Using Argumentation Logic for Firewall Policy Specification and Analysis

Arosha K Bandara[1], Antonis Kakas[2], Emil C Lupu[1], Alessandra Russo[1]

1: Department of Computing, Imperial College London, London SW7 2AZ
2: Department of Computer Science, University of Cyprus, Cyprus
{bandara, ack, ecl1, ar3}@doc.ic.ac.uk

Firewalls are important perimeter security mechanisms that implement an organisation's network security requirements and can be notoriously difficult to configure correctly. Given their widespread use, it is crucial that network administrators have tools to translate their security requirements into firewall configuration rules and ensure that these rules are consistent with each other. In this paper we propose an approach to firewall policy specification and analysis that uses a formal framework for argumentation based preference reasoning. By allowing administrators to define network abstractions (e.g. subnets, protocols etc) security requirements can be specified in a declarative manner using high-level terms. Also it is possible to specify preferences to express the importance of one requirement over another. The use of a formal framework means that the security requirements defined can be automatically analysed for inconsistencies and firewall configurations can be automatically generated. We demonstrate that the technique allows any inconsistency property, including those identified in previous research, to be specified and automatically checked and the use of an argumentation reasoning framework provides administrators with information regarding the causes of the inconsistency.

1. Introduction

Firewalls are widely used perimeter security mechanisms that filter packets based on a set of configuration rules that are derived from the organisation's network security requirements. The rules are specified in priority order and are of the form:

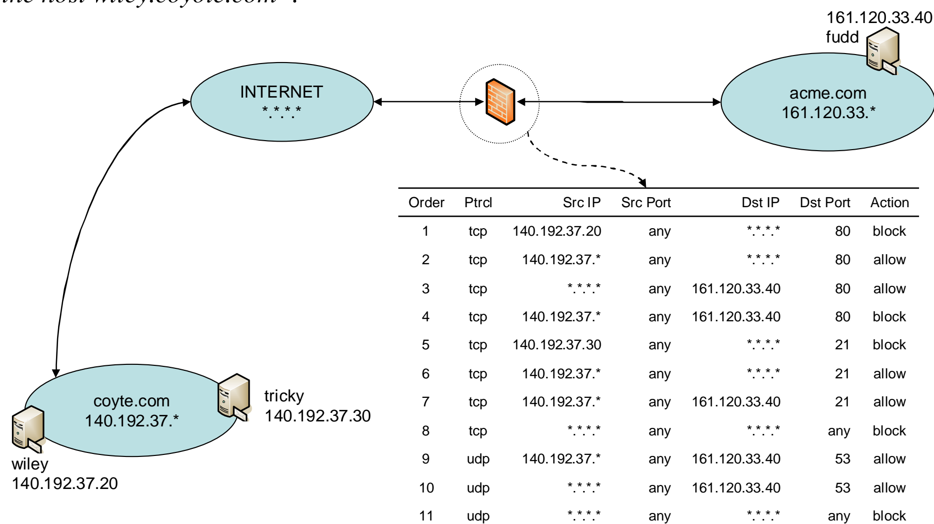
```
<order> : <action> if <network conditions>
```

where the <network conditions> identify a certain type of traffic, typically from one domain to another under some protocol, and the action field, <action>, typically takes the values "allow" or "deny" thus specifying if the traffic is to be allowed to flow or stopped. The semantics of the firewall policy is given operationally and it is crucially dependent on the total ordering of its rules. The ordering position of a rule is given by a (unique) number in <order> and for a given packet the firewall will check the rules in ascending order. The action field of the first rule whose network conditions are satisfied by the packet determines if it will be allowed or blocked. All subsequent rules are ignored.

In this paper we propose a technique for specifying security requirements within an argumentation based framework for Logic Programming with Priorities (LPP).

This allows us to specify and use high-level abstractions for network entities, e.g. protocols, applications, sub-networks. It also allows us to specify relative ordering between security requirements. The framework supports automatic generation of firewall configuration rules that satisfy the requirements including the relative ordering. Additionally, we demonstrate that the framework can detect a range of inconsistencies, including the anomaly types identified by Al-Shaer and Hamed [1], and also perform anomaly resolution.

Figure 1 shows an example system taken from [1] where a firewall is used to protect hosts in an enterprise network (acme.com) from malicious network traffic together with the set of rules that control the behaviour of the firewall. In this example, Rules 8 and 11 implement the default security requirement that all traffic should be blocked unless there is a specific requirement to allow specific types of traffic. These exception cases to the default requirement are implemented by specifying firewall policy rules that have a higher priority ordering than the default policy rule. For example, Rule 7 implements the requirement to *"allow FTP connections from hosts in the coyote.com network to the host ftp.acme.com"* and Rule 1 and 2 to *"allow all HTTP requests from coyote.com to acme.com except those from the host wiley.coyote.com"*.



Order	Ptrcl	Src IP	Src Port	Dst IP	Dst Port	Action
1	tcp	140.192.37.20	any	*.*.*.*	80	block
2	tcp	140.192.37.*	any	*.*.*.*	80	allow
3	tcp	*.*.*.*	any	161.120.33.40	80	allow
4	tcp	140.192.37.*	any	161.120.33.40	80	block
5	tcp	140.192.37.30	any	*.*.*.*	21	block
6	tcp	140.192.37.*	any	*.*.*.*	21	allow
7	tcp	140.192.37.*	any	161.120.33.40	21	allow
8	tcp	*.*.*.*	any	*.*.*.*	any	block
9	udp	140.192.37.*	any	161.120.33.40	53	allow
10	udp	*.*.*.*	any	161.120.33.40	53	allow
11	udp	*.*.*.*	any	*.*.*.*	any	block

Fig. 1. Example network and associated firewall policy rules [1]

In this example, the translation of these requirements into the rules shown is done manually and depends on administrators' knowledge of the low-level network topology and protocols and also having the expertise to assign the correct priority order to the rules. This method of policy specification has the added disadvantage that no link is maintained between the security requirements and the policy rules that implement them. This makes policy specifications hard to understand and it is easy for the administrator to make errors, particularly when dealing with large distributed systems that involve many networks, hosts and applications. Some firewall solution vendors have made an attempt to support high-level abstractions by mapping named

traffic classes to low-level properties such as host IP addresses, port numbers and protocols [2]. However, this process involves a significant amount of manual effort on the part of the administrator and the tools provided do not maintain any link between the security requirements and the underlying policy rules.

Another shortcoming with existing approaches to firewall policy specification is the limited support for automated analysis that verifies that the specification satisfies desired security properties and does not contain any inconsistency. Work done by Al-Shaer et al. goes some way to addressing this problem by identifying a number of inconsistency types (or *anomaly types*) and defining an algorithm for detecting the presence of these inconsistencies [1]. Whilst this technique has been extended to detect inconsistency in complex scenarios that involve distributed firewalls, it only detects a fixed set of inconsistency types [3]. Additionally, given that the analysis algorithm operates on the low-level firewall policy rules, it is not able to provide any information about the reasons for an anomaly to exist.

The rest of this paper is organised as follows. In the next section we present information regarding the capabilities of LPP framework together with examples of how the notation can be used to specify network abstractions and security requirements. In section 3 we present the different types of analysis supported by our technique followed by a discussion of our work in section 4. We describe how our work compares with related research in the field in section 5 before presenting our conclusions and plans for further work in section 6.

2. Security Requirements and the Argumentation Framework

One of the objectives of our work is to provide administrators with the ability to specify their security requirements using high-level abstractions that are closer to their natural specifications. We wish to do this in the context of a formal reasoning framework that supports the prioritised ordering of firewall rules and also provides automated analysis capabilities. In this section we present a formal language based on an argumentation framework that is capable of representing background information regarding the network, hosts and traffic types together with network security requirements and relative priorities between rules.

Argumentation has been shown to be a useful framework for formalizing non-monotonic reasoning and other forms of reasoning [4-7]. In general, an argumentation framework is a pair <T,*A*> where T is a theory in some background (monotonic) logic, equipped with an entailment relation, $\models$, and *A* is a binary relation on the subsets of T. These subsets of T form the **arguments** of the framework and *A* is a non-symmetric **attacking relation** between arguments. For any two arguments A1 and A2 we say that **A1 attacks A2** when (A1,A2) belongs to the attacking relation *A*. In this context, an argument A1 attacks A2 if, given the same background knowledge, A1 supports a conclusion that is incompatible with a conclusion supported by A2 and A1 is defined to be stronger than A2. Argumentation reasoning is given through the notion of an *admissible argument*, i.e. an argument that counterattacks another argument. The formal definition of the argumentation framework is presented in [4, 6].

2. 1 Representing Security Requirements and Firewall Rules

In the specific argumentation reasoning framework we use in this paper, a theory T is represented in the background logic (L, ⊨), where the language L consists of (extended) logic programming rules of the form:

```
Name: L ← L1, . . . , Ln, (n ≥ 0).
```

Here, $L, L_1, \ldots, L_n$ are positive or negative literals. A negative literal is a literal of the form ¬A, where A is an atom. As usual in Logic Programming a rule containing variables is a compact representation of all the ground rules obtained from this under the Hebrand universe. Each ground rule has a unique (parametric) name, *Name*, given at the front of the rule. Using this notation we can specify a security requirement by defining a rule with name `req(…)` that associates a given action with packets that match the source, destination and traffic type. For example, the requirement to *"allow HTTP requests from the coyote.com network to web servers in the acme.com network"* would be defined as follows:

```
req(allow_http_coyote, allow, Pkt):
action(allow, Pkt) ←
  packetFrom(coyote, Pkt), packetTo(Server, Pkt),
  property('web', host, Server), traffic(http, Pkt).
```

The packet terms in the above rule are defined using 5-tuples of the form `pkt(Protocol, SourceIP, SourcePort, DestIP, DestPort)`. In the above definition, the `packetFrom(…)` and `packetTo(…)` predicates are used to map the name of a source or destination entity to the appropriate IP address fields of the packet. In the above definition, the `packetFrom(…)` and `packetTo(…)` predicates are used to map the name of a source or destination entity to the appropriate IP address fields of the packet. These predicates are defined as follows:

```
pktSource(SrcIP):
packetFrom(From, pkt(_,SrcIP, _, _, _)) ← ipaddr(From, SrcIP).

pktDest(DstIP):
packetTo(To, pkt(_, _, _, DstIP, _)) ← ipaddr(To, DstIP).
```

The `ipaddr(…)` predicate is used to define background information regarding the network, namely the IP address of a given network entity. This is described in more detail in the next section.

The overall theory T is separated into two parts: the **basic** part and the **strategy** part. The basic part contains rules (of the form given above) whose conclusions, L, are any literal except the special predicate, `prefer(…)`, which is the only predicate that can appear in the conclusion of rules in the strategy part. Hence rules in the strategy part take the special form

```
Name: prefer(rule1, rule2)← L1, . . . ,Ln, (n ≥ 0).
```

where rule1 and rule2 are the names of any other two rules in the theory. A rule of this form then means that under the conditions $L_1, \ldots, Ln$, the rule with name, rule1, has priority over the rule with name, rule2. The role of this priority relation is therefore to encode locally the relative strength of (argument) rules in the theory. The priority relation `prefer(…)` is required to be irreflexive. The rules `rule1` and `rule2` can themselves be rules expressing priority between other rules and hence the framework allows higher-order priorities.

We can use the `prefer(…)` predicate to defined security requirements express a precedence relationship between two simpler requirements. For example, the administrator might specify a requirement to *"block any traffic except HTTP requests to fudd.acme.com"*. This type of requirement can be composed from *"block any traffic"* and *"allow HTTP requests to fudd.acme.com"* together with the addition of a rule that makes the latter requirement take precedence. The two simple requirements would be specified as follows:

```
req(block_any, block, Pkt):
action(block, Pkt) ←
  packetFrom(any, Pkt), packetTo(any, Pkt), traffic(any, Pkt).

req(allow_http_fudd, allow, Pkt):
action(allow, Pkt) ←
  packetFrom(any, Pkt), packetTo(fudd, Pkt), traffic(http, Pkt).
```

This is followed by the precedence relationship between these requirements using the `prefer(…)` predicate:

```
 order(allow_http_fudd, block_any):
prefer(req(allow_http_fudd, allow, Pkt)),
       req(block_any, block, Pkt) ).
```

In addition to representing security requirements, the notation described can be used to specify legacy firewall rules, denoted by the term `fwr(Order, Action, Pkt)`. For example, the first rule shown below represents Rule 9 given in Figure 1. Notice that we use finite domain constraints to specify IP address and port ranges:

```
fwr(9, allow, pkt(udp,ip(140,197,37,D),SP,ip(161,120,33,40), 53)):
action(allow, pkt(udp,ip(140,197,37,D),SP,ip(161,120,33,40), 53))
          ← D in 0..255, SP in 1..65536.

 order(N1, N2):
prefer(fwr(N1, A1, Pkt), fwr(N2, A2, Pkt)) ← N1 < N2.
```

The second rule shows how we can use the `prefer(…)` predicate to specify the ordering of legacy firewall rules. In this fashion our formal framework can combine the requirements specifications described above with legacy firewall rules.

2.2 Representing Background Information

We can separate out an auxiliary part, T_0, of a given theory, T, from which the other rules can draw background information in order to satisfy some of their conditions. The reasoning of the auxiliary part of a theory is independent of the main argumentation-based preference reasoning of the framework and hence any appropriate logic can be used. In the context of this paper, we can use this feature to specify subnets, hosts and traffic types in a network using the following three predicates:

```
network(Name, [Properties]).
host(Name, [Properties]).
ipaddr(Name, ip(A, B, C, D)).
```

The `network(…)` predicate defines a named network (e.g. acme.com, coyote.com etc) together with a list of associated properties (e.g. wireless, WEP, etc.). Similarly, the `host(…)` predicate defines a host name together with a list of properties associated with that host. Finally, the `ipaddr(…)` predicate associates a particular host (or

network) with an IP address (or address range). The `ip(…)` function has four arguments that correspond to each byte of a 32-bit IP address. Using these predicates, the 'acme.com' network and 'fudd.acme.com' host in the example (Figure 1) would be specified as follows:

```
network(acme, ['acme.com', 'wired']).
ipaddr(acme, ip(161, 120, 33, D)) ← D in 0..255.

host(fudd, ['fudd.acme.com', 'web', 'ftp', 'dns']).
ipaddr(fudd, ip(161, 120, 33, 40)).
```

We use the finite domain constraint 0..255 to specify the range of values for the 'acme' network. Notice that each rule in the above definitions is prefixed with a parameterised name, which becomes part of the arguments for the answer derived if the rule forms part of the theory that supports a given goal. For example, if we query the system for the available networks, each answer will be accompanied by an argument set that includes the term `network(…)`, identifying the network rule that defines each network. We can also specify a auxiliary predicate, `property(…)`, which can be used to identify the network elements that have a given property:

```
property(Prop, network, Element)
          ←  network(Element, [Props]), member(Prop, Props).

property(Prop, host, Element)
          ←  host(Element, [Props]), member(Prop, Props).
```

In the above definition, the member(Property, Properties) predicate holds if the property denoted by the first parameter is a member of the list denoted by the second. The `property(…)` predicate can also be used to express higher-level, composite properties. For example, the notion that all Linux hosts on a wired network are considered to be secure can be expressed as follows:

```
property(secure, host, Element)
          ←  property(wired, network, Network),
             property(Network, host, Element),
             property(linux, host, Element).
```

Finally we define a predicate, `traffic(Name, Pkt)` that associates the protocol and ports fields of an IP packet with a given type of traffic. This predicate can be also be used to define ranges of ports. For example, we can define the following rules to specify an application called 'http' which matches TCP packets from any non-reserved port (1024-65536) to port 80; and a generic application called 'any' which matches packets containing any port number and protocol:

```
traffic(http, pkt(Prtcl, SrcIP, SP, DstIP, DP)) ←
  Prtcl=tcp, SP in 1024..65536, DP = 80.

traffic(any, pkt(Prtcl, SrcIP, SP, DstIP, DP))) ←
  (Prtcl=tcp; Prtcl=udp), SP in 1..65536, DP in 1..65536.
```

It is important to note that specifying this background information is a one-time task that can be automated using host/service discovery tools. Of course the specification will have to be updated if there are any changes in the system, but this process can also be automated.

In addition to background information regarding the network, the auxiliary part of our theory also contains the definition of what constitutes a conflict (over and above the standard conflict of classical negation, i.e. between an atom, A and its negation

¬A). This is given through the definition of an auxiliary predicate, `complement(…)`, which is of the form:

```
complement(L1, L2) ← B.
```

stating that literals L1 and L2 are conflicting under some (auxiliary) conditions B. Typically, the conditions B are empty and the definition of the `complement(…)` predicate is kept simple. Also we will assume that the conditions of any rule in the theory do not refer to the predicate `prefer(…)` thus avoiding self-reference problems. Note also that the definition of `complement(…)` always includes that any ground atom, `prefer(rule1, rule2)`, is incompatible with the atom `prefer (rule2, rule1)` and vice-versa. In the context of firewall policies, we would define the actions *allow* and *block* to be complementary using the following rule:

```
complement(action(allow, _), action(block, _)).
```

The logical framework for argumentation and preference reasoning described here has been realised in the GORGIAS tool developed at the University of Cyprus [8] and has been used to implement the examples and generate the results presented in this paper. This tool provides a query, `prove([L1, L2, …, Ln], Args)`, which generates the set of admissible arguments, Args, that support the conjunction of terms L_1, …, L_n for a given theory. In order to support the analysis of security requirements and firewall policies, we define the following auxiliary query to determine if a particular packet will be allowed or blocked by a firewall together with the rule (or requirement) that causes this decision and the supporting arguments:

```
packet_action(Action, Pkt, Rule, Args) ←
  prove([action(Action, Pkt)], Args), member(Rule, Args),
  (Rule=requirement(R, Action, Pkt); Rule=fwr(N, Action, Pkt)).
```

3. Analysing Firewall Policies

As a network grows, the task of managing the network security policies quickly becomes unwieldy. Therefore it is very important to provide administrators with support to analyse the policy specification and ensure that desired properties hold. These analysis tasks can be divided into the following categories:

1. **Anomaly Detection:** Analysing the policy specification for potential anomalies.
2. **Property Checking:** Performing "what-if" analysis to determine if a given class of traffic will be forwarded or blocked. For example, *"Which packets are allowed to reach the host fudd.acme.com?"* This type of query can also be used to verify that the policy specification satisfies desired behaviour.
3. **Anomaly Resolution:** Determining the correct ordering of policy rules to ensure that anomalies are avoided (i.e. ensuring that rules related to exception cases are given a higher precedence than general rules)

3.1 Anomaly Detection

Al-Shaer et al [1] have identified four firewall policy anomaly types – shadowing, generalisation, correlation and redundancy and here we show how these anomalies

can be detected using the argumentation logic framework. From the description of the various anomaly types it is clear that the key determinants of an anomaly is whether the packets that match a rule are a subset (or superset) of the packets matched by another rule; and the relative ordering of the rules. For example, rule R2 is said to be shadowed by R1 if the rules specify incompatible actions, R1 has preference over R2 and every packet that matches R2 is matched by R1. In order to detect this type of anomaly we define the following rule:

```
anomaly(shadow, R1, R2, Pkt1)←
          packet_action(A1, Pkt1, R1, _),
          complement(action(A1,_,_), action(A2,_,_)),
          packet_action(A2, Pkt2, R2, _),
          match(subset, Pkt1, Pkt2).
```

The above rule identifies a requirement R1 where the matching packets, Pkt1, are a subset of the packets that match another requirement R2 and R2 defines an incompatible action. The preference reasoning capabilities of the LPP framework ensures that the above query identifies rules derived from R2 that have higher precedence than rules derived from R1. Rules that participate in a generalisation anomaly would cause a shadow anomaly if their relative order was reversed. We use this property to define the following rule to detect this type of anomaly:

```
anomaly(generalisation, R1, R2, Pkt2)←
          packet_action(A1, Pkt1, R1, _),
          complement(action(A1,_,_), action(A2,_,_)),
          packet_action(A2, Pkt2, R2, _),
          match(subset, Pkt2, Pkt1).
```

The above definition identifies a policy rule derived from requirement R2 that takes precedence over a rule derived from requirement R1, where the packets matched by R2 are a subset of those matched by R1 and the actions of R1 and R2 are complementary.

Correlation anomalies occur when two rules with complementary actions match the same packets, and the rules are not part of a shadowing or generalisation anomaly. These can be detected using the following rule:

```
anomaly(correlation, R1, R2, Pkt)←
          packet_action(A1, Pkt, R1, _),
          complement(action(A1,_,_), action(A2,_,_)),
          packet_action(A2, Pkt, R2, _),
          ¬ anomaly(generalisation, R1, R2, _),
          ¬ anomaly(generalisation, R2, R1, _),
          ¬ anomaly(shadow, R1, R2, _).
```

Redundancy anomalies differ from the other types in that they involve rules that specify the same action. We define the following rule to detect this type of anomaly:

```
anomaly(redundant, R1, R2, Pkt1)←
  packet_action(A, Pkt1, R1, _), packet_action(A, Pkt2, R2, _),
  R1 \== R2, match(subset, Pkt1, Pkt2).
```

Using these rules, we can detect all the anomalies in a specification using a single high-level query. For example, performing such a query on the example system shown in Figure 1 would generate the following result:

```
?- findall(Type-(R1, R2), anomaly(Type, R1, R2, _), List).
List =  shadow-(deny_coyote_http_fudd,allow_coyote_http)
        shadow-(deny_coyote_http_fudd,allow_http_fudd)
        generalise-(deny_wiley_http,allow_coyote_http)
```

```
...
generalise-(allow_udpdns_fudd,deny_all)
correlated-(deny_wiley_http,allow_http_fudd)
correlated-(deny_tricky_ftp,allow_coyote_ftp_fudd)
redundant- (allow_coyote_ftp_fudd, allow_coyote_ftp)
redundant- (allow_coyote_udpdns_fudd,allow_udpdns_fudd)
```

3.2 Property Checking

In addition to checking for the anomaly types identified in the literature, the formal framework for firewall policy specification described in this paper is a general one that can be used to check if a specification satisfies other properties. For example, the administrator might wish to verify which packets are allowed to reach the host fudd.acme.com. This property would be checked by the following high-level query:

```
?- packet_action(allow, Pkt, Rule, Args), packetTo(fudd, Pkt).

      Rule = allow_coyote_http
    Packet = pkt(tcp, ip(140,192,37,D1), SP, ip(161,120,33,40), 80)
        D1 = 0..255, SP = 1024..65536
Arguments:
requirement(allow_coyote_http, allow, pkt(tcp, coyote, SP, any, 80)).
pktDst(any,ip(161,120,33,40)).
pktSrc(coyote,ip(140,192,37,D1)).
...
```

The arguments explain that a TCP packet from 140.192.37.*-port:1024-65536 to 161.120.33.40-port:80 is allowed because the requirement 'allow_coyote_http_allow' specifies that packets from the 'coyote.com' network to port 80 of any host should be allowed. Furthermore, the arguments show how the IP address and port ranges in the allowed packet match the IP addresses of 'coyote.com' and 'fudd'.

Notice that the use of the finite domain constraints for IP address and port ranges means that the query returns an expression that describes all the packets that are allowed to reach the host 'fudd'. The ability to consider the relative priorities between security requirements and also provide this type of coverage of the potential packet space when reporting results is possible because we are using a logic programming based approach that supports preference reasoning.

3.3 Anomaly Resolution

Of the anomaly types defined in the previous section, only redundancies and shadowing anomalies are considered to be errors. Of these, shadowing anomalies can be resolved by reversing the relative ordering of the two rules. This can be expressed in our framework using a 'higher-order' preference reasoning rule as follows:

```
resolve(shadow, R1, R2):
prefer(R1, R2) ← anomaly(shadow, R1, R2, _).
```

The above rule states that preference should be given to rule R1 over R2, i.e. the shadowed rule is given higher priority. Redundancy anomalies on the other hand can be resolved by ensuring the redundant rule has lower priority. This resolution process is specified in our formal framework as follows:

```
resolve(redundancy, R1, R2):
prefer(R2, R1) ← anomaly(redundant, R1, R2, _).
```

Here the `anomaly(…)` predicate holds if R1 is redundant to R2 and the `prefer(…)` predicate defines that R2 should take precedence over R1. In our framework, performing the resolution actions shown above will remove any redundancy and shadowing anomalies from the specification. Additionally, the decision to perform a particular resolution action will be explained with a set of arguments.

4. Discussion

In the study of the analysis of firewall policies we have shown specifically that the various types of anomalies in firewall policies, identified separately in the literature, can be captured naturally under the same and unified definition based on the standard notion of an admissible argument in Logic Programming with Priorities (LPP). This high level definition means (a) that we are more complete in capturing the notion of anomaly and (b) that our definitions remain invariant as we further develop the types of policy supported by the notation, e.g. as we consider extensions of policies for distributed firewalls. The high-level of expressivity of the LPP framework, particularly its ability to represent preference orderings which can be conditional on some background properties means that the formalism can accurately capture the behaviour of a firewall where policies are specified with an explicit priority order. The LPP framework can be used to detect all the anomaly types identified in the literature and also supports other types of property checking, thus allowing an administrator to verify the behaviour of a firewall that is controlled by a given set of requirements. Whilst we have yet to complete experiments on large policy sets, the complexity of the argumentation reasoning framework for the restricted type of theory described in this paper has been shown to be P-complete [9]. We are working to validate the scalability of our approach as part of our ongoing research efforts.

In addition to experimenting with larger policy sets, we also hope to work on more complex scenarios involving multiple firewalls in the network. In such a system, where policies will be distributed across the network the problem of the existence of anomalies is more severe as there are more possibilities for conflicts to occur. We can have situations where one component decides to accept traffic whereas another component decides to deny it. For example, an upstream firewall blocking a traffic that is permitted by a downstream firewall is a type of inter-firewall shadowing anomaly. In a "classical" approach to anomaly detection the definition of this anomaly requires a detailed (and somewhat ad hoc) examination of the pairs of rules from the two firewalls. In our declarative approach this anomaly falls under the same definition given above.

5. Related Work

Work presented by Wool et al., proposes a high-level language for specifying network information and firewall policies that allows firewall configuration to be performed at an abstraction level that is closer to high-level programming. This work has led to the development of a number of tools that support offline firewall policy analysis and

management [10]. However, the analysis process does not detect specific anomaly types such as shadowing and redundancy.

Uribe and Cheung have developed a technique for automating the analysis of firewall and network intrusion detection systems that uses constraint logic programming to model the networks and policies [11]. The use of finite domain constraints to specify IP address and port ranges means that the analysis process covers all IP address and port combinations for potential problems. However, the technique does not support specification of explicit priorities between firewall policy rules and the tool does not provide administrators with any explanation to support the analysis results generated.

Al Shaer et al. and Yuan et al., have focussed on tools and techniques for analysing legacy firewall policies for networks with centralised and distributed firewalls [1, 3, 12]. We use the classification of anomalies into the types: shadow, correlation, generalisation and redundancy anomalies presented in [1] to specify the analysis rules used in the framework presented in this paper. One shortcoming of their approach is the dependence on legacy firewall policies in order to perform anomaly detection and resolution. In contrast, our approach allows network security requirements to be specified using high-level notations whilst still being capable of a range of analysis tasks such as anomaly detection, resolution and property checking. Additionally by using an argumentation reasoning framework, our approach has the advantage that the administrator is given an explanation of the analysis results and resolution actions.

6. Conclusions and Future Work

We have presented an approach to specifying network security requirements that is based on Argumentation for Logic Programming with Priorities (LPP). The use of logic programming allows the specification to include high-level abstractions such as networks, hosts, traffic types and their associated properties. This means that administrators can specify their network security requirements in more familiar terms, without having to know the exact IP address and port ranges for a given traffic flow. We have shown that the technique is capable of performing a range of analysis tasks, from detecting the firewall anomaly types identified in the literature to performing more general property checking and conflict resolution. The use of LPP allows preferences to be encoded, thus allowing complex reasoning over the relative priorities between rules. Additionally, the encoded preferences can be conditional on arbitrary system properties, an approach that allows greater flexibility than simple assigned priorities between rules. Also, because LPP is implemented using argumentation reasoning, the results of performing queries are enhanced by explanations containing the rules that support a particular conclusion. This information is particularly helpful to the user in understanding the reason for a traffic flow to be allowed or blocked by the firewall. The current implementation of the technique presented in this paper focuses on security requirements specification for firewalls. However, given an appropriate formalisation of the underlying system, the use of LPP can be extended to other application domains, such as network QoS management.

Our system is implemented using the GORGIAS tool running in a standard Prolog environment. Given a formal description of the network elements and security

requirements, it provides support for checking general properties, including checking for the presence of the anomaly types identified in the literature, and also supports anomaly resolution. At present we are focused on extending the tool to provide automated generation of 'anomaly-free' operational firewall policies. Additionally are developing a GUI that will shield the administrator from the underlying formal notation, providing an interface that simplifies the process of defining their network security requirements and analysing them for consistency. Our future work also includes extending the formal notation to include information required to specify and analyse network security requirements that are implemented using distributed firewalls.

Acknowledgements

We acknowledge financial support for this work from the EPSRC (Grant Numbers - GR/R31409/01, GR/S79985/01 and GR/T29246/01) and IBM Research.

References

[1] E. S. Al-Shaer and H. H. Hamed. "Firewall Policy Advisor for Anomaly Doscovery and Rule Editing." In Proceedings of 8th IFIP/IEEE International Symposium on Integrated Network Management, Colarado Springs, CO, IEEE, March 2003.

[2] Cisco. "Cisco PIX Firewall Configuration White Paper (DOCID: 68815), http://www.cisco.com/warp/public/707/ezvpn-asa-svr-871-rem.pdf", Cisco Inc, 2006.

[3] E. S. Al-Shaer and H. H. Hamed. "Discovery of Policy Anomalies in Distributed Firewalls." In Proceedings of 23rd IEEE Communications Society Conference (INFOCOM), Hong Kong, IEEE, March 2004.

[4] P. M. Dung (1995). "On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games." Artificial Intelligence(77): 321-357, 1995.

[5] A. Bondarenko, P. M. Dung, R. A. Kowalski, and F. Toni (1997). "An abstract argumentation theoretic approach to default reasoning." Artificial Intelligence 93: 63-101, 1997.

[6] A. Kakas, P. Mancerella, and P. M. Dung. "The acceptability semantics for logic programs." In Proceedings of 11th International Conference on Logic Programming, Santa Marherita Ligure, Italy, 1994.

[7] H. Prakken and G. Sartor. "A system for defeasible argumentation, with defeasible priorities." In Proceedings of International Conference on Formal and Applied Practical Reasoning, Springer-Verlag, LNAI 1085, 1996.

[8] Gorgias. "Argumentation and Abduction, http://www2.cs.ucy.ac.cy/~nkd/gorgias/",

[9] Y. Dimopoulos, B. Nebel, and F. Toni (2002). "On the Computational Complexity of Assumption-based Argumentation for Default Reasoning." Artificial Intelligence 141: 57-78, 2002.

[10] A. Mayer, A. Wool, and E. Ziskind (2006). "Offline firewall analysis." International Journal on Information Security 5(3): 125-144, 2006.

[11] T. E. Uribe and S. Cheung. "Automatic Analysis of Firewall and Network Intrusion Detection System Configurations." In Proceedings of ACM Workshop on Formal Methods in Security Engineering, Washington, DC, ACM Press, October 2004.

[12] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. "FIREMAN: a toolkit for FIREwall Modeling and ANalysis." In Proceedings of IEEE Symposium on Security and Privacy, Oakland, CA, May 2006.