-rectangle and generates suggestions to expand the hyper-rectangle to include it. Now, given a single point, we wish to find the closest of several hyper-rectangles. We shall use this to suggest changes to the job attributes so that the point may be relocated within the closest hyper-rectangle, and thus the job ClassAd will be accepted by some machine ClassAd's requirements expression.

Figure 4 shows the geometric equivalent of two clauses where clause 1 is

```
(ImageSize >= 128M) &&
(MemoryRequirements >= 512M)
```

and clause 2 is

```
(ImageSize >= 64M) &&
(MemoryRequirements >= 1024M)
```

We can find the closest hyper-rectangles by applying the distance algorithm discussed in the beginning of the section. This method is sufficient for finding the nearest point, or smallest overall change to the attributes in the job ClassAd. However, one might wish for more detailed information, such as how many machines would match with the job if the changes described above are made. In order to be concise we should partition the space covered by the hyper-rectangles representing machine requirements expressions into equivalence classes. Each partition corresponds to a range of job attribute values that satisfy the requirements of a unique set of machines.

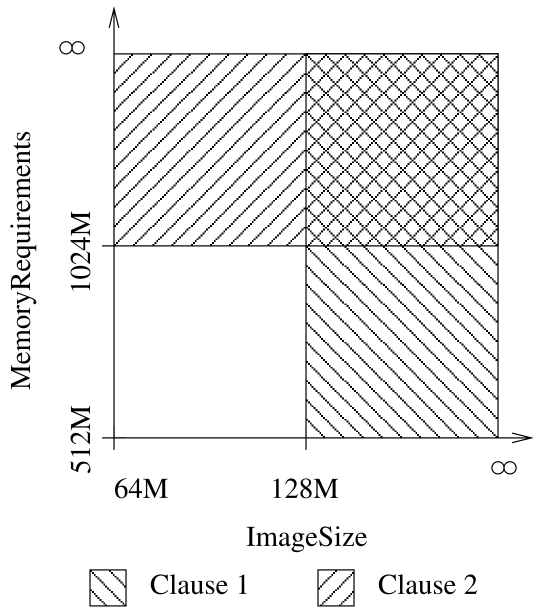

Figure 4: A geometric representation of two clauses

The first step in this process is generating the hyper-rectangles. If a machine requirements expression contains more than one clause, we create a separate hyper-rectangle for each clause, as the union of all such hyper-rectangles represents the space covered by the entire expression. Given a clause, we may treat each predicate as an interval (or set of intervals) in the dimension corresponding to its attribute.

An equals operator in the predicate defines a point, a not-equals operator defines the union of two open intervals comprising all values except for the value in the predicate, and any other inequality operator defines an open or closed interval from the value to positive or negative infinity. If there are multiple predicates with the same attribute, we find the intersection of all of the intervals they represent. If this intersection is empty, we have found a logical conflict as described in the previous section. Since we are dealing with machine ClassAds, we simply throw out this clause as it will never be satisfied.

The second step is partitioning along each dimension separately, taking care to keep track of which partition any given clause of a machine requirements expression belongs to. For example, clause 1 (c_1) in Figure 4 contains the predicate

```
other.Memory >= 512M
```

and clause 2 (c_2) contains the predicate

```
other.Memory >= 1024M.
```

These clauses creates two intervals: $[512, 1024)$ and $[1024, +\infty)$ corresponding to the sets $\{c_1\}$ and $\{c_1, c_2\}$. We continue this process until all clauses in each machine requirements expression are processed, then repeat the process for all dimensions. If there is an attribute that is referenced in some clauses but not others we simply represent that clause as $(-\infty, +\infty)$, the set of all strings, or the set of all Boolean values depending on the inferred type of the attribute (determined by the first value associated with that attribute).

One tricky aspect is dealing with a predicate like

```
other.Owner != "ncoleman".
```

If the type of the value were Boolean the solution would be trivial, and we have already defined how a not-equals operator is to behave among numerical values.

In this case we invent a special string value called *AnyOtherString*. If a clause of a machine requirements expression contains such a predicate, that clause is added to every partition in the `Owner` dimension except the one corresponding to the string `"ncoleman"`, as well a partition corresponding to AnyOtherString (if the AnyOtherString partition does not exist we create one). We shall see that it is important to keep track of which strings are not represented by AnyOtherString in a given dimension.

The third step is constructing the n dimensional partitions by taking the cross product of the vectors of intervals in each dimension. Given interval $[64, 128)$ in the `ImageSize` dimension with clause set $S1 = \{c_2\}$ and interval $[1024, +\infty)$ in the `MemoryRequirements` dimension with clause set $S2 = \{c_1, c_2\}$ we create a rectangle defined by $[64, 128) \times [1024, +\infty)$, and associate it with the intersection of $S1$ and $S2$, namely $\{c_2\}$. What this means is that job ClassAds with attribute values in the range defined by $[64, 128) \times [1024, +\infty)$ will match the machine corresponding to c_2. We continue the process with all intervals, and with all dimensions.

If we run into the AnyOtherString placeholder in a dimension with string values, we make note of all of the other string values we have encountered in that dimension and annotate any hyper-rectangle created using this instance of AnyOtherString with these values. For example, if the dimension corresponds to the `Owner` attribute and the other string values are `"ncoleman"` and `"raman"`, then AnyOtherString means any value for `Owner` except `"ncoleman"` and `"raman"`.

Finally we have a set of hyper-rectangles each associated with a subset of clauses (and therefore a subset of machines) that partitions the space. We may need to clean up this set by adjoining hyper-rectangles corresponding to identical sets of machines.

We can now not only find the closest range of values to those in our job ClassAd, we can specify how many and which machines will match with a job ClassAd with attribute values in that range. In addition we can present several alternative value ranges, each with a distance defined at the beginning of the section and a set of matching machines. This extra information opens the door for more complex policies for determining suggestions based on distance as well as user preferences for certain machines.

Chapter 4

ClassAd Representation of SPKI/SDSI Certificates

The ClassAd representation of SPKI/SDSI certificates is fairly simple. Each certificate ClassAd consists zero or more *cert request ports* and a *cert offer port*. A cert offer port contains attributes corresponding to the type (name or auth), issuer, identifier (name certs only), and subject of the cert. The `Subject` attribute is a literal value if the subject of the cert is directly specified using a public key, or an attribute reference if the subject is indirectly specified using a SPKI/SDSI name with one or more identifiers. In the indirect case the ClassAd also contains one or more cert request ports, each of which requests a name cert (or chain of certs) to resolve the SPKI/SDSI name. If the ClassAd represents an auth cert with the delegation bit turned on, there is an additional cert request port requesting an additional auth cert (or chain of certs) issued by the subject of the cert.

4.1 Transforming SPKI/SDSI Certificates to ClassAds

Given a name cert $C = K\ A \rightarrow S$ where K is the key of the issuer, A is the identifier being defined in K's name space, and S is the subject of the definition and $S = K_0$ $A_0\ A_1\ ...\ A_{n-1}$ we define the ClassAd Ad(C) as shown in Figure 5. Note: if S = K_0 then there is only one port. Returning to our example certs in Figure 2, Figure 6 shows the ClassAd for cert (2).

4.2 Compatibility and Composition

Given two certs C_1 and C_2, we claim that C_1 is compatible with C_2 if and only if the offer port of Ad(C_2) matches the first request port of Ad(C_1). Note that the `Requirements` expression of the first request port of Ad(C_1) is:

```
other.Type == "cert_offer" &&
other.CertType == "Name" &&
other.Issuer == "K_2" &&
other.Identifier == "A_2"
```

matching the offer port of Ad(C_2).

Given two chains Ch_1 and Ch_2 the Ad function can be extended as follows: Ad($Ch_1 \circ Ch_2$) is defined by filling the first open request port in the gang defined by Ad(Ch_1) with the gang defined by Ad(Ch_2) assuming that the two chains are compatible.

4.3 Name Resolution

Consider a name cert $C = K\ A \rightarrow S$. We wish to assemble a chain of name certs to resolve subject S (rewrite it as a key). The corresponding concept in gangmatching is marshaling a gang using Ad(C) as a root and evaluating the Subject attribute in the offer port of Ad(C) to determine the key that S resolves to.

Lemma *Given a chain of name certs Ch, the subject of Ch is equal to the value of the* `Subject` *attribute of the offer port of Ad(Ch).*

Proof: The proof is inductive.

Base Case: $S = K_0$, so S resolves to itself.
Ad(C) = Ad($K\ A \rightarrow K_0$) is shown in Figure 7 The `Subject` attribute of the offer port of Ad(C) evaluates to "K_0".

Induction: First we shall show that the existence of a chain implies the existence of a gang. Then we shall show the reverse. In both cases we assume S has $n \geq 1$ identifiers so:

$$S = K_0\ A_0\ A_1\ ...\ A_{n-1}$$

Chain → Gang: If there is a resolution for S then the following cert chains must exist:

$$Ch_0 = K_0\ A_0 \rightarrow K_1 \text{ (the composition of all of the certs in } Ch_0)$$

$$Ch_1 = K_1\ A_1 \rightarrow K_2,$$

$$...,$$

$$Ch_{n-1} = K_{n-1}\ A_{n-1} \rightarrow K_n$$

so that $Ch_n = C \circ Ch_0 \circ ... \circ Ch_{n-1} = K\ A \rightarrow K_n$ for some not necessarily distinct keys K_1, ..., K_n

Assume a gang can be marshaled for cert chains of length $< |Ch_n|$ and for $0 < i < n$ - 1 the offer port of Ad(Ch_i) is as follows:

```
[
  other = request;
  Type = "cert_offer";
  CertType = "Name";
  Issuer = "K_(i)";
  Identifier = "A_(i)";
  Subject = "K_(i+1)";
  Requirements =
    other.Type == "cert_request"
]
```

The first request port of Ad(C), labeled `chain1` can be filled by the offer port of Ad(Ch_0) since the request port `Requirements` expression is:

```
other.Type == "cert_offer" &&
other.CertType == "Name" &&
other.Issuer == "K_0" &&
other.Identifier == "A_0"
```

and the offer port of Ad(Ch_0) is:

```
[
  other = request;
  Type = "cert_offer";
  CertType = "Name";
  Issuer = "K_0";
  Identifier = "A_0";
  Subject = "K_1";
  Requirements =
    other.Type == "cert_request"
]
```

The second request port of Ad(C), labeled `chain2` can be filled by the offer port of Ad(Ch_1) since the request port `Requirements` expression is:

```
other.Type == "cert_offer" &&
other.CertType == "Name" &&
other.Issuer = chain1.Subject &&
other.Identifier == "A_1"
```

and `chain1.Subject` is bound to the `Subject` attribute of the offer port of Ad(Ch_0) with the value `"K_1"`. In the same way the request port labeled `chain`(i) can be filled by Ad(Ch_i). It follows that the `Subject` attribute of the offer port of Ad(Ch_n) is equal to the `Subject` attribute of the offer port of Ad(Ch_{n-1}) which evaluates to `"K_n"`.

Gang → Chain: If a gang G (which can be thought of as a structured set of Class-Ads) can be marshaled to satisfy Ad(C), then a sub-gang satisfying each request port of Ad(C) must exist. We shall refer to these sub-gangs as G_1, ..., G_{n-1}.

Assume that a corresponding certificate chain exists for any sub-gang of size less than $|G|$ and for $0 < i < n-1$ $\exists$ Ch_i such that the offer port of G_i = Ad(Ch_i) is:

```
[
  other = request;
  Type = "cert_offer";
  CertType = "Name";
  Issuer = "K_(i)";
  Identifier = "A_(i)";
  Subject = "K_(i+1)";
  Requirements =
    other.Type == "cert_request"
]
```

$$Ch_i = K_i\ A_i \rightarrow K_{i+1}$$

for some identifier A_i, and keys K_i and K_{i+1}.

Since the offer port of G_0 satisfies the first request port of Ad(C) C is compatible with Ch_0. Similarly since the offer port of G_i satisfies the first open request port of Ad($C \circ Ch_0 \circ ... \circ Ch_{i-1}$) the chain $C \circ Ch_0 \circ ... \circ Ch_{i-1}$) is compatible with Ch_i. By induction we can build a chain:

$$Ch_n = C \circ Ch_0 \circ ... \circ Ch_{n-1} = K\ A \rightarrow K_n$$

Note that once again the `Subject` attribute of the offer port of Ad(Ch_n) equals the `Subject` attribute of the offer port of Ad(Ch_{n-1}) which evaluates to K_n. $\square$

4.4 Authorization and Delegation

Given an auth cert $C = K\ \square \rightarrow S\ D$ where K is the key of the issuer, S is the subject of the authorization, and D is the delegation bit ($D = \square$ if on and $\blacksquare$ if off) and $S = K_0\ A_0\ A_1\ ...\ A_{n-1}$ we define Ad(C) to be the ClassAd in Figure 8.

If D is off, the last two ports are replaced by the following single port:

```
[
  other = request;
  Type = "cert_offer";
  CertType = "Auth";
  Issuer = "K";
  Subject = chain(n).Subject;
  Requirements =
    other.Type == "cert_request"
]
```

Note: if $S = K_0$ and D is off then there is only one port with its `Subject` attribute equal to K_0. Returning again to the rules in Figure 2, Figure 9 shows the ClassAd for cert(1) and Figure 10 shows the ClassAd for cert (3).

Certs $C_1 = K_1 \square \rightarrow S_1\ D_1$, $C_2 = K_2\ A_2 \rightarrow S_2$ are compatible if $S_1 = K_2\ A_2$ X for some sequence of zero or more identifiers X. As Ad(C_1) is identical to Ad(C_1) in Section 4.2 in all aspects relevant to the proof in that section we can conclude the equivalence of compatibility with matching the first open request port (not including the final request port if D is on).

Certs $C_1 = K_1 \square \rightarrow S_1\ D_1$, $C_2 = K_2 \square \rightarrow S_2\ D_2$ are compatible if $S_1 = K_2$ and D_1 is on. To show that this is equivalent to the offer port of Ad(C_2) matching the first open request port of Ad(C_1) we note the `Requirements` expression of the first request port of Ad(C_1) is:

```
other.Type == "cert_offer" &&
other.CertType == "Auth" &&
other.Issuer == "K_2"
```

matching the offer port of Ad(C_2). The extended definition of composition follows from the above.

We can now consider the equivalence of certificate chains with auth certs. A certificate chain that contains auth certs must begin with an auth cert, as name certs can only be composed with other name certs. Let $C = K \square \rightarrow S\ D$ be an auth cert. Our root ClassAd will be Ad(C). If D is off, all of the request ports of Ad(C) are identical to that of a name cert with the same subject. Therefore the equivalence of a cert chain beginning with auth cert C and a gang with Ad(C) as its root was proved in Section 4.3. If D is on, we may also assume equivalence up to the filling of the final port.

With all but the last request port filled we have an incomplete gang G and a chain Ch such that:

$$\text{Ad}(Ch) = G$$

$$S \text{ resolves to } K_n$$

$$Ch = K \square \rightarrow K_n \square$$

If we wish to view Ch as a complete gang we need only fill the final request port of G with the seed ClassAd shown in Figure 11. This seed ClassAd matches the final clause in the `Requirements` expression of the final request port of C. Note that the `Subject` attribute of the offer port of Ad(C) is equal to the `Subject` attribute of the offer port of Ad(Ch_{n-1}).Subject which evaluates to K_n.

If we wish to use the delegation option we can compose Ch with another auth cert C' = K' $\square \rightarrow S'$ D' as long as K' = K_n. This is equivalent to filling the final request port with Ad(C') as shown above. In this case the `Subject` attribute of the offer port of Ad(C) is `chain(`n`).Subject` which is equal to the `Subject` attribute of the offer port of Ad(C')

$Ch \circ C'$ = K $\square \rightarrow S'$ $\square$ (or $\blacksquare$ if D' is off) At this point we may use the same proof recursively as filling the request ports of Ad(C') is equivalent to building a chain from $Ch \circ C'$.

4.5 Certificate Chain Gangmatching

In Section 4.4 we demonstrated how to use an auth cert as the root of a gang. If we wish to used gangmatching to find a certificate chain satisfying a given authorization, the root of the gang we wish to assemble must be an authorization request. Specifically, the request is for the head of a certificate chain which must be an auth cert issued by the principal granting the authorization. Given such a principal I, Figure 12 shows the request ClassAd Ad_{Root}.

Certificate chain discovery can now be described as a gangmatching problem.

We are given an authorization request represented by ClassAd Ad_{Root}, and a set of certificates $\mathcal{C}$ = C_1, ..., C_n. Let **C** = $\{Ad_1$ = Ad(C_1), ... , Ad_n = Ad(C_n), Ad_{K_0}, ... , $Ad_{K_m}\}$ where Ad_{K_i} is the seed ClassAd described in Section 4.4 for a given key K_i. The request is satisfied if a gang can be marshaled that satisfies the constraints of the request. A certificate chain authorizing the request can be extrapolated from the gang.

Now that we have defined a ClassAd representation for SPKI/SDSI certificates and shown that a valid gang of such ClassAds is equivalent to a valid certificate chain, we must describe a suitable algorithm for assembling gangs. As stated before, a depth first search algorithm as described in [51] is insufficient because re-use of ClassAds introduces the possibility an infinite number of gangs. There is also the possibility that the algorithm may enter an infinite loop and not produce any gangs at all. As an example, consider the SPKI/SDSI certificate $K\ A \rightarrow K\ A\ A$. Since this certificate is compatible with itself, it can be applied to itself repeatedly without $K\ A$ ever being resolved. The ClassAd equivalent of this certificate matches itself, so a depth first search gangmatching algorithm would get stuck in a loop repeatedly adding the same certificate to a gang.

The solution based on push down systems (PDS) described in [29] deals with these problems by generating a finite representation of a set of found certificate chains and ignoring new chains that are equivalent to chains in the set. We can adapt this approach to gangmatching by generalizing the notion of equivalence to gangs.

```
[
  Ports = {
    [
      other = chain1;
      Type = "cert_request";
      Requirements = other.Type == "cert_offer" &&
        other.CertType == "Name" && other.Issuer == "K_0" &&
        other.Identifier == "A_0"
    ],
    [
      other = chain2;
      Type = "cert_request";
      Requirements = other.Type == "cert_offer" &&
        other.CertType == "Name" && other.Issuer == chain1.Subject
        && other.Identifier == "A_1"
    ],
   ...
    [
      other = chain(n);
      Type = "cert_request";
      Requirements = other.Type == "cert_offer" &&
        other.CertType == "Name" &&
        other.Issuer == chain(n-1).Subject &&
        other.Identifier == "A_(n-1)"
    ],
    [
      other = request;
      Type = "cert_offer";
      CertType = "Name";
      Issuer = "K";
      Identifier = "A";
      Subject = chain(n).Subject;
      Requirements = other.Type == "cert_request";
    ]
  }
]
```

Figure 5: ClassAd for generic name cert

```
[
  Ports = {
    [
      other = request;
      Type = "cert_offer";
      CertType = "Name";
      Issuer = "K_A";
      Identifier = "Bob";
      Subject = "K_B";
      Requirements =
        other.Type == "cert_request"
    ]
  }
]
```

Figure 6: ClassAd for certificate (2)

```
[
  Ports = {
    [
      other = request;
      Type = "cert_offer";
      CertType = "Name";
      Issuer = "K";
      Identifier = "A";
      Subject = "K_0";
      Requirements =
        other.Type == "cert_request"
    ]
  }
]
```

Figure 7: ClassAd equivalent to Ad($K\ A \rightarrow K_0$)

```
[
  Ports = {
    [
      other = chain1;
      Type = "cert_request";
      Requirements = other.Type == "cert_offer" &&
        other.CertType == "Name" && other.Issuer == "K_0" &&
        other.Identifier == "A_0"
    ],
    [
      other = chain2;
      Type = "cert_request";
      Requirements = other.Type == "cert_offer" &&
        other.CertType == "Name" && other.Issuer == chain1.Subject
        && other.Identifier == "A_1"
    ],
    ...
    [
      other = chain(n+1);
      Type = "cert_request";
      Requirements = other.Type == "cert_offer" &&
        other.CertType == "Auth" && other.Issuer == chain(n).Subject
    ],
    [
      other = request;
      Type = "cert_offer";
      CertType = "Auth";
      Issuer = "K";
      Subject = chain(n+1).Subject;
      Requirements = other.Type == "cert_request"
    ]
  }
]
```

Figure 8: ClassAd for generic auth cert

```
[
  Ports = {
    [
      other = chain1;
      Type = "cert_request";
      Requirements =
        other.Type == "cert_offer" &&
        other.CertType == "Name" &&
        other.Issuer == "K_A" &&
        other.Identifier == "Bob";
    ],
    [
      other = chain2;
      Type = "cert_request";
      Requirements =
        other.Type == "cert_offer" &&
        other.CertType == "Auth" &&
        other.Issuer == chain1.Subject
    ],
    [
      other = request;
      Type = "cert_offer";
      CertType = "Auth";
      Issuer = "X";
      Subject = chain2.Subject;
      Requirements =
        other.Type == "cert_request"
    ]
  }
]
```

Figure 9: The ClassAd for cert(1)

```
[
  Ports = {
    [
      other = chain1;
      Type = "cert_request";
      Requirements =
        other.Type == "cert_offer" &&
        other.CertType == "Name" &&
        other.Issuer == "K_B" &&
        other.Identifier == "Carol";
    ],
    [
      other = request;
      Type = "cert_offer";
      CertType = "Auth";
      Issuer = "K_B";
      Subject = chain1.Subject;
      Requirements =
        other.Type == "cert_request"
    ]
  }
]
```

Figure 10: The ClassAd for cert (3)

```
[
  Ports = {
    [
      other = request;
      Type = "cert_offer";
      CertType = "Auth";
      Issuer = "K_(n)";
      Subject = "K_(n)";
      Requirements =
        other.Type == "cert_request";
    ]
  }
]
```

Figure 11: A generic seed ClassAd

```
[
  Ports = {
    [
      other = chain;
      Requirements =
        chain.CertType == "Auth" &&
        chain.Issuer == "I";
    ]
  }
]
```

Figure 12: The request ClassAd Ad_{Root}

Chapter 5

Gangmatching: Structures, Concepts and Algorithms

The gangmatching algorithm requires more formal definitions of the concepts and structures involved. In this chapter we will formally define the static and dynamic structures. Next, we formally define the concepts of equivalence, partial evaluation, and validity. We then present the algorithm, along with proofs of correctness and a complexity analysis. Finally we describe problems that arise from gangmatching, and propose solutions using ClassAd analysis techniques.

The primary structures involved in gangmatching are *ClassAds*, *ports*, *gangs* and *gangsters*. A gangmatching ClassAd is made up of a set of ports, each of which represents a request for another ClassAd. A port that has not been matched is called an *open* port. A gang is a set of ClassAds that match one another's ports. If all ports are satisfied, the gang is *complete*. A gangster is an intermediate structure that represents an open port in an incomplete gang along with a set of *bindings* that assign values to attribute references and express dependencies between other ports and the gangster's port. Ports, gangs, and gangsters will be defined more formally in Section 5.1

The key concepts used by the algorithm are *equivalence*, *partial evaluation* and *validity*. Two gangsters are equivalent if they are structurally the same, but contain attributes from different ClassAds. Partial evaluation is used for expressions in incomplete gangs, where not all of the attribute references are bound to literal values. These partially evaluated expressions must be satisfied by bindings generated by subsequent matches in order for the resulting complete gang to be *valid*. We will define equivalence, partial evaluation, and validity more precisely in Section 5.2.

The gangmatching algorithm is based on the structures and concepts outlined above. The input to the algorithm is a root ClassAd C_0 with one port, and a set of ClassAds $\mathcal{C}$. Beginning with the gangster composed of the single port of C_0, the algorithm creates new gangsters by matching existing gangsters to parent ports of other ClassAds. Whenever a new gangster is created, a new rule in a regular grammar is generated. When the algorithm terminates, this grammar generates all complete valid gangs built from C_0 and the ClassAds in $\mathcal{C}$. In order to avoid repeated work and, more importantly, infinite loops, the algorithm tests if new gangsters are equivalent to previously encountered gangsters. If an equivalent gangster is found, the algorithm adds a new rule to the grammar, but does not attempt to match the new gangster. Otherwise, the new gangster is tested against the parent port of each ClassAd in C_0 for a potential match. If the match is conditionally valid, the constraint formulas are partially evaluated, and the resulting expression is passed to the first new gangster created by the match. Further matches must satisfy this expression in addition to the appropriate constraint formulas. The algorithm is presented and explained in more detail in Section 5.3.

5.1 Gangmatching Structures

The input for the gangmatching algorithm consists of a set of ClassAds. Each ClassAd consists of an ordered list of ports. A *port* consists of a set of *attribute definitions* and a *constraint formula*. The attributes defined in a port are also exported by the port to potential matches, and are referred to as *exported attributes*. The attributes referred to in the definitions and constraint formula of a port P may be imported via P or via a port preceding P in the same ClassAd. These attributes are referred to as *imported attributes*. For example in Figure 13, there are two ports: one to request a workstation to run a simulation on, and one to request a license to use the software needed to run the simulation. The attribute definitions in the first port define the exported attributes `Type` and `ImageSize`. The constraint formula in the first port is the attribute expression of the `Requirements` attribute and contains constraints on the imported attributes `Type`, `Arch`, `OpSys` and `Memory`. We know that these are imported attributes because they are in the form `other.attr`. Note also the reference to the exported attribute `ImageSize` in the constraint formula. The second port is similar to the first, with a notable exception that it contains a reference to an attribute (`Name`) imported from the first port. We know that this attribute is imported from the first port because it is in the form `cpu.attr` where `cpu` is the label assigned to the port of the ClassAd that matches the first port, as indicated by the definition `other = cpu`. ClassAds and ports make up the static structures used in the gangmatching process and are defined formally in Section 5.1.1.

```
[
  Ports = {
     [ // request a workstation
       other = cpu;
       Type = "cpu_request";
       ImageSize = 28M;
       Requirements =
         other.Type == "Machine" &&
         other.Arch == "INTEL" &&
         other.OpSys == "LINUX" &&
         other.Memory >= ImageSize
     ],
     [ // request a license
       other = license;
       Type = "license_request";
       CPUName = cpu.Name;
       Cmd = "run_sim";
       Requirements =
         other.Type == "License" &&
         other.App == Cmd
     ]
   }
]
```

Figure 13: A gangmatching ClassAd for a job

The output for the gangmatching algorithm consists of a set of gangs. A gang represents a tree of ClassAds where each ClassAd is connected to its parent or child through one of its ports. The ClassAd at the root of the tree is referred to as the *root ClassAd*, ClassAds with more than one port that are not the root ClassAd are *intermediate ClassAds*, and ClassAds with only one port that are not the root are *leaf ClassAds*. A port connecting a ClassAd C to one of its children is called a *child port*, and the port connecting C to its parent is the *parent port*. Given two ClassAds C and C' where C is the parent of C', the connection between the child port of C

corresponding to C' and the parent port of C' is called a *match*. A gang is *complete* if it contains a root ClassAd with no parent port and every ClassAd in the gang has a child for each of its child ports. If any ClassAd in a gang has an unmatched port the gang is *incomplete*.

The process for creating gangs involves building incomplete gangs from ClassAds and building complete gangs from incomplete gangs. In order to prevent the construction of infinitely large gangs, we need a way to indicate when an incomplete gang is equivalent to a previously encountered incomplete gang. An incomplete gang can be thought of as a set of unmatched or *open* ports. An open port in an incomplete gang also has a corresponding set of assignments or *bindings* of attributes imported via matched ports in the gang to literal values. Conversely, each open port has a set of imported attributes that other open ports contain references to. The set of bindings between attributes from other ports and attributes in an open port, as distinguished from the bindings of attributes to literal values, are referred to as *links*. We refer to an open port together with a set of bindings and a set of links as a *gangster*. Gangs and gangsters make up the dynamic structures used in the gangmatching process and are defined formally in Section 5.1.2.

5.1.1 Static Structures

Before we formally define what a port is, we must first define the atoms from which a port is constructed. Attributes describe an advertisement and are exported by a port or imported via a port. An attribute definition consists of a name and an expression.

For the purpose of this chapter we restrict attribute expressions to attribute references imported via other ports or literal values. Let $\mathcal{V}$ denote the set of all literal values, $\mathcal{E}$ the set of all exported attributes, $\mathcal{I}$ the set of all imported attributes, and let $\mathcal{U} = \mathcal{I} \cup \mathcal{E} \cup \mathcal{V}$. A set of attribute definitions is represented by a total function $\delta : \mathcal{E} \rightarrow (\mathcal{I} \cup \mathcal{V})$. We shall refer to this function as a *definition function*.

We now define a predicate logic for expressing constraints on attributes. Let $\mathcal{B} = \{\mathbf{T}, \mathbf{F}, \mathbf{U}, \mathbf{E}\}$ be a set of literal values in four valued ClassAd Boolean logic. The Boolean operators $\wedge$, $\vee$, and $\neg$ are well defined over values in $\mathcal{B}$. An n-ary predicate is a function $\kappa : \mathcal{U}^n \rightarrow \mathcal{B}$. We define Φ as the set of all Boolean formulas over predicates in $\mathcal{K}$ and values in $\mathcal{B}$. Formally we define Φ as follows:

- if $b \in \mathcal{B}$ then $b \in \Phi$
- if κ is an n-ary predicate and $\vec{u} \in \mathcal{U}^n$ then $(\kappa, \vec{u}) \in \Phi$
- if $\phi, \psi \in \Phi$ then $\phi \wedge \psi, \phi \vee \psi, \neg\phi \in \Phi$

Henceforth, we shall refer to elements of Φ as constraint formulas.

We can now formally define a port P as a 5-tuple $(E_P, I_P, J_P, \delta_P, \phi_P)$ where $E_P \subseteq \mathcal{E}$ is the set of all attributes exported by P, $I_P \subseteq \mathcal{I}$ is the set of all attributes imported via P, $J_P \subseteq \mathcal{I}$ is the set of all attributes referenced in P that are imported via other ports, $\delta_P : E_P \rightarrow (J_P \cup \mathcal{V})$ is a function representing the attribute definitions in P, $\phi_P \in \Phi$ is a constraint formula over I_P, J_P, and $\mathcal{V}$ representing the match requirements of P. A ClassAd $C_i = \{P_0^i, \ldots, P_{n-1}^i\}$ is an ordered list of ports. The cardinality of C_i is expressed as $|C_i|$ and is equal to n, the number of ports in C_i. An attribute named $ImpAttr$ imported by port P_j^i is abbreviated $ImpAttr_j^i$, and an attribute named $ExpAttr$ exported by port P_j^i is abbreviated $ExpAttr_{j(e)}^i$. The superscript and subscript are omitted when the name of the attribute and its designation as imported or exported are unambiguous.

Let C_0 be the ClassAd in figure 13. C_0 consists of two ports, P_0^0 and P_1^0.

$$E_{P_0^0} = \{Type, ImageSize\}$$

$$I_{P_0^0} = \{Type, Arch, OpSys, Memory, Name\}$$

$$J_{P_0^0} = \varnothing$$

$$\delta_{P_0^0} = \{Type \mapsto \texttt{"cpu\_request"}, ImageSize \mapsto \texttt{28M}\}$$

$$\phi_{P_0^0} = (Type == \texttt{"Machine"}) \wedge (Arch == \texttt{"INTEL"}) \wedge$$
$$(OpSys == \texttt{"LINUX"}) \wedge (Memory >= ImageSize)$$

$$E_{P_1^0} = \{Type, CPUName, Cmd\}$$

$$I_{P_1^0} = \{Type, App\}$$

$$J_{P_1^0} = \{Name\}$$

$$\delta_{P_1^0} = \{Type \mapsto \texttt{"license\_request"}, CPUName \mapsto Name,$$
$$Cmd \mapsto \texttt{"run\_sim"})\}$$

$$\phi_{P_1^0} = (Type == \texttt{"License"}) \wedge (App == Cmd)$$

5.1.2 Dynamic Structures

Given a set $I \subseteq \mathcal{I}$ of imported attributes we define a *binding function* as a partial function $\beta : I \to \mathcal{V}$. A binding function is analogous in form to a definition function, but in the process of gangmatching, definitions are static while bindings are dynamically created. A set of links is represented by a *binding relation* $L \subseteq \mathcal{I} \times \mathcal{I}$ that associates one set of imported attributes with another.

A gangster G is an intermediate structure in the gangmatching process that represents an open port in an incomplete gang. We define G as a triple (P, β, L) where P =(E, I, J, δ, ϕ) is a port, $\beta : J \rightarrow \mathcal{V}$ is a binding function and $L \subseteq \mathcal{I} \times I$ is a binding relation. β binds the attributes imported via prior ports and referenced in P to literal values. L associates imported attributes from elsewhere in a gang with attributes imported via P on which they depend. The set $D = J \setminus \{i \mid (i, v) \in \beta$ for some $v \in \mathcal{V}\}$ represents the dependencies of G. G is independent if D is empty, otherwise G is dependent.

Returning to the example ClassAd in Figure 13, let G_0 be the gangster for P^0_0, and G_1 be the gangster for P^0_1.

$$G_0 = (P^0_0, \varnothing, \varnothing),\ P^0_0 = (E^0_0, I^0_0, \varnothing, \delta^0_0, \phi^0_0)$$

$$G_1 = (P^0_1, \varnothing, \varnothing),\ P^0_1 = (E^0_1, I^0_1, \{Name^0_0\}, \delta^0_1, \phi^0_1)$$

Note that G_0 is independent since J^0_0 is empty, but G_1 is dependent since J^0_1 is nonempty, so $D = J^0_1 \setminus \varnothing = \{Name^0_0\}$. If we find a match for G_0 that defines the attribute `Name`, we can create a new gangster G_2 from G_1:

$$G_2 = (\ P^0_1, \{Name^0_0 \mapsto \texttt{"paneer.cs.wisc.edu"}\})$$

Now that we have a binding for $Name^0_0$ in β_{G_2}, G_2 is independent.

```
[
  Ports = {
    [
      other = cpu1;
      Type = "cpu_request";
      Requirements =
        other.Type == "Machine" &&
        other.IsDedicated;
    ],
    [
      other = cpu2;
      Type = "cpu_request";
      Requirements =
        other.Type == "Machine" &&
        other.IsDedicated &&
        other.Host == cpu1.Host
    ],
    [
      other = request;
      Type = "Machine";
      Arch = cpu1.Arch;
      OpSys = cpu1.Opsys;
      Memory = cpu1.Memory;
      Name = cpu1.Name;
      Requirements = other.Type == "cpu_request"
    ]
  }
]
```

Figure 14: A gangmatching ClassAd for a multi-processor

Let C_1 be the ClassAd in Figure 14. Suppose that we have matched the first port of C_0 (P^0_0) with the third port of C_1 (P^1_2). This means that the imported attribute `cpu.Name` ($Name^0_0$) in C_0 is bound to the exported attribute $Name^1_{2(e)}$ in C_1, that in turn is bound to the imported attribute `cpu1.Name` ($Name^1_0$). When we create the gangster G_3 for the first open port of the gang created when C_0 is matched with C_1 it must contain this binding in its binding relation:

$$G_3 = (P_0^1, \varnothing, \{(Name_0^0, Name_0^1)\})$$

This way, when G_3 is matched and $Name_0^1$ is bound to a literal value, $Name_0^0$ can also be bound to that value.

A match M is a pair (G, P) where $G = (P_G, \beta_G, L_G)$ is an independent gangster and $P = (E_P, I_P, J_P, \delta_P, \phi_P)$ is a port. A match defines two binding functions $\beta_{G \to P} : I_{P_G} \to E_P$ and $\beta_{P \to G} : I_P \to E_{P_G}$ that bind the imported attributes in P_G to the attributes of the same name exported by P and the imported attributes in P to the attributes of the same name exported by P_G respectively.

A gang $\Gamma = (C_0, \cdots, C_{n-1})$ is an ordered list of ClassAds. This list represents a tree of ClassAds where the parent port of each non-root ClassAd is matched with the first open child port of the gang constructed from the preceding ClassAds. We define the size of Γ as the number of ClassAds (n). Γ is complete iff $\sum_{i=0}^{n-1} |C_i| = 2(n-1)$, i.e. the total number of ports in the gang is twice the total number of non-root ClassAds. $\Gamma_0 = (C_0, C_1)$ is not complete since it contains five ports and only one match.

5.2 Gangmatching Concepts

The principal concepts needed for the gangmatching algorithm and gangmatching analysis are analogous to similar concepts used in the lambda calculus. Every gangster has a *signature* that can be derived by applying the bindings to the occurrences of imported attributes in the port's attribute definitions and constraint formula. Two gangsters are *equivalent* if they have identical signatures. The function used to translate one gangster to an equivalent gangster is analogous to alpha reduction of lambda expressions. We defined a match in Section 5.1.2 as a pairing of a gangster G in an incomplete gang with the parent port P of a ClassAd C. A match is *valid* if G satisfies P's constraint formula and P satisfies the constraint formula of G's port. A gang is valid if all of its matches are valid. It may be the case that the constraint formulas contain imported attribute references that are still unbound. We call such a match *conditionally valid*. This scenario can be dealt with by partially evaluating the constraint formulas, and using the bindings created by subsequent matches to satisfy the resulting formula. This use of partial evaluation corresponds to beta reduction of lambda expressions.

5.2.1 Equivalence

A signature S is a 6-tuple $(E,\ I,\ J,\ \delta,\ L,\ \phi)$ where $E \subseteq \mathcal{E}$ is a set of exported attributes, I and $J \subseteq \mathcal{I}$ are sets of imported attributes, $\delta : E \to J \cup \mathcal{V}$ is a definition function, ϕ is a constraint formula over elements of I, J and $\mathcal{V}$, and $L \subseteq \mathcal{I} \times I$ is a binding relation. Let $\mathcal{G}$ be the set of all possible gangsters and $\mathcal{S}$ be the set of all possible signatures. We define a function $\Sigma : \mathcal{G} \to \mathcal{S}$ as follows. Given a gangster G = $(P_G, \beta_G, L_G) \in \mathcal{G}$, $\Sigma(G)$ = $(E_{P_G}, I_{P_G}, \delta, L_G, \phi)$ where δ = $\delta_{P_G} \circ \beta_G$ and ϕ = $\phi_{P_G} \circ \delta_{P_G} \circ \beta_G$.

Returning to our example:

$$\Sigma(G_0) = E_{P_0^0}, I_{P_0^0},$$
$$\{(Type, \texttt{"cpu\_request"}), (ImageSize, \texttt{28M})\},$$
$$\varnothing,$$
$$(Type, == \texttt{"Machine"}) \wedge (Arch\ (== \texttt{"INTEL"}) \wedge$$
$$(OpSys\ (== \texttt{"LINUX"}) \wedge (Memory\ (== \texttt{28M}))$$

$$\Sigma(G_1) = E_{P_1^0}, I_{P_1^0},$$
$$\{(Type, \texttt{"license\_request"}), (CPUName, Name),$$
$$(Cmd, \texttt{"run\_sim"})\},$$
$$\varnothing,$$
$$(Type == \texttt{"Machine"}) \wedge (App == \texttt{"run\_sim"})$$

$$\Sigma(G_2) = (E_{P_1^0}, I_{P_1^0},$$
$$\{(Type, \texttt{"license\_request"}), (CPUName, \texttt{"paneer.cs.wisc.edu"}),$$
$$(Cmd, \texttt{"run\_sim"})\},$$
$$\varnothing,$$
$$(Type == \texttt{"Machine"}) \wedge (App == \texttt{"run\_sim"}))$$

$$\Sigma(G_3) = (E_{P_0^1}, I_{P_0^1},$$
$$\{(Type, \texttt{"cpu\_request"})\},$$
$$\{(Name_0^0, Name)\},$$
$$Type == \texttt{"Machine"}) \wedge IsDedicated)$$

Two gangsters G and G' are equivalent ($G \equiv G'$) iff $S = \Sigma(G)$, $S' = \Sigma(G')$ and there exists a bijection $\alpha : E_S \cup I_S \cup J_S \to E_{S'} \cup I_{S'} \cup J_{S'}$ such that:

- for each $attr \in E_S$ $\alpha(attr) = attr' \in E_{S'}$ for some P'
- for each $attr \in I_S$ $\alpha(attr) = attr' \in I_{S'}$ for some P'
- for each $attr \in J_S$ $\alpha(attr) = attr' \in J_{S'}$ for some P'
- $\delta_S \circ \alpha = \delta_{S'}$ and $\delta_{S'} \circ \alpha^{-1} = \delta_{S'}$
- $\phi_S \circ \alpha = \phi_{S'}$ and$\phi_{S'} \circ \alpha^{-1} = \phi_S$
- $L_S \circ \alpha = L_{S'}$ and $L_{S'} \circ \alpha^{-1} = L_S$

5.2.2 Partial Evaluation and Validity

We define the partial evaluation function $B : \Phi \to \Phi$ as follows:

- if $b \in \mathcal{B}$, then $B(b) = b$.
- if κ is an n-ary predicate, $\vec{u} \in \mathcal{U}^n$ then $B(\kappa, \vec{u}) = (\kappa, \vec{u})$ if $\kappa(\vec{u}) = \mathbf{U}$ (the ClassAd boolean value `undefined`), otherwise $B(\kappa, \vec{u}) = \kappa(\vec{u})$.
- if $\phi \in \Phi$ and $B(\phi) \in \mathbf{B}$ then $B(\neg, \phi) = \neg\, B(\phi)$, otherwise $B(\neg, \phi) = (\neg, \phi)$.
- if $op \in \{\vee, \wedge\}$ and $\phi, \psi \in \Phi$ then $B(\phi\ op\ \psi)$ = the value of $B(\phi)\ op\ B(\psi)$ if both $B(\phi)$ and $B(\psi)$ are in $\mathcal{B}$ or one of them is in $\mathcal{B}$ and applying op results in the same value regardless of the value of the other. Otherwise $B(\phi\ op\ \psi) = B(\phi)\ op\ B(\psi)$.

Given match M = (G, P), let ψ_M =

$$B((\phi_{P_G} \circ \delta_{P_G} \circ \beta_G \circ \beta_{G\to P} \circ \delta_P) \wedge (\phi_P \circ \delta_P \circ \beta_{P\to G} \circ \delta_{P_G} \circ \beta_G)).$$

If ψ_M = **T**, then M is *valid*. If $\psi_M \in \{$**F**, **U**,**E**$\}$ then M is *invalid*. If $\psi_M \notin$ **B** then M is *conditionally valid*, i.e. a binding function β must be supplied such that $B(\psi_M \circ \beta)$ = **T** in order for M to be a valid match. We can construct β from binding functions created by other matches. Assuming we have a satisfactory β, the binding function created by M is $\beta_M = \beta_{G\to P} \circ \delta_P \circ \beta$.

For example, let M_1 be the match (G_0, P_2^1):

$\psi_{M_1} = B((\phi_{P_0^0} \circ \delta_{P_0^0} \circ \beta_{G_0} \circ \beta_{G_0\to P_2^1} \circ \delta_{P_2^1}) \wedge (\phi_{P_2^1} \circ \beta_{P_2^1\to G_0} \circ \delta_{P_0^0} \circ \beta_{G_0}))$

= $B(((Type_0^0 ==$ `"Machine"`$) \wedge$
$(Arch_0^0 ==$ `"INTEL"`$) \wedge (OpSys_0^0 ==$ `"LINUX"`$) \wedge$
$(Memory_0^0 == ImageSize_0^0)$
$\circ\, \delta_{P_0^0} \circ \beta_{G_0} \circ \beta_{G_0\to P_2^1} \circ \delta_{P_2^1}) \wedge$
$((Type_2^1 ==$ `"cpu_request"`$) \circ \beta_{P\to G} \circ \delta_{P_G} \circ \beta_G))$

= $B(((Type_0^0 ==$ `"Machine"`$) \wedge$
$(Arch_0^0 ==$ `"INTEL"`$) \wedge (OpSys_0^0 ==$ `"LINUX"`$) \wedge$
$(Memory_0^0 ==$ `28M`$)$
$\circ\, \beta_{G_0} \circ \beta_{G_0\to P_2^1} \circ \delta_{P_2^1}) \wedge$
$((Type_{0(e)}^0 ==$ `"cpu_request"`$) \circ \delta_{P_G} \circ \beta_G))$

= $B(((Type^0_0$ == `"Machine"`$) \wedge$

$(Arch^0_0$ == `"INTEL"`$) \wedge (OpSys^0_0$ == `"LINUX"`$) \wedge$

$(Memory^0_0$ == `28M`)

$\circ\, \beta_{G_0 \rightarrow P^1_2} \circ \delta_{P^1_2}) \wedge$

((`"cpu_request"` == `"cpu_request"`) $\circ\, \beta_G$))

= $B(((Type^1_{2(e)}$ == `"Machine"`$) \wedge (Arch^1_{2(e)}$ == `"INTEL"`$) \wedge$

$(OpSys^1_{2(e)}$ == `"LINUX"`$) \wedge (Memory^1_{2(e)}$ == `28M`)

$\circ\, \delta_{P^1_2}) \wedge$

(`"cpu_request"` == `"cpu_request"`))

= B((`"Machine"` == `"Machine"`) $\wedge$

$(Arch^1_0$ == `"INTEL"`$) \wedge (OpSys^1_0$ == `"LINUX"`$) \wedge$

$(Memory^1_0$ == `28M`) $\wedge$ (`"cpu_request"` == `"cpu_request"`))

= $(Arch^1_0$ == `"INTEL"`$) \wedge (OpSys^1_0$ == `"LINUX"`$) \wedge$

$(Memory^1_0$ == `28M`)

Note that ψ_{M_1} is a constraint formula over attributes imported via P^1_0, so M_1 is conditionally valid.

A gang Γ = (C_0, …, C_{n-1}) is valid iff the order of C_i's defines a breadth first ordering of a tree of ClassAds where each child C_j of C_i corresponds a port P in C_i

(P is called a child port of C_i) and there is one port P' in each C_j corresponding to P (P' is called the parent port of C_j) and for each M = (G, P) used to construct the gang:

- P is the parent port of C_i
- P_G is a child port of the parent ClassAd of C_i
- M is valid or M is conditionally valid and for $\beta = \cup \{\beta_{M'} \mid C_j$ is a child of $C_i\}$ $B(\psi_M \circ \beta) = \mathbf{T}$, where M' is the match between a child port of C_i with C_j.

5.3 Gangmatching Algorithm

The gangmatching algorithm builds individual gangs in a top-down (root to leaves) fashion. The premise of the algorithm is that if an infinite number of gangs can be composed from a finite set of ClassAds, then there must be a repeating pattern – in the same way that a finite automaton can define an infinite but regular language. These repetitions can be prevented by detecting new gangsters that are equivalent to previously encountered gangsters. Thus, we can assemble a finite grammar that may produce an infinite number of gangs. In addition, this algorithm makes use of the partial evaluation facility described in Section 5.2.2 to build gangs that satisfy conditionally valid matches.

The algorithm takes as input a set $\mathcal{C}$ of ClassAds, and a root ClassAd C_0. Without loss of generality we will assume C_0 has only one port. We also assume that each ClassAd $C \in \mathcal{C} \cup \{C_0\}$ satisfies the following properties:

- The requirements expression ϕ_P of each port P of C consists of a conjunction of binary or unary predicates over attributes imported via P (I_P), attributes imported via previous ports in C (J_P) and literal values ($\mathcal{V}$) in which no predicate contains attributes imported from more than one previous port in C and every predicate contains at least one attribute imported via P.
- The last port in C is the parent port of C, and all other ports are child ports.
- C has no more than 2 child ports.

The primary data structures in the algorithm are:

- $SEEN$ - a set of previously encountered gangsters indexed by signature
- ALT - a set of equivalence classes of gangsters indexed by signature
- $RULES$ - a set of rules for a regular grammar that generates all possible gangs
- $NEXT$ - a mapping from gangsters to gangsters to assure that the grammar generates each gang in the correct order

In order to facilitate the handling of conditionally valid matches we will add an additional component ψ_G to each gang G. The purpose of ψ_G will become clear as we discuss the algorithm.

The GANGMATCH method shown in Figure 15 adds a gangster created from the single port of C_0. The algorithm then enters a loop in which gangsters are removed and added to a list of gangs using the ADDGANGSTER and REMOVEGANGSTER methods. At the beginning of each loop, an independent gangster is added to $SEEN$

GANGMATCH($C_0, \mathcal{C}$)
$P \leftarrow C_0$'s port
$G \leftarrow$ ADDGANGSTER(P,$\varnothing$,$\varnothing$,$\mathbf{T}$)
$RULES \leftarrow \{(G \rightarrow C_0)\}$
while $G \leftarrow$ REMOVEGANGSTER()
  if $SEEN[\Sigma(G)]$
    $ALT[\Sigma(G)] \leftarrow ALT[\Sigma(G)] \cup \{G\}$
    for each $(G' \rightarrow G''\ C') \in RULES$ where $G'' \equiv G$
      $RULES \leftarrow RULES \cup (G' \rightarrow G\ C')$
  else $SEEN[\Sigma(G)] \leftarrow$ **true**
    for each $C \in \mathcal{C}$
      $\psi \leftarrow$ MATCH(G, C)
      if $\psi \notin \{\mathbf{F},\mathbf{U},\mathbf{E}\}$
        GENERATENEWGANGSTERS(G, C, ψ)

MATCH(G, C)
$P \leftarrow C$'s parent port
$\delta_G \leftarrow \delta_{P_G} \circ \beta_G$
$\phi_G \leftarrow (\phi_{P_G} \circ \beta_G) \wedge \psi_G$
$\psi \leftarrow B((\phi_G \circ \beta_{G \rightarrow P} \circ \delta_P) \wedge (\phi_P \circ \beta_{P \rightarrow G} \circ \delta_G))$
return ψ

Figure 15: The GANGMATCH algorithm

if there is no equivalent gangster already in $SEEN$ or added to the appropriate equivalence class in ALT otherwise. If G is not equivalent to a gangster in $SEEN$, the parent port P of each ClassAd $C \in \mathcal{C}$ is tested to see if it matches G. The MATCH method takes G and P as input and returns an element of $\mathcal{B}$ for valid or invalid matches or a constraint formula ψ for conditionally valid matches. If the match M = (G, P) is valid or conditionally valid the method GENERATENEWGANGSTERS is called. When the GANGMATCH method has completed, $RULES$ will produce a set of matches representing all complete valid gangs rooted at C_0. Each gang is a list of ClassAds in order of appearance in the gang, with the parent port of each ClassAd matching the first open port of the gang made up of the previous ClassAds.

GENERATENEWGANGSTERS(G, C, ψ)
$P \leftarrow C$'s parent port; $L_M \leftarrow \varnothing$;
for each $attr_{(e)} \mapsto Y \in \delta_P$
    if $(X, attr) \in L_G$
        $L_M \leftarrow L_M \cup \{(X, Y)\}$
    $L_M \leftarrow L_M \cup \{(attr, Y)\}$
$G_{last} \leftarrow$ **null**
for each child port P' of C
    $L \leftarrow \{(X, Y) \in L_M \mid Y \in I_{P'}\}$
    $\psi' \leftarrow \wedge$ {predicates in ψ containing an $i \in I_{P'}$}
    $G_{new} \leftarrow$ ADDGANGSTER(P', $\varnothing$, L, ψ')
    if G_{last} = **null**
        $RULES \leftarrow RULES \cup \{(G_{new} \rightarrow G\ C)\}$
    else $NEXT[G_{last}] \leftarrow G_{new}$
    $G_{last} \leftarrow G_{new}$
$G' \leftarrow NEXT[G]$
if $G' \neq$ **null**
    $\beta \leftarrow \{(X, Y) \in L_M \mid X \in J_{P_{G'}}, Y \in \mathcal{V}\}$
    $G_{new} \leftarrow$ ADDGANGSTER($P_{G'}$, β, $L_{G'}$, $\psi_{G'}$)
    if G_{last} = **null**
        $RULES \leftarrow RULES \cup \{(G_{new} \rightarrow G\ C)\}$
    else $NEXT[G_{last}] \leftarrow G_{new}$
    $NEXT[G_{new}] \leftarrow NEXT[G']$
elsif $G_{last} \neq$ **null**
    $NEXT[G_{last}] \leftarrow$ **null**
else $RULES \leftarrow RULES \cup \{(S \rightarrow G\ C)\}$

Figure 16: The GENERATENEWGANGSTERS method

The GENERATENEWGANGSTERS method shown in Figure 16 creates new gangsters from the child ports of a ClassAd C whose parent port P has just been matched with gangster G. It keeps track of the correct order of gangsters using the $NEXT$ mapping. It then creates a new gangster from $NEXT[G]$, and the bindings created by matching G with P. If this method produces no new gangsters, then a gang has been completed and $(S \rightarrow G\ C)$ is added to $RULES$.

The first part of the GENERATENEWGANGSTERS method uses the binding relation L_G, the binding function $\beta_{G\rightarrow P}$, and the definition function δ_P to construct new bindings. Recall that $L_G \subseteq \mathcal{I} \times I_{P_G}$ binds attributes imported via other ports in an incomplete gang to the attributes imported via P_G, $\beta_{G\rightarrow P} : I_{P_G} \rightarrow E_P$ binds each attribute imported via P_G to an attribute of the same name exported by P , and δ_P: $E_P \rightarrow (\mathcal{V} \cup J_P)$ defines each attribute exported by P as a literal value or an attribute imported from port preceding P in the same ClassAd (C). If we compose $\beta_{G\rightarrow P}$ with δ_P we get a new binding function $\beta_M : I_{P_G} \rightarrow (\mathcal{V} \cup \mathcal{J}_{\mathcal{P}})$. Furthermore we can apply β_M to the bindings in L_G to get an additional set of bindings $L_M \subseteq \mathcal{I} \times (\mathcal{V} \cup J_P)$. The set $\beta_M \cup L_M$ contains all of the bindings that ensue from matching G with P. Let $\beta = (\beta_M \cup L_M) \cap (\mathcal{I} \times \mathcal{V})$, and $L = (\beta_M \cup L_M) \cap (\mathcal{I} \times J_P)$.

The second part of the GENERATENEWGANGSTERS method creates new gangsters from ports in C and bindings in L_M. Given a port $P' \in C$, the bindings relation $L = L_M \cap (\mathcal{I} \times J_{P'})$ can be used to create a new gangster $G_{new} = (P', \varnothing, L, \psi')$.

The third part of the GENERATENEWGANGSTERS method creates a new gangster by applying bindings in β to $NEXT[G]$. Given a gangster $G' = NEXT[G]$, the bindings in $\beta \cap (J_{P'} \times \mathcal{V})$ can be used to create a binding function for the new gangster $G_{new} = (P_{G'}, \beta_G \cup \beta, L_{G'}, \psi_{G'})$.

To demonstrate this algorithm we return to the ClassAd representation of SPKI/SDSI certificates discussed in Chapter 4. Given the following certificates expressed as rewrite rules:

(1) $X \square \rightarrow K_A$ Bob $\square$

(2) K_A Bob $\rightarrow K_B$

(3) $K_B \square \rightarrow K_B$ Carol $\blacksquare$

(4) K_B Carol $\rightarrow K_C$

the ClassAd representations of these certificates ($C_1, \cdots, C_4$) are shown in Figures 17 and 18. The ClassAd (C_0) representing a request for authorization for principal K_C to access resource X is shown in Figure 19. The following table shows an example run of the gangmatching algorithm.

Match	**Gangsters**	**Rules**
	$G_0 = (P_0^0, \varnothing, \varnothing, \mathbf{T})$	$G_0 \rightarrow C_0$
(G_0, C_1)	$G_1 = (P_0^1, \varnothing, \varnothing, \mathbf{T})$ $G_2 = (P_1^1, \varnothing, \{(Subject_0^0, Subject_1^1)\},$ $(Subject_1^1 ==$ `"K_C"`$))$	$G_1 \rightarrow G_0\ C_1$
(G_1, C_2)	$G_3 = (P_1^1, \{(Subject_0^1,$ `"K_B"`$)\},$ $\{(Subject_0^0, Subject_1^1)\},$ $(Subject_1^1 ==$ `"K_C"`$))$	$G_3 \rightarrow G_1\ C_2$
(G_3, C_3)	$G_4 = (P_0^3, \varnothing, \{(Subject_0^0, Subject_0^3)\},$ $(Subject_0^3 ==$ `"K_C"`$))$	$G_4 \rightarrow G_3\ C_3$
(G_4, C_4)		$S \rightarrow G_4\ C_4$

The gang generated by the grammar is (C_0, C_1, C_2, C_3, C_4).

5.3.1 Correctness

Termination: As defined in Section 5.1.1 an attribute expression in a port can either be an attribute reference or a literal value. Therefore, the possible attribute values in a port are limited to the literal values in $\mathcal{V}$. Therefore there are finitely many variations of a given port of a given ClassAd. Since there are finitely many ClassAds, there are finitely many gangsters that can be added to $SEEN$. If a gangster has an equivalent gangster in $SEEN$ it is skipped and no new gangsters are generated during that iteration of the while loop. Since the only gangsters that are considered are gangsters that have no equivalent in $SEEN$ and there are only finitely many possible gangs that can be added to $SEEN$, the while loop must eventually terminate. $\square$

Soundness: Every gang output by the algorithm is a collection of valid (or conditionally valid) matches between a valid gangster and a parent port of a constituent ClassAd. In the case of a conditionally valid match between a gangster G and a ClassAd C with parent port P, parts of the resulting formula ψ are attached to gangsters, based on which imported attributes occur in ψ. Line 9 of the GENERATENEWGANGSTERS method in Figure 16 shows that predicates in ψ are parceled out to a port P' if they contain an attribute in $I_{P'}$. In order for the algorithm to be correct, the only imported attributes occurring in ψ must be elements of $J_P \subseteq \cup \{I_{P'} \mid P'$ is a child port of $C\}$, the attributes referenced in P imported from prior ports in C. We shall prove this inductively.

The formula ψ is generated from three sources: ϕ_{P_G}, ϕ_P, and ψ_G. We can assume that ψ_G either contains no imported attributes (the base case) or that ψ_G contains only elements of I_{P_G} (which serves as the inductive hypothesis). Applying the match function $\beta_{G \to P}$ to ψ_G takes all elements of I_{P_G} to the corresponding elements of E_P. The definition function of P, δ_P, takes all elements of E_P to elements of J_P or literal values in $\mathcal{V}$.

The constraint formula ϕ_{P_G} is defined over E_{P_G}, I_{P_G}, and J_{P_G}. Applying δ_{P_G}, the definition function of P_G, takes all elements of E_{P_G} to elements of I_{P_G} or literal values in $\mathcal{V}$. Further applying β_G, the binding function of G, takes all elements of J_{P_G} to literal values in $\mathcal{V}$. Applying the match function $\beta_{G \to P}$ takes all elements of I_{P_G} to the corresponding elements in E_P. Finally, applying the definition function δ_P takes all elements of E_P to elements of J_P or literal values in $\mathcal{V}$.

The constraint formula ϕ_P is defined over E_P, I_P, and J_P. Applying δ_P takes elements of E_P to elements of J_P or literal values in $\mathcal{V}$. The match function $\beta_{P \to G}$ takes all elements of I_P to the corresponding elements in E_{P_G}. The definition function δ_{P_G} takes these elements of E_{P_G} to elements of J_{P_G} or literal values in $\mathcal{V}$. Finally the binding function β_G takes all elements of J_{P_G} to literal values in $\mathcal{V}$.

Once the correct parts of ψ are attached to gangsters, any match between those gangsters and other ClassAds must also satisfy ψ, or pass along new formulas to new gangsters. The passing of formulas can not last indefinitely, since the last ClassAd added to complete a can must not have any child ports and thus must not have any dependencies (i.e. $J_P = \varnothing$). Since all matches in a complete gang are either valid or conditionally valid with their conditions satisfied, and only complete gangs are output, all gangs output are valid. □

Completeness: To prove completeness we must first show that $NEXT$ always contains the correct values.

Lemma *For any gangster G, $NEXT$[G] is equal to the gangster representing the next open port, or **null** if there is no next open port.*

Proof: The proof is inductive.

Base Case: Let G_0 be the first gang created in line 2 of the GANGMATCH method in Figure 15. As G_0 represents the only port in the gang consisting of ClassAd C_0, $NEXT[G_0]$ is **null**.

Induction: We assume that at the beginning of the GENERATENEWGANGSTERS method in Figure 16 $NEXT$ has the correct values for all existing gangsters. If the ClassAd C has no child ports, then at line 15 G_{last} is **null**. This means that lines 21 and 23 will not be executed, and no changes will be made to $NEXT$. If C has one child port, then G_{last} will be set to the gangster created from the child port in line 14. We shall refer to this gangster as G_1. If C has two child ports, then during the second iteration of the **for** loop beginning at line 7 $NEXT[G_1]$ will be set to the gangster created from the second child port, designated G_2, in line 13. Line 14 then sets G_{last} to G_2. In either case G_{last} is set to the last new gangster created. In line 15 G' is set to $NEXT[G]$, by assumption the gangster representing the next open port in the gang after the one represented by G. If G' is **null** then $NEXT[G_{last}]$ is correctly set to null in line 24. Otherwise $NEXT[G_{last}]$ is correctly set to the new gangster created from G', designated G_3, in line 21. In line 22 $NEXT[G_3]$ is then set to $NEXT[G']$ insuring that the next open port after the port represented by G' remains the next open port when G_3 is created. $\Box$

A complete gang Γ is a sequence of ClassAds $C_0, \cdots, C_n$ where the parent port of each ClassAd C_i (where $i \neq 0$) matches with the first open port of the gang comprised of the ClassAds prior to C_i. We need to prove that for each valid gang Γ there is a sequence of rules in $RULES$ that generates the corresponding sequence of ClassAds. Let G_i be the gangster representing the first open port of the gang constructed from $C_0, \cdots, C_i$ (where $i < n$). We need to show that $G_0 \rightarrow C_0$, $G_{i+1} \rightarrow G_i\ C_{i+1}$ for each $0 < i < n$, and $S \rightarrow G_{n-1}\ C_n$ are all present in $RULES$ The first part is clear from line 3 of the GANGMATCH method in Figure 15. In line 8 for any rule containing a gangster equivalent to G in the right hand side a new rule is added with G substituted for the equivalent gangster. The remaining rules are generated in the GENERATENEWGANGSTERS method in Figure 16.

Assume GENERATENEWGANGSTERS is called on G_i and C_{i+1}. If C_{i+1} has one or more child ports, then the first child port is represented by a new gangster G_{new} in line 10. In line 12 a the rule $G_{new} \rightarrow G_i\ C_{i+1}$ is added to $RULES$. G_{new} represents the first open port in the gang constructed from $C_0, \cdots, C_{i+1}$, so G_{new} = G_{i+1}. If C_{i+1} has no child ports then G_{last} is **null** at line 15. By the above Lemma, G' is set to the next open port after the port represented by G_i ($NEXT[G_i]$) in line 15. If G' is not **null** then a new gangster G_{new} is created from G' in line 18 and the rule G_{new} $\rightarrow G_i\ C_{i+1}$ is added to $RULES$ in line 20. Since G_{new} represents the first open port in the gang constructed from $C_0, \cdots, C_{i+1}$, G_{new} = G_{i+1}. If G' is **null** then we have generated a complete gang. This is reflected in line 25 where the rule $S \rightarrow G_i\ C_{i+1}$ is added to rules. The only way this is possible is if i = $n - 1$. □

5.3.2 Complexity

The complexity of the algorithm may be described in terms of the following variables:

- c is the number of ClassAds in **C**.
- v is the maximum number of possible unique values for a given attribute.
- j is the maximum number of attributes imported from prior ports (dependencies) in all ports of all ClassAds in **C**

The number of possible unique gangsters is equal to the number of ports, which is $O(c)$, times the number of combinations of values for each port's dependencies, which is $O(v^j)$. If there are $O(cv^j)$ possible unique gangsters, then lines 11-13 of the GANGMATCH method in Figure 15 are executed $O(c^2v^j)$ times. The MATCH method in Figure 15 and the GENERATENEWGANGSTERS method in Figure 16 are each constant time if we assume that the number of attributes defined in each port is a constant. This means that there are $O(c^2v^j)$ total gangsters generated, so lines 5 and 6 of GANGMATCH are executed $O(c^2v^j)$ times. For each unique gangster G, only one rule with G on the left hand side is added to $RULES$, so line 8 is also executed $O(c^2v^j)$ times. Therefore the time complexity of the gangmatching algorithm is $O(c^2v^j)$. The space complexity is also $O(c^2v^j)$ since $SEEN$, ALT, $RULES$, and $NEXT$ are all bounded by the total number of gangsters.

The complexity of the certificate chain discovery algorithms in [12, 29] is expressed in terms of $|\mathcal{C}|$, the sum of the lengths of the right hand sides of all rules corresponding to certs in $\mathcal{C}$, and n_K, the number of unique public keys occurring in $\mathcal{C}$. In the ClassAd representation of SPKI/SDSI certificates described in Chapter 4 c is $O(|\mathcal{C}|)$, v is $O(n_k)$, and j = 1. Given these ClassAds, the algorithm has a time and space complexity of $O(n_K|\mathcal{C}|^2)$, the same worst case time complexity as the *post** algorithm for certificate chain discovery presented in [29].

```
[ // certificate (1)
  Ports = {
    [
      other = chain1;
      Type = "cert_request";
      Requirements = other.Type == "cert_offer" &&
        other.CertType == "Name" &&
        other.Issuer == "K_A" &&
        other.Identifier == "Bob";
    ],
    [
      other = chain2;
      Type = "cert_request";
      Requirements = other.Type == "cert_offer" &&
        other.CertType == "Auth" &&
        other.Issuer == chain1.Subject
    ],
    [
      other = request;
      Type = "cert_offer";
      CertType = "Auth";
      Issuer = "X";
      Subject = chain2.Subject;
      Requirements = other.Type == "cert_request"
    ]
  }
]

[ // certificate (2)
  Ports = {
    [
      other = request;
      Type = "cert_offer";
      CertType = "Name";
      Issuer = "K_A";
      Identifier = "Bob";
      Subject = "K_B";
      Requirements = other.Type == "cert_request"
    ]
  }
]
```

Figure 17: Gangmatching ClassAds for the SPKI/SDSI certificates (1) and (2)

```
[ // certificate (3)
  Ports = {
    [
      other = chain1;
      Type = "cert_request";
      Requirements = other.Type == "cert_offer" &&
        other.CertType == "Name" &&
        other.Issuer == "K_B" &&
        other.Identifier == "Carol"
    ],
    [
      other = request;
      Type = "cert_offer";
      CertType = "Auth";
      Issuer = "K_B";
      Subject = chain1.Subject;
      Requirements = other.Type == "cert_request"
    ]
  }
]

[ // certificate (4)
  Ports = {
    [
      other = request;
      Type = "cert_offer";
      CertType = "Name";
      Issuer = "K_B";
      Identifier = "Carol";
      Subject = "K_C";
      Requirements = other.Type == "cert_request"
    ]
  }
]
```

Figure 18: Gangmatching ClassAds for SPKI/SDSI certificates (3) and (4)

```
[
  Ports = {
    [
      other = chain;
      Type = "cert_request";
      Requirements = chain.Type == "cert_offer" &&
        chain.CertType == "Auth" &&
        chain.Issuer == "X" &&
        chain.Subject == "K_C"
    ]
  }
]
```

Figure 19: Gangmatching ClassAd for a SPKI/SDSI authorization request for access to resource X by principal K_C

Chapter 6

Gangmatching Analysis

Gangmatching analysis is essentially an extension of bilateral matching analysis. Between any two given ports, the same techniques can be used to determine why the first port does not match the second and vice versa. However, the presence of prior ports in a ClassAd introduces the possibility that one match may be dependent on the results of other matches. In addition, new problems arise from the more complex structure of a gang as opposed to two matching ClassAds.

A common problem in authorization systems is how to revoke a principal's access to a resource. For example, in SPKI/SDSI a principal may have access to a resource via several different certificate chains containing certificates issued by several different principals. In order to revoke the principal's access to the resource, at least one certificate in each such chain must be revoked. To avoid unnecessary disruption caused by certificate revocation, the set of certificates revoked should be minimal.

For example, given the following set of SPKI/SDSI certificates:

(1) $X\ \Box \rightarrow K_A$ Bob $\Box$

(2) $X\ \Box \rightarrow K_C$ Bob $\Box$

(3) K_A Bob $\rightarrow K_D$ Bob

(4) K_C Bob $\rightarrow K_D$ Bob

(5) K_D Bob $\rightarrow K_B$

(6) K_A Bob $\rightarrow K_B$

the request to authorize principal K_B (Bob) to authorize resource X can be satisfied by three different chains corresponding to three different subsets of certificates:

(1)(3)(5)

(1)(6)

(2)(4)(5)

An example of a minimal cut would be the removal of certs (1) and (4). Note that removing (1) or (5) affects two chains. One can not remove a certificate from one chain without removing it from another.

The *Break the Chain* problem may be abstracted to the problem of finding a minimal element in a subset lattice that passes a given test. In this case the top set in the lattice is the set of all certificates in C. The test on a given $C' \subseteq C$ is whether the certificates in C' grant the principal access to the resource. The problem of finding all such minimal elements has been shown to be NP-hard [26], but the problem of finding one such element is linear. Furthermore, finding k such elements for a constant k is polynomial: for $k > 1$ the complexity is $O(n^{k-1})$. In Section 6.1 we will apply this abstraction, then improve the performance by optimizing to reduce repeated work.

The *Missing Link* problem is the opposite of the Break the Chain problem. In this case a principal has no access to a resource, but may have elements of a certificate chain that would grant access. The problem is to find which certificates are needed to complete a chain that will authorize the principal to access the resource.

For example, given the following set of SPKI/SDSI certificates:

$$(1)\ X\ \square \rightarrow K_A\ \text{Admin}\ \square$$

$$(2)\ K_B\ \text{Carol} \rightarrow K_C$$

the request to authorize principal K_C (Carol) to authorize resource X can not be satisfied by any subset of certificates. An example of a certificate whose addition would satisfy the request for authorization is K_A Admin $\rightarrow K_B$ Carol.

The gangmatching equivalent of this problem is finding which ClassAds are needed to complete a gang. The solution to this problem is to run the gangmatching algorithm with a slight modification: When a port does not match any other ports, the gang is not abandoned; instead, the algorithm continues to match the rest of the ports in the gang and any dependencies on the unmatched port are ignored. When a partial gang has been completed, the "missing links" in the gang can be determined by using the requirements expressions of the unmatched ports, and the references to imported attributes in these ports. Satisfied requirements expressions elsewhere in the gang that contain such references can be partially evaluated to produce additional constraints for missing links. In Section 6.2 we will modify the gangmatching algorithm to accept prototype ClassAds that will capture these additional constraints.

6.1 Break the Chain

In order to revoke a principal's access to a resource, at least one certificate in every chain granting access must be revoked. To avoid unnecessary disruption caused by certificate revocation, the set of certificates revoked should be minimal. This problem may be abstracted to the problem of finding a minimal element in a subset lattice that passes a given test.

The revocation or *Break the Chain* problem can be stated as follows: given a resource R, a principal P, and a set of SPKI/SDSI certificates C find a minimal set of certs C_M such that C - C_M does not grant P access to R. The minimal element of a lattice (MEL) problem is as follows: given set S and test $T \subseteq 2^S$ where T is monotonic with respect to subset, find a minimal element of the subset lattice of S that passes test T (i.e. no subset of M is in T).

We now show that the *Break the Chain* problem can be reduced to the MEL problem. Let S = C and T = $\{C' \subseteq C \mid C$ - C' fails to authorize P to access $R\}$. We must now prove the monotonicity of T. Let $C' \in T$. If C' = C it has no supersets in 2^C. If $C' \subset C$, let C'' = $C' \cup c$ where $c \in C$ - C'. C - C'' = C - ($C' \cup c$) = (C - C') - c. Since C - C' fails to authorize P to access R, (C - C') - c must also fail to authorize P to access R. So $C'' \in T$. So T is monotonic with respect to subset.

A MEL M of (S,T) has the following properties:

- C - M will not authorize P to access R ($M \in T$)

- if $M' \subset M$, C = M' will authorize P to access R (M is minimal)

These are precisely the criteria of the "Break the Chain" problem. □

To find one MEL requires $O(|C|)$ calls to test membership in T. The gangmatching algorithm in Section 5.3 can be used to perform the test. However, repeated calls to the gangmatching algorithm result in a great deal of repeated work. A closer look at the algorithm to find a MEL reveals a way to avoid this extra computation. The MEL algorithm starts with the set S, repeatedly removes elements and tests the result. If the result passes the test, the algorithm continues removing elements. If the result fails the test, the element removed is added to the MEL set, and is included in each successive test.

In the *Break the Chain* problem, T is testing to see if the certificates not in the current set fail to authorize P to access R. Since we are dealing with the set complement, each successive step either adds a new certificate to the complement, or removes the last certificate added (adding it to the MEL set) and adds a new certificate. Using this information, and the grammar produced by the gangmatching algorithm, we can devise an algorithm that performs these two set operations without any re-computation.

The algorithm in Figure 20 takes as input a set of ClassAds $\mathcal{C}$ that may be removed during the course of the algorithm, a second set of ClassAds $\mathcal{C}_{base}$ that may not be removed, and a set of rules generated by the gangmatching algorithm from the ClassAds in $\mathcal{C}$ and $\mathcal{C}_{base}$. In the case of the ClassAd representation of SPKI/SDSI certificates, $\mathcal{C}_{base}$ consists of the root ClassAd C_0 and the seed ClassAds described in Section 4.4, while $\mathcal{C}$ consists of the ClassAds representing actual certificates. Two sets of ClassAds, $\mathcal{C}_{curr}$ and $\mathcal{C}_{prev}$ are maintained throughout the algorithm to represent the current set of ClassAds being tested and the most recent set of ClassAds that passed the test. The reason for keeping track of two sets is to easily rollback to the previous state if adding a ClassAd causes the test to fail. Similarly, the sets $\mathcal{G}_{curr}$ and $\mathcal{G}_{prev}$ represent the set of gangsters that can be generated by ClassAds in $\mathcal{C}_{curr}$ and $\mathcal{C}_{prev}$. We initialize $\mathcal{C}_{prev}$ as $\mathcal{C}_{base}$ and $\mathcal{G}_{prev}$ as G_0, the gangster created from C_0. In the while loop, ClassAds from $\mathcal{C}$ are added along with the ClassAds in $\mathcal{C}_{prev}$ to $\mathcal{C}_{curr}$, then the ADDC method is called to determine the consequences of adding C. If ADDC returns false, then adding C results in the test failing, in which case all new additions are ignored and $\mathcal{C}_{prev}$ and $\mathcal{G}_{prev}$ remain the same. Otherwise, the test has passed and $\mathcal{C}_{prev}$ and $\mathcal{G}_{prev}$ are replaced by $\mathcal{C}_{curr}$ and $\mathcal{G}_{curr}$.

The ADDC method matches up C with each gangster G in $\mathcal{G}_{curr}$ to see if there are any rules in $RULES$ that are of the form ($G' \rightarrow G\ C$). If there are, then G' must be added to $\mathcal{G}_{curr}$ since it can be generated by ClassAd C and gangster G. This also results in the recursive method ADDG that, like ADDC, checks if any rules have a right hand side containing G' and some ClassAd in $\mathcal{C}$. If at any point in ADDC or ADDG a rule is found with S as the left hand side, then the set of ClassAds in $\mathcal{C}_{curr}$ generates a complete gang corresponding to a certificate chain authorizing P to access R. There are two consequences to encountering such a rule: we must roll back our sets of ClassAds and gangsters to ones in $\mathcal{C}_{prev}$ and $\mathcal{G}_{prev}$, and we must add C to CUT. At the end of the algorithm, the set CUT will contain a minimal set of (certificate) ClassAds whose removal will prevent the authorization we wish to revoke.

Returning to the *Break the Chain* example at the beginning of the chapter, the grammar generated by the gangmatching algorithm, given this request and these certificates, is as follows (S_{K_B} refers to the seed ClassAd for principal K_B):

$$S \rightarrow G_{11}\ S_{K_B} \mid G_{10}\ S_{K_B} \mid G_7\ S_{K_B}$$

$$G_{11} \rightarrow G_8\ C_5$$

$$G_{10} \rightarrow G_5\ C_5$$

$$G_8 \rightarrow G_3\ C_4$$

$$G_7 \rightarrow G_1\ C_6$$

$$G_5 \rightarrow G_1\ C_3$$

$$G_3 \rightarrow G_0\ C_2$$

$$G_1 \rightarrow G_0\ C_1$$

$$G_0 \rightarrow C_0$$

Passing this grammar to the *Break the Chain* algorithm, one of the possible runs (that vary depending on the permutation of ClassAds picked) is shown in Figure 21. The value of CUT at the end of the run is $\{C_1, C_4\}$, corresponding to certificates (1) and (4).

6.2 Missing Link

The solution to the missing link problem takes advantage of the partial evaluation facility of the gangmatching algorithm. In order to find a missing link, one must first provide prototypes of acceptable ClassAds. These prototypes are then added to the set $\mathcal{C}$, and the gangmatching algorithm, with a few minor modifications, is run. If the algorithm generates any valid complete gangs, it will also generate constraints on any prototype ClassAds in each gang. These constraints, along with the prototypes, can be used to specify candidate missing link ClassAds.

A prototype ClassAd is structured the same way as a normal ClassAd, except with dummy variables substituting for literal values. These dummy variables must be treated as literal values by the methods of the gangmatching algorithm, with the exception of the MATCH partial evaluation method where they are treated as if they were attribute references. Thus, when a potential match is tested between an independent gangster G and a port P of a dummy ClassAd, the MATCH method will return a formula over the dummy variables in P and the attributes imported from ports prior to P (J_P).

Let $\mathcal{D}$ be a set of dummy variables. Let G = (P_G, β_G, L_G, $\varnothing$) be an independent gangster. Let P = (E_P, I_P, J_P, δ_P, ϕ_P) be a parent port of a prototype ClassAd D. ψ = MATCH(G, D) is a formula over $J_P \cup \mathcal{V} \cup \mathcal{D}$. Figure 22 shows a revised version of the GENERATENEWGANGSTERS method from Figure 16. GENERATE-NEWGANGSTERS(G, D, ψ) creates a binding relation $L_M \subseteq \mathcal{I} \times (\mathcal{V} \cup \mathcal{D} \cup J_P)$. In line 8 of Figure 22 a binding relation L is created from $L_M \cap (\mathcal{I} \times I_{P'})$ for each child port P' of D. Since we wish to treat dummy variables as literal values, no elements of L_M containing elements of $\mathcal{D}$ are included in these binding relations. Similarly ψ' contains no occurrences of elements of $\mathcal{D}$. Therefore all occurrences of dummy variables in each G_{new} created are restricted to those in the respective child port P'

of D. Note that for each G_{new} created, a rule ($G_{new} \rightarrow G\ D$) is added to $RULES$. All of the changes we need to make to the GENERATENEWGANGSTERS method are in lines 15, 17, 18, and 25.

The first change we make to the GENERATENEWGANGSTERS method (line 15) is to create a new constraint formula $\psi_{\mathcal{D}}$ made up of all of the predicates in ψ that contain occurrences of dummy variables. Again, we wish to treat dummy variables as literal values here, so in line 17 the binding function β must include elements of L_M that contain occurrences of elements of $\mathcal{D}$. In line 18 we append $\psi_{\mathcal{D}}$ to $\psi_{G'}$, ensuring that the predicates over dummy variables generated by matching G with C are preserved. By passing $\psi_{\mathcal{D}}$ along in this way we assure that when a complete gang is created the ψ from the last match contains all of the predicates over dummy variables, hence the addition of (ψ) to the rule generated in line 25.

Returning to the *Missing Link* examples at the beginning of the chapter, given this request and this set of certificates we can use the gangmatching algorithm in Figure 15 with the modified GENERATENEWGANGSTERS method in Figure 22 to find a missing link to complete the desired certificate chain. We will also need a prototype ClassAd with dummy variables in place of literal values. Let $\mathcal{D} = \{IssuerValue_0^D, IdentifierValue_0^D, IssuerValue_1^D, IdentifierValue_1^D\}$. Let D be a prototype ClassAd with two ports P_0^D and P_1^D where:

$$\delta_0^D = \{Type \mapsto \texttt{"cert\_request"}\}$$

$$\phi_0^D = (Type == \texttt{"cert\_offer"}) \wedge (CertType == \texttt{"Name"}) \wedge (Issuer == IssuerValue_0^D) \wedge (Identifier == IdentifierValue_0^D)$$

$$\delta_1^D = \{Type \mapsto \texttt{"cert\_offer"}, CertType \mapsto \texttt{"Name"}, Issuer \mapsto$$
$$IssuerValue_1^D, Identifier \mapsto IdentifierValue_1^D\}$$
$$\phi_1^D = (Type == \texttt{"cert\_request"})$$

Figure 23 shows example run of the *Missing Link* version of the gangmatching algorithm. The end result is a complete gang including the dummy ClassAd, annotated with the formula $\psi = (IssuerValue_1^D == \texttt{"K\_A"}) \wedge (IdentifierValue_1^D == \texttt{"Admin"}) \wedge (\texttt{"K\_B"} == IssuerValue_0^D) \wedge (\texttt{"Carol"} == IdentifierValue_0^D)$. These constraints on the dummy variables define the certificate K_A Admin $\rightarrow$ K_B Carol.

BREAKTHECHAIN($\mathcal{C}$, $\mathcal{C}_{base}$, $RULES$)
$CUT \leftarrow \varnothing$
$\mathcal{G}_{prev} \leftarrow \{G_0\}$
$\mathcal{C}_{prev} \leftarrow \mathcal{C}_{base}$
while $\mathcal{C} \neq \varnothing$
  pop C from $\mathcal{C}$
  $\mathcal{G}_{curr} \leftarrow \mathcal{G}_{prev}$
  $\mathcal{C}_{curr} \leftarrow \mathcal{C}_{prev} \cup C$
  if ADDC(C, $\mathcal{G}_{curr}$, $\mathcal{C}_{curr}$, CUT, $RULES$)
    $\mathcal{G}_{prev} \leftarrow \mathcal{G}_{curr}$
    $\mathcal{C}_{prev} \leftarrow \mathcal{C}_{curr}$
return CUT

ADDC(C, $\mathcal{G}$, $\mathcal{C}$, CUT, $RULES$)
for each G in $\mathcal{G}$
  if $(S \rightarrow G\ C) \in RULES$
    $CUT \leftarrow CUT \cup C$
    return false
  else for each G' where $G' \rightarrow G\ C$
      if ADDG(G, $\mathcal{G}$, $\mathcal{C}$, $RULES$) = **false**
        $CUT \leftarrow CUT \cup C$
        return false
return true

ADDG(G, $\mathcal{G}$, $\mathcal{C}$, $RULES$)
for each C in $\mathcal{C}$
  if $(S \rightarrow G\ C) \in RULES$
    return false
  else $\mathcal{G} \leftarrow \mathcal{G} \cup G$
    for each G' where $G' \rightarrow G\ C$
      if ADDG(G, $\mathcal{G}$, $\mathcal{C}$, $RULES$) = **false**
        return false
return true

Figure 20: The *Break the Chain* algorithm

$\mathcal{G}_{curr}$ ($\mathcal{G}_{\mathbf{prev}}$)	$\mathcal{C}_{curr}$ ($\mathcal{C}_{\mathbf{prev}}$)	**Action**	CUT
$\{G_0\}$	$\{\mathbf{S_{K_B}}, \mathbf{C_0}\}$	add C_2	$\varnothing$
$\{\mathbf{G_0}, G_3\}$	$\{\mathbf{S_{K_B}}, \mathbf{C_0}, C_2\}$	add C_6	$\varnothing$
$\{\mathbf{G_0}, \mathbf{G_3}\}$	$\{\mathbf{S_{K_B}}, \mathbf{C_0}, \mathbf{C_2}, C_6\}$	add C_5	$\varnothing$
$\{\mathbf{G_0}, \mathbf{G_3}\}$	$\{\mathbf{S_{K_B}}, \mathbf{C_0}, \mathbf{C_2}, \mathbf{C_6}, C_5\}$	add C_1	$\varnothing$
$\{\mathbf{G_0}, \mathbf{G_3}, G_1, G_5, G_7, G_{10}, S\}$	$\{\mathbf{S_{K_B}}, \mathbf{C_0}, \mathbf{C_2}, \mathbf{C_6}, \mathbf{C_5}, C_1\}$	remove C_1 add C_4	$\{C_1\}$
$\{\mathbf{G_0}, \mathbf{G_3}, G_8, G_{11}, S\}$	$\{\mathbf{S_{K_B}}, \mathbf{C_0}, \mathbf{C_2}, \mathbf{C_6}, \mathbf{C_5}, C_4\}$	remove C_4 add C_3	$\{C_1, C_4\}$

Figure 21: An example run of the *Break the Chain* algorithm

GENERATENEWGANGSTERS(G, C, ψ)

$P \leftarrow C$'s parent port; $L_M \leftarrow \varnothing$;
for each $attr_{(e)} \mapsto Y \in \delta_P$
    if $(X, attr) \in L_G$
        $L_M \leftarrow L_M \cup \{(X, Y)\}$
    $L_M \leftarrow L_M \cup \{(attr, Y)\}$
$G_{last} \leftarrow$ **null**
for each child port P' of C
    $L \leftarrow \{(X, Y) \in L_M \mid Y \in I_{P'}\}$
    $\psi' \leftarrow \wedge$ {predicates in ψ containing an $i \in I_{P'}$}
    $G_{new} \leftarrow$ ADDGANGSTER(P', $\varnothing$, L, ψ')
    if G_{last} = **null**
        $RULES \leftarrow RULES \cup \{(G_{new} \rightarrow G\ C)\}$
    else $NEXT[G_{last}] \leftarrow G_{new}$
    $G_{last} \leftarrow G_{new}$
$G' \leftarrow NEXT[G]$; $\underline{\psi_{\mathcal{D}} \leftarrow \wedge \{\text{predicates in } \psi \text{ containing an element of } \mathcal{D}\}}$
if $G' \neq$ **null**
    $\beta \leftarrow \{(X, Y) \in L_M \mid, X \in J_{P_{G'}}, Y \in \mathcal{V} \underline{\cup \mathcal{D}}\}$
    $G_{new} \leftarrow$ ADDGANGSTER($P_{G'}$, β, $L_{G'}$, $\psi_{G'}$ $\underline{\wedge\ \psi_{\mathcal{D}}}$)
    if G_{last} = **null**
        $RULES \leftarrow RULES \cup \{(G_{new} \rightarrow G\ C)\}$
    else $NEXT[G_{last}] \leftarrow G_{new}$
    $NEXT[G_{new}] \leftarrow NEXT[G']$
elsif $G_{last} \neq$ **null**
    $NEXT[G_{last}] \leftarrow$ **null**
else $RULES \leftarrow RULES \cup \{(S \rightarrow G\ C\ \underline{(\psi)})\}$

Figure 22: The modified version of the GenerateNewGangsters method (changes indicated by underlines)

Match	**Gangsters**	**Rules**
	$G_0 = (P_0^0, \varnothing, \varnothing, \mathbf{T})$	$G_0 \to C_0$
(G_0, C_1)	$G_1 = (P_0^1, \varnothing, \varnothing, \mathbf{T})$ $G_2 = (P_1^1, \varnothing, \{(Subject_0^0, Subject)\},$ $(Subject == \texttt{"K\_C"})$	$G_1 \to G_0\ C_1$
(G_1, D)	$G_3 = (P_0^D, \varnothing, \{(Subject_0^1, Subject)\},$ $(Subject == \texttt{"K\_C"})$ $G_4 = (P_1^1, \varnothing, \{(Subject_0^0, Subject)\},$ $(Subject == \texttt{"K\_C"}) \wedge$ $(IssuerValue_1^D == \texttt{"K\_A"}) \wedge$ $(IdentifierValue_1^D == \texttt{"Admin"}))$	$G_3 \to G_1\ D$
(G_3, C_2)	$G_5 = (P_1^1, \{Subject_0^1 \mapsto \texttt{"K\_C"}\},$ $\{(Subject_0^0, Subject)\},$ $(Subject == \texttt{"K\_C"}) \wedge$ $(IssuerValue_1^D == \texttt{"K\_A"}) \wedge$ $(IdentifierValue_1^D == \texttt{"Admin"}) \wedge$ $(\texttt{"K\_B"} == IssuerValue_0^D) \wedge$ $(\texttt{"Carol"} == IdentifierValue_0^D))$	$G_5 \to G_3\ C_2$
(G_5, S_{K_C})		$S \to G_5\ S_{K_B}\ (\psi)$

Figure 23: An example run of the *Missing Link* algorithm

Chapter 7

Related Work

The research related to the work presented in this dissertation can be divided into five general categories: matchmaking, resource management, trust management, policy languages and frameworks, and query analysis. Firstly, a survey of matchmaking research covers gangmatching, alternatives to gangmatching, agent matchmaking, and unification based matchmaking. Secondly, a review of work on resource management includes Condor, Globus, and other resource management frameworks based on service level agreements (SLAs). Thirdly, SPKI/SDSI research is explored along with alternate trust management schemes. Fourthly, several policy languages and frameworks for networks, distributed systems, the semantic web, and grid computing are investigated. Finally, research on database query analysis is compared to the policy analysis methods presented in Chapter 3.

7.1 Matchmaking

The fundamental concepts of gangmatching are laid out in [53]. A more thorough discussion of gangmatching as well as two optimizations of the original gangmatching algorithm can be found in [51]. Both optimizations — the first involves indexing

ClassAds, the second involves out-of-order matching — are potentially compatible with the enhanced gangmatching algorithm described in Chapter 5. Out-of-order matching may require a more sophisticated method for assembling the regular grammar representing valid gangs.

Building on work using the ClassAd language to specify set-matching policies [42] for grid resource selection, a new language and matchmaking mechanism called Redline [41] has been developed based on a constraint language model. Set-matching involves matching a single request ClassAd with an unspecified number of offer ClassAds. Redline expresses both the requirements and the attributes of an entity as constraints, making it unclear which entities attributes belong to. However, Redline does allow querying of requirements, a feature that is not currently possible in the ClassAd language.

Matchmaking has been explored in the field of agent technology [59, 48, 49, 60]. There are some similarities between ClassAds and agent communication languages [25, 22, 58], though ClassAds employ a representation more akin to a database record than the rule-based representation used by these languages. Several matchmaking frameworks [63, 56, 50, 47, 18, 36] based on description logics [19, 27, 5] have also been proposed. Like the agent communication languages, these employ a rule-based representation.

There are also similarities between ClassAds matchmaking and the unification-based matching used by Linda [24] and Datalog. Linda uses tuples containing variables or literals to search a tuple space for a matching tuple. Datalog operates similarly on relational databases. While unification is certainly powerful enough to encompass the functionality of ClassAd-based matchmaking, the syntax of boolean expressions used by the ClassAd language is clearer and more concise. There have been efforts to draw on literature on constraint logic programming (CLP) [28], constraint query languages (CQL) [35], and constraint databases (CDB) [54] to add constraints to Datalog [55, 39]. However, CQL and CDB typically assume a fixed schema whereas ClassAds use a semi-structured data model.

7.2 Resource Management

Resource selection policy specification is an important issue in grid computing. Condor uses the ClassAd language to specify resource selection policies in grid computing [61]. In Globus [13, 14], a suite of grid computing applications, customers describe required resources in a resource specification language (RSL) based on a predefined schema of the resources database in contrast to the schema free ClassAd language. However, resources cannot place constraints on requests as in the bilateral matchmaking model utilized by Condor.

A more powerful framework for resource management in distributed systems, the Services Negotiation and Acquisition Protocol (SNAP) [15], maps resource interactions to platform-independent service level agreements (SLAs). SNAP uses an extensible language J for describing jobs along with a subset language R to describe resource requirements. J is similar in purpose to RSL and ClassAds, but is more extensible than the former and more rigorously typed than the latter. However, the simplicity of the ClassAd language is one of its most attractive features. A more complex language like J may not be as easy to use.

Another approach to resource management, also using SLAs and geared towards grid computing, has been described here [20]. This framework uses a resource allocation policy language that is more expressive than RSL, but is not as expressive as the ClassAd language. In particular there is no support for arbitrary boolean expressions like those available with ClassAds.

7.3 Trust Management

Authorization policy is a key component of trust management systems. The SPKI/SDSI [21] framework has been discussed in some detail in Section 2.2. The term rewriting approach to SPKI/SDSI was introduced in [12] along with an algorithm for certificate chain discovery.

It is also possible to use pushdown systems (PDS) to represent SPKI/SDSI rewrite rules [29, 30]. The enhanced gangmatching algorithm in Chapter 5 began as a generalization of the *post** algorithm for PDS reachability. A PDS is essentially a Pushdown Automaton without the capacity for generating a language. SPKI/SDSI certificates expressed as rewrite rules can be converted into a set of rules for a PDS, and algorithms are available to enumerate all possible resulting stack states given an initial stack state (*post**) and all stack states antecedent to a given stack state (*pre**). Either the *pre** or *post** algorithm can be used to generate SPKI/SDSI certificate chains. In addition several other aspects of Pushdown Systems are exploited to answer specific authorization questions. However, neither the *Break the Chain* nor the *Missing Link* problems are discussed. As a side note, it is possible to represent a generic pushdown system using ClassAds and gangmatching, though the enhanced gangmatching algorithm above would require some modifications.

Another trust management system, Keynote [8], uses the concept of assertions to specify authorization policy. An assertion is very similar in function and form to a SPKI/SDSI authorization certificate in that it identifies a principal making the assertion (similar to an issuer of an auth cert in SPKI/SDSI) the recipients (subjects) of the authorization and conditions of authorization. It may be possible to use the ClassAd language as a concrete representation of the KeyNote model.

7.4 Policy Languages and Frameworks

The resource selection and authorization policies discussed in this dissertation both fall under the category of *provisions*. Provisions are conditions that must be satisfied or actions that must occur before a decision takes place. In contrast *obligations* are conditions or actions that must be fulfilled after a decision has been made [7]. One example of an obligation policy is a service level agreement (SLA). An SLA is an agreement between a service provider and a customer that specifies certain attributes of the service such as availability, serviceability, performance and operation [66].

Obligation policy is the main focus in policy based management of networks. The WSLA [3, 37, 17] framework for service level agreements uses a somewhat cumbersome XML based representation for specification of obligation policy. PDL [43] expresses obligation policies as event-condition-action rules. This framework is somewhat similar to the use of the ClassAd language to specify policy in Condor, except that the events and actions are not formally defined. For example before an execute machine can run a job Condor must evaluate a `Start` expression (usually identical to the machine's `Requirements` expression). In this case the event is a successful match, the condition is the `Start` expression, and the action is running the job on the machine. Hawkeye [2], a system monitoring application, uses ClassAds to describe system events, ClassAd expressions to serve as triggers for identifying interesting behavior, and matchmaking to detect when events set off these triggers.

The Ponder policy language [16] can also be used to express both obligation and authorization policies. Ponder is an object oriented language that allows for declaration of policy types and instantiation of those types. Authorization policies contain a subject (corresponding to a SPKI/SDSI issuer), a target (a SPKI/SDSI subject), a set of actions being authorized, and a constraint expression indicating when this action may be authorized. Additional authorization related polices in Ponder include information filtering, delegation, and refrain policies. Obligation policies also contain a subject, a target, a set of actions, and a "when" expression, in addition to a specification of an event that triggers the policy. Ponder is clearly the most expressive of the policy languages described thus far, but it does not have the capability to express resource selection policies.

Several other policy languages – such as Rei [33, 32, 34], Kaos [65, 64, 45], have been developed specifically for the semantic web and grid computing applications. These languages are typically based on description logics such as DAML and OWL. A comparison of Rei, Kaos and Ponder is presented here [62]. Another language based on description logics called PeerTrust [23, 6] was developed specifically for automated trust negotiation. The rule language used by the PROTUNE [10] trust negotiation framework is partially based on PeerTrust. A framework for policy analysis of rule-based policies [11], similar to the work presented in Chapters 3 and 6, has been proposed. Aside from a rule-based notation, the most significant difference between these languages and ClassAds is that none of them have been applied to resource selection policy.

7.5 Query Analysis

Most of the work relevant to ClassAd analysis is in literature on databases, particularly on cooperative query answering. In [46] a mechanism called SEAVE is presented for extracting and verifying presuppositions from queries. This mechanism identifies queries that result in null answers, then finds more general queries by weakening or deleting query sub-expressions. The result is a set of maximally general erroneous presuppositions that may be of more value to the user than a simple null answer.

Similar techniques are discussed more formally in [26]. Godfrey discusses identification of *minimal failing sub-queries* (MFSs) and *maximal succeeding sub-queries* (MSSs). Godfrey's MFSs are analogous to the erroneous presuppositions generated by Motro's SEAVE mechanism. The MSSs are the least general generalizations of the initial successful query. An algorithm called ISHMAEL is presented that enumerates MFSs and MSSs. This algorithm is NP-hard for queries of arbitrary length, but remains polynomial for fixed length queries.

Finally, in [44] the notion of a *query difference operator* is introduced to indicate missing information in query results. The authors discuss a system of resource agents, brokers, and user agents that resembles the distributed framework used by Condor. The primary focus of this work is to indicate the incompleteness of query answers. The query difference operator is used to generate the description of the set of results covered by the query, but not covered by the query answer. This set is expressed in relational algebra and can presumably be converted into a pseudo-English response for the user.

ClassAd analysis uses similar techniques and notions to provide useful information regarding matchmaking failure. As discussed previously, our conflict detection algorithm covers similar territory as [26]. One key difference, as with the CQL and CDB work discussed earlier, is the semi-structured data model which, unlike the relational model discussed in the cited publications, does not require a fixed schema. Another important difference is the reflexive nature of ClassAds. In database terms a ClassAd contains both a query (the requirements expression) and a record (the set of attributes with literal values). Nevertheless, many of the issues encountered in ClassAd analysis are applicable in database query analysis, web search, or any other field where boolean expressions are used as constraints.

Chapter 8

Conclusions and Future Work

Distributed computing environments provide users with a wide range of services that a single isolated system can not provide. However, as in this world, with great power there must also come – great responsibility [38]. Policies must be designed and enforced to protect the interests of users and providers of these services. Resource selection policies address the question: What kind of resource does a principal want, and is such a resource available? Access control policies address the question: Can a principal be trusted to have access to a given resource?

The framework for policy specification and interpretation presented in this dissertation provides a clearing house for both types of policies. It is built on the simple yet powerful concept of matchmaking. The ClassAd language and matchmaking algorithms were initially developed to solve resource selection problems in a distributed system. As we have shown, the same framework with some minor modifications is applicable to managing access control policies.

We have demonstrated that the ClassAd language can be used to specify SPKI/SDSI authorization policies, and an enhanced gangmatching algorithm can be used to assemble SPKI/SDSI certificate chains correctly and efficiently. We have also presented the necessary theoretical underpinnings of the enhanced gangmatching algorithm which generalize beyond the specific instance of SPKI/SDSI certificate chain discovery. Finally, we have demonstrated analysis techniques for bilateral and multilateral matchmaking that serve as essential tools for comprehending matchmaking results. Taken together these contributions provide a robust framework for specifying and interpreting resource allocation policies.

Further research is possible in a number of areas. The enhanced gangmatching algorithm presented in Chapter 5 allows for the creation of gangs with unlimited depth, such as the ClassAd equivalent of a SPKI/SDSI certificate chain. Set-matching [42] is an extension of gangmatching that allows for the creation of gangs with unlimited *breadth*, such as an unspecified number of compute machines satisfying a requirement for minimum total processing power. It is possible that enhanced gangmatching and set-matching could be integrated into a more powerful matchmaking process.

In Chapter 7, a distinction was drawn between two types of policy: *provisions* and *obligations* [7]. The resource selection and authorization policies discussed in this dissertation fall under the category of provisions, or policies that must be adhered to before a resource or service can be used. Obligations are policies that dictate the terms of use for a resource or service while it is being used. A common type of obligation is a service level agreement (SLA). The service level agreements described in the WSLA [3] framework could be expressed as ClassAds, a matchmaking process could be used to determine if the SLA is violated, and matchmaking analysis could be used to determine the cause of the violation.

An aspect of SPKI/SDSI authorization that is not dealt with in this dissertation is the actual rights granted by a certificate. Specifying these rights and determining how they are delegated can be very complex. However, a basic implementation of rights delegation could be added to the ClassAd representation of SPKI/SDSI certificates described in Chapter 4 without impacting the algorithm discussed in Chapter 5.

The algorithm presented in Chapter 5, like the algorithms in [12] and [29] assume a centralized facility for certificate chain assembly. This means that while the specification of policy may be distributed, the interpretation of policy is not. A distributed algorithm for assembling credential chains using weighted pushdown systems has been proposed for SPKI/SDSI [31]. Additionally, the trust management language RT_0 [40] was designed to support a distributed algorithm that can be applied to SDSI. In the general case of gangmatching, a distributed algorithm would certainly be possible, but not without some added complications. The primary problem is how to make temporary reservations of resources that may be canceled if a complete gang can not be found. As in distributed transactions in database systems, some sort of two-phased commit could be used.

In the realm of resource allocation, matchmaking provides the convenience of bringing together principals with common interests. The facility for policy diagnostics using matchmaking analysis provides the equally important role of determining why policies are successful or unsuccessful. Included in the great responsibility referred to above is the responsibility to evaluate the effectiveness of policies and to change these policies when they are ineffective or doing harm. The maintenance of healthy communities, whether real or virtual, requires the free flow of information built on the foundation of trust, integrity, and common goals.

Bibliography

[1] Condor version 6.8 manual, section 3.6: Security in condor. http://www.cs.wisc.edu/condor/manual/v6.8/3_6Security.html.

[2] A monitoring and management tool for distributed systems. http://www.cs.wisc.edu/condor/hawkeye/.

[3] Web Service Level Agreements (WSLA) Project. http://www.research.ibm.com/wsla/.

[4] A. Arpaci-Dusseau, R. Arpaci-Dusseau, N. Burnett, T. Denehy, T. Engle, H. Gunawi, J. Nugent, and F. Popovici. Transforming policies into mechanisms with infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[5] F. Baader and W. Nutt. Basic description logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P.F. Patel-Schneide, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[6] J. Basney, W. Nejdl, D. Olmedilla, V. Welch, and M. Winslett. Negotiating trust on the grid. In *Semantic Grid*, 2005.

[7] C. Bettini, S. Jajodia, S. Wang, and D. Wijesekera. Provisions and obligations in policy rule management and security applications. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 502–513, Hong Kong, China, August 2002.

[8] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote trust-management system version 2. RFC 2704, September 1999.

[9] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Conference on Security and Privacy*, May 1996.

[10] P. A. Bonatti and D. Olmedilla. Driving and monitoring provisional trust negotiation with metapolicies. In *POLICY*, pages 14–23, 2005.

[11] P. A. Bonatti, D. Olmedilla, and J. Peer. Advanced policy explanations on the web. In *ECAI*, 2006.

[12] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[13] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proceedings of IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[14] Karl Czajkowski, Ian T. Foster, and Carl Kesselman. Resource co-allocation in computational grids. In *HPDC*, 1999.

[15] Karl Czajkowski, Ian T. Foster, Carl Kesselman, Volker Sander, and Steven Tuecke. Snap: A protocol for negotiating service level agreements and coordinating resource management in distributed systems. In *JSSPP '02: Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 153–183, London, UK, 2002. Springer-Verlag.

[16] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–38, 2001.

[17] M. Debusmann and A. Keller. Sla-driven management of distributed systems using the common information model. mar 2003.

[18] G. Denker, L. Kagal, T. Finin, M. Paolucci, and K. P. Sycara. Security for daml web services: Annotation and matchmaking. In *International Semantic Web Conference*, pages 335–350, 2003.

[19] F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Stanford, California, 1996.

[20] C. Dumitrescu, M. Wilde, and I. T. Foster. A model for usage policy-based resource allocation in grids. In *POLICY*, pages 191–200, 2005.

[21] C. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, September 1999.

[22] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Proc. of the Third Int'l Conf. on Information and Knowledge Management, CIKM-94*. ACM press, nov 1994.

[23] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. Seamons, and M. Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *ESWS*, pages 342–356, 2004.

[24] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[25] M. Genesereth, , N. Singh, and M. Syed. A distributed anonymous knowledge sharing approach to software interoperation. In *Proc. of the Int'l Symposium on Fifth Generation Computing Systems*, pages 125–139, 1994.

[26] P. Godfrey. Minimization in cooperative response to failing database queries. *International Journal of Cooperative Information Systems (IJCIS)*, 6(2):95–149, June 1997.

[27] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming*

and Automated Reasoning (LPAR99), number 1705, pages 161–180. Springer-Verlag, 1999.

[28] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[29] S. Jha and T. Reps. Analysis of SPKI/SDSI certificates using model checking. In *Proceedings of IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 2002.

[30] S. Jha and T. W. Reps. Model checking spki/sdsi. *Journal of Computer Security*, 12(3-4):317–353, 2004.

[31] S. Jha, S. Schwoon, H. Wang, and T. W. Reps. Weighted pushdown systems and trust-management systems. In *TACAS*, pages 1–26, 2006.

[32] L. Kagal, T. Finin, and A. Joshi. A policy based approach to security for the semantic web. In *International Semantic Web Conference*, pages 402–418, 2003.

[33] L. Kagal, T. W. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *POLICY*, pages 63–, 2003.

[34] L. Kagal, M. Paolucci, N. Srinivasan, G. Denker, T. Finin, and K. Sycara. Authorization and privacy for semantic web services. *IEEE Intelligent Systems*, 19(4):50–56, 2004.

[35] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *PODS*, pages 299–313, 1990.

[36] T. Kawamura, J. De Blasio, T. Hasegawa, M. Paolucci, and K. Sycara. Public deployment of semantic service matchmaker with uddi business registry. In *International Semantic Web Conference*, pages 752–766, 2004.

[37] A. Keller and H. Ludwig. Defining and monitoring service level agreements for dynamic e-business. nov 2002.

[38] S. Lee. Amazing Stories #15, August 1962.

[39] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *PADL*, pages 58–73, 2003.

[40] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, 2003.

[41] C. Liu and I. Foster. A constraint language approach to matchmaking. In *Proceedings of the 14th International Workshop on Research Issues on Data Engineering (RIDE'04)*, pages 7–14, March 2004.

[42] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC11)*, July 2002.

[43] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *AAAI/IAAI*, pages 291–298, 1999.

[44] M. Minock, M. Rusinkiewicz, and B. Perry. The identification of missing information resources by using the query difference operator. Technical report, MCC, April 1999.

[45] L. Moreau, J. M. Bradshaw, M. Breedy, L. Bunch, P. J. Hayes, M. Johnson, S. Kulkarni, J. Lott, N. Suri, and A. Uszok. Behavioural specification of grid services with the kaos policy language. In *CCGRID*, pages 816–823, 2005.

[46] A. Motro. SEAVE: A mechanism for verifying user presuppositions in query systems. *ACM Transactions on Office Information Systems*, 4(4):312–330, October 1986.

[47] T. Di Noia, E. Di Sciascio, F. M. Donini, and M. Mongiello. Abductive matchmaking using description logics. In *Proceedings of Eighteenth International Joint Conference on Artificial Intelligence (IJCAI03)*, pages 337–342, aug 2003.

[48] E. Ogston and S. Vassiliadis. Local distributed agent matchmaking. In *CoopIS*, pages 67–79, 2001.

[49] E. Ogston and S. Vassiliadis. Unstructured agent matchmaking: experiments in timing and fuzzy matching. In *SAC*, pages 300–305, 2002.

[50] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *International Semantic Web Conference*, pages 333–347, 2002.

[51] R. Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin, Madison, 2000.

[52] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high-throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, July 1998.

[53] R. Raman, M. Livny, and M. Solomon. Policy driven heterogeneous resource co-allocation with gangmatching. In *Proceedings of the Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC12)*, Seattle, WA, June 2003.

[54] P. Z. Revesz. Constraint databases: A survey. *Lecture Notes in Computer Science*, 1358:209–246, 1998.

[55] P. Z. Revesz. Safe datalog queries with linear constraints. In *CP*, pages 355–369, 1998.

[56] E. Di Sciascio, F. M. Donini, and M. Mongiello. Knowledge representation for matchmaking in p2p e-commerce. In *Atti del VIII Convegno dell'Associazione Italiana di Intelligenza Artificiale*, sep 2002.

[57] M. Solomon. The ClassAd language reference manual version 2.4, May 2004. http://www.cs.wisc.edu/condor/classad/refman/.

[58] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, pages 36–46, dec 1996.

[59] K. Sycara, K. Decker, and M. Williamson. Matchmaking and brokering. In *Proc. of the Second Int'l Conf. on Multi-Agent Systems (ICMAS-96)*, Dec 1996.

[60] K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous Agents and Multi-Agent Systems*, 5(2):173–203, 2002.

[61] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[62] G. Tonti, J. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok. Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder. In *International Semantic Web Conference*, pages 419–437, 2003.

[63] D. Trastour, C. Bartolini, and J. Gonzalez-Castillo. A semantic web approach to service description for matchmaking of services. In *SWWS*, pages 447–461, 2001.

[64] A. Uszok, J. Bradshaw, and R. Jeffers. Kaos: A policy and domain services framework for grid computing and semantic web services. In *iTrust*, pages 16–26, 2004.

[65] A. Uszok, J. M. Bradshaw, R. Jeffers, N. Suri, P. J. Hayes, M. R. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *POLICY*, pages 93–, 2003.

[66] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Policy terminology. RFC 3198, November 2001.