Specifications of A High-level Conflict-Free Firewall Policy Language for Multi-domain Networks

Bin Zhang, Ehab Al-Shaer, Radha Jagadeesan, James Riely, Corin Pitcher
School of Computer Science, Telecommunications and Information Systems
DePaul University,
{bzhang, ehab, rjagadeesan, jriely, cpitcher}@cs.depaul.edu

ABSTRACT

Multiple firewalls typically cooperate to provide security properties for a network, despite the fact that these firewalls are often spatially distributed and configured in isolation. Without a global view of the network configuration, such a system is ripe for misconfiguration, causing conflicts and major security vulnerabilities.

We propose *FLIP*, a high-level *firewall configuration policy language* for traffic access control, to enforce security and ensure seamless configuration management. In FLIP, firewall security policies are defined as high-level service-oriented goals, which can be translated automatically into access control rules to be distributed to appropriate enforcement devices. FLIP guarantees that the rules generated will be conflict-free, both on individual firewall and between firewalls. We prove that the translation algorithm is both sound and complete.

FLIP supports policy inheritance and customization features that enable defining a global firewall policy for large-scale enterprise network quickly and accurately. Through a case study, we argue that firewall policy management for large-scale networks is efficient and accurate using FLIP.

Categories and Subject Descriptors: F.m [Theory of Computation]: Miscellaneous

General Terms: Languages, Security

Keywords: firewall, policy language, conflicts free

1. INTRODUCTION

As the size and complexity of enterprise network has increased, traditional manual configuration of security devices has proved inadequate to bridge the gap between the high-level security requirement and low-level device implementations. Security requirements are defined by a set of policies scattered over different security devices and environments [15]. Varied network security devices have different goals and are typically configured in isolation. Managing the security policy in such a large and heterogeneous environment is a difficult task: the final security policy is the combination of the local policy of each device. Conflicts between the configurations of different devices are common [11]. How can one resolve these conflicts to produce non-conflicting local policies that satisfy the desired global security requirements?

Virtual networks [5, 7] are usually created to enable sharing and aggregation of resources of various network domains according to user-defined security requirements. Without a global view of the network security devices configuration, such a system is ripe for misconfiguration, causing problems that range from unnecessary/redundant processing to major security vulnerabilities. To enforce security and ensure seamless device configuration management, security requirements must be defined as high-level service-oriented goals, rather than device related configuration rules, that can then be translated into rules distributed in security devices automatically. Many critical but cumbersome tasks including policy definition, rule distribution and enforcement, and conflicts resolution must also be automated. Traditional work on firewalls [17, 6] has focused on nodes and enforcement mechanisms rather than global network protection, policy coordination and validation.

In this paper, we propose FLIP, a high-level firewall configuration policy language for traffic access control. FLIP addresses the security policy management problem in large-scale heterogeneous network. FLIP is a high-level language that decouples the policy from network topology and hides the configuration details related to different vendors and devices. The FLIP language is easy to understand and use, and can be compiled to conflict-free device configurations.

The rest of this paper is organized as follows. In Section 2, we describe the syntax and semantics of FLIP, as well as the the design principles that guided us to the language. In Section 3, we describe the algorithm that translates FLIP into lower-level policies. We prove that the translation exactly preserves the high-level semantics of FLIP. We present a case study in Section 4, demonstrating the utility of the language. We discuss the scalability and performance of rule translation Section 5. Related and future work are discussed in the last two sections.

2. HIGH LEVEL FIREWALL POLICY LANGUAGE

We believe that a high level security policy language should have the following features to meet the requirements of defining and managing security policies in a large-scale enterprise network:

SACMAT'07, June 20-22, 2007, Sophia Antipolis, France.

```
<domain_def> ::= "domain" <domain_name> "=["
                 <domain> { "," <domain> }
                 "]"
    <domain> ::= <ip_address> | <ip_range>
                 | <domain_name> { <operator> <domain> }
   <service> ::= <protocol> ".["
                 <predicates>
                 "]"
  <protocol> ::= "tcp" | "udp" | "icmp" | "ip"
<predicates> ::= <predicate> { ("," | "OR") <predicate> }
 <predicate> ::= <field_name> <operator> <value>
```

Figure 1: Domain and Service definition syntax.

```
 <group> ::= "policy_group" <name> [ "extends" <name> ] "{"
             [ "incoming:" <block> ]
             [ "outgoing:" <block> ]
             "}"
 <block> ::= [ "enforce" | "restrict" ]
             ( <service_name> | <service> ) "{"
             { <rule> }
             "}"
  <rule> ::= [ "enforce" | "restrict" ]
             ( "allow" | "deny" )
             <domain> { "except" <domain> }
```

Figure 2: Policy group definition syntax.

- Service-oriented: The language should focus the user's attention on the security requirements of network services rather than low-level traffic details as in the rule-oriented approach.
- Modular and reusable: The language objects are self contained and can be reused and extended to scale to large networks.
- Rule order independent: The user should be able to define the security policy in any order without worrying about introducing conflicts.
- Conflict free: The compiled policies should not have conflicts in a single device (intra-policy) or across multiple devices (inter-policy).

In this section we present in detail each of the basic constructs comprising FLIP and show the above principles are embodied in the syntax and semantics of FLIP.

2.1 Key Language Constructs and Grammar

FLIP includes constructs to describe *services*, *rules*, *policy groups* and *domains*. Rules define the action performed on the traffic flow of specific network service that satisfy the rule conditions. A policy group is a logical unit which aggregate related rules together and can be applied on domains. The grammars and relationships of these constructs are detailed below.

Domain.

A domain is a logical unit which contains the network addresses of entities (workstations, servers and network devices) that share the same security requirements. The entities in one domain can come from different physical subnets. A domain can be defined by a set of IP ranges, IP addresses with wildcard or host names. Also, one domain can be constructed by combining other domains. A domain can be viewed as a special set contains only network address, so we can apply the set operations on domain: intersection (`*`), union (`+`), subtraction (`-`). The syntax of domain definitions is shown in Figure 1, in EBNF. As usual square braces indicate optional items and curly braces indicate potentially empty repetition. The following example defines a domain *students*:

```
domain students = [140.192.90.1-140.192.95.5];
```

Service.

A service is defined as the combination of a protocol name and a set of properties associated with that protocol. Each property of a protocol is a predicate which is defined by the field name in that protocol header, the operator and value of that field. The field names in common protocol headers have been predefined in FLIP and the supporting protocols can be extended if needed in the future. Predicates can be linked together using the logical operator *AND* and *OR*, where the comma represents *AND*. The syntax of service definition is shown in Figure 1.

For example, `tcp.[port = 80]` means `http` traffic. One service can represent multiple traffic flows in the network as long as those traffic flows can satisfy the conditions defined in the properties set. For example, `tcp.[port > 2045, port < 3078]` represents all tcp traffics with destination port between 2045 and 3078. We can define the yahoo instant messaging (*yahoo_msg*) and Bit Torrent [3] (*torrent*) service as follow:

```
service yahoo_msg = tcp.[port=5050],
        torrent = tcp.[port >= 6881, port <= 6999];
```

Policy Group.

A policy group is a aggregation and abstraction of detail rules which are related to a set of entities (domains, hosts). A policy group consists of a set of service blocks. Each service block consists of rules associated with that specific network service. The rules in each service block are not complete firewall rules which contains all the 5 tuples (source address, source port, destination address, destination port, and protocol). For incoming traffics, the destination addresses are undefined, and for outgoing traffics, the source addresses are undefined. So in FLIP, a policy group should be applied on a policy target (domains, hosts) to complete the rules. The policy targets shared the same policy group can receive or send out the same types of traffic. In each policy group, the incoming and outgoing services are organized int two service groups. The syntax of policy group definition is shown in Figure 2.

The following example shows the definition of a simple policy group which block the yahoo instant messaging and Bit Torrent downloading but allow access to the internet.

```
policy_group student_policy {
  incoming:
    yahoo_msg { deny any }
    torrent { deny any }
  outgoing:
    http { allow any }
}
```

Apply Policy Group on Domains.

In FLIP, we allow multiple policy groups to be applied to a single domain. The syntax is shown in Figure 3. We can apply the policy group `student_policy` on domain `students` as follow:

```
<apply_block> ::= "apply" <policy_set> "on" <domain_set>
 <policy_set> ::= <policy_group_name>
                  { "," <policy_group_name> }
 <domain_set> ::= <domain_name>
                  { ("+"|"-"|"*"|",") <domain_name> }
```

Figure 3: The syntax of applying policy groups on domains.

```
apply student_policy on students
```

Policy Group Hierarchy and Inheritance.

In order to increase the modularity and reusability of the policy group, we organize policy groups hierarchically. Every policy group specifies its parent group, similar to the superclass specification in common object-oriented languages. If no parent group is specified, the default virtual policy group is assigned as the parent group. FLIP only allows single inheritance: each policy group can extends from only one parent group. This restriction enforces a tree structure on policy groups, making inheritance relationships between policy group clear and helping users locate sources of conflict.

Restrict and Enforce Rules.

There are two special types of rules in FLIP: *restrict* and *enforce*. Restrict rules are visible only in the current policy group and can not be inherited by child groups. Restrict rules are used to define special policy which should only be applied to special domain. Restrict rules can also be used to define policies for those services in which parent and child have different policy. Enforce rules can not be overwritten by child policy group, which are used to define those policies the administrator want to be enforced across multiple domains without any violation. Enforce rules must be inherited by child group.

2.2 Conflict Detection and Resolution

An extensive study [2, 8] shows that various conflicts can happen between rules in single or distribute firewalls. One of the most important objectives of FLIP is to help user design and enforce security policies without introducing conflicts. The policies defined by FLIP should exactly reflect the security requirements without any ambiguity. FLIP handles the conflicts from the following three perspectives:

Conflict in Single Policy Group.

In order to prevent conflict existence in policy group, FLIP has two constraints for policy group:

- The rules in each service block must be disjoint. This means that there should be no overlapping between the source or destination address between different rules for the same service.
- The services in incoming or outgoing traffic block must be totally disjoint. For example, `tcp.[port = 80]` service can not coexist in the same incoming traffic block with service `tcp.[port < 1024]`.

The first restriction guarantees no conflicts between rules in the same service block; the second, between services in same direction. These two constraints ensure that no conflicts can occur between rules in same policy group.

Conflicts Between Parent and Child Policy Group.

Conflicts can be introduced when parent and child group have different rules for same services. Whereas the high level rules defined in the FLIP language are order-independent, the low-level rules generated during translating are order-sensitive. FLIP resolves the conflicts between parent and child policy group by adjusting the order of low level rules for child group generated by FLIP rule translation algorithm. Conflicts between parent and child are resolved assuming that the child group specifies a more detailed security policy which further reflect the user's objective. So FLIP gives the rules in child group higher priority than the rules in parent group, unless they conflict with the `enforce` rules of the parent: these rules may not be overridden by child policies. The high priority of rules in child group and enforced rules in parent group is reflected by the rules order in the generated low level rules. The translation is explained in section 3.3.

The following example shows the conflicts in single policy group and between parent and child policy group. We now define a new policy group `dom_std_policy` which contains the security policy for students live in university dormitory. The security objectives are the following: 1) allow students use yahoo instant messaging, 2) block the online game *World of Warcraft* [20](tcp, port=3724), 3) block web proxy cache squid (tcp, port 3128) because it can also be used by Trojans. 4) allow windows remote desktop(tcp, port=3389) from network 140.192.*. In order to block both *World of Warcraft* and *squid*, the administrator in dormitory choose the block all tcp traffic use port between 3100 and 3800. The policy is shown below:

```
policy_group dom_std_policy extend student_policy {
  incoming:
    yahoo_msg { allow any }
    tcp.[port>=3100, port < 3800] { deny any }
    tcp.[port=3389] { allow 140.192.* }
}
```

We can easily see that there is a conflict between blocking all tcp service using port from 3100 to 3800 and allow windows remote desktop. Also, there is another conflict between parent and child group for service *yahoo_msg*

Conflicts Between Policy Groups Applied on Same Domain.

Different policy groups may have different policies for same service. Conflicts can be introduced when one domain (or subdomain) is applied with more than one policy groups. This situation can happen under one of the following two scenarios:

- If two or more policy groups must be enforced in a single domain, those policy groups are explicitly applied on that domain.
- When two or more domains share some common network addresses and each domain is applied with different policy group, the common part is implicitly applied to more than one groups.

For the first scenario, FLIP resolves the conflicts between policy groups based on the sequence in which these policy groups are applied on the domain. The earlier policy groups applied first are given higher priority. For example, `apply p1, p2 on D` specifies that both policy group `P1` and `P2` should be applied on domain `D`, and that `P1` should be given higher priority. If conflicts exist, the rules in high priority policy group overwrite the rules in low priority policy

Algorithm 1 policyTranslation

```
1: if exist conflicts then
2:    stop and send conflict reports
3: else
4:    PGRules()
5:    put all subdomain which has been assigned with more
      than one policy group in D_many.
6:    put all domain and subdomain with only one policy
      group in D_one
7:    if D_many ≠ ∅ then
8:       for each D_i ∈ D_many do
9:          put rules from each policy group related to D_i into
            R_list based on the priority
10:      end for
11:   end if
12:   put rules of policy group for each D ∈ D_one into R_list
13: end if
```

Algorithm 2 PGRules

```
1: for each policy group P do
2:    P.state ← unfinish
3: end for
4: for each P with P.state = unfinish do
5:    ruleGeneration(P)
6: end for
```

group. The conflicts can be detected at compilation time, and indeed our compiler gives the error reports about the conflicts for the common addresses. This allows the user to resolve the conflicts by explicitly applying the policy groups with correct order.

3. RULE TRANSLATION ALGORITHM

The policy defined with FLIP must be translated into lower level rules in order to be enforced by a firewall. Translation uses an intermediate language of common-format packet-filtering rules: high-level policies are translated into common format rules, and then the common format rules are translated to specific device configurations. It is crucial for that these steps not introduce ambiguous or false low-level rules.

In this section, we introduce our rule translation algorithm and show its soundness and completeness: that is, the output IL program exactly captures the semantics of the original FLIP program.

3.1 Semantics of the Intermediate Language

The common-format packet filtering rules (also called IL, or intermediate language) have the format:

```
    <IL> ::= <rule> { ";" <rule> }
  <rule> ::= <header> "->" <action>

<header> ::= <protocol>
             <src_ip> ":" <src_port>
             <dst_ip> ":" <dst_port>
<action> ::= "accept" | "deny"
```

The source and destination IP addresses in IL rules may include wildcards and subnet masks.

A *packet* consists of a header and data. An IL program determines whether a given packet is accepted or denied by going through the rules in order. The first rule header that matches the packet header determines the outcome (accept or deny).

Algorithm 3 ruleGeneration. Input: policy group P

```
1: if P.state = unfinish then
2:    put enforced rules at the top of high-level policy defi-
      nition
3:    ER ← translate enforce rules of each service
4:    NR ← translate normal rules of each service
5:    Q ← P.parent
6:    if Q = NULL then
7:       P.Rules.add(ER)
8:       P.Rules.add(NR)
9:    else
10:      parentRules ← ruleGeneration(Q)
11:      P.Rules.add(parentRules.ER)
12:      P.Rules.add(ER)
13:      P.Rules.add(NR)
14:      P.Rules.add(parentRules.NR)
15:   end if
16:   P.state ← finish
17: end if
18: return P.Rules
```

The semantics of an IL program is given as a pair (A, D) where A is the set of packets accepted by the program and D is the set of packets denied by the program. Clearly A and D must be disjoint. Note that if no rule header matches a packet, then the packet is neither accepted or denied. Two IL programs are equivalent if the accept and deny exactly the same packets.

3.2 Semantics of FLIP

In the next subsection we present an algorithm that translates FLIP into IL. Let A_{FLIP} be the accept set of the original FLIP program and let A_{IL} be the accept set of the IL program which is the result of translation. The translation is *sound* if $A_{IL} \subseteq A_{FLIP}$ and *complete* if $A_{FLIP} \subseteq A_{IL}$, and similarly for the denied sets.

To give a compositional semantics for FLIP, we consider a slightly richer model than that used for IL. The meaning of a rule (or a group of rules such as a policy group applied to a domain) is a four-tuple (A, D, EA, ED), where each element of the tuple is a set of packets.

- A represents the packets accepted by the rule and D represents the packets that are denied by the rule. Packets that are neither explicitly accepted nor explicitly denied will not appear in either A or D.
- EA (resp. ED) stand for the packet-headers that correspond to the *enforced* allow (resp. deny rules).

The following invariants hold: $EA \subseteq A$, $ED \subseteq D$, $A \cap D = \emptyset$, and $EA \cap ED = \emptyset$. The four-tuples carry redundant information used for combining FLIP programs together. When finally executed, the sets A and D determine the packets accepted and denied.

Definition 1. *A FLIP program with meaning* (A, D, EA, ED) *is* equivalent *to an IL program with meaning* (A', D') *if* $A' = A$ *and* $D' = D$.

We understand the inheritance construct of FLIP denotationally as an operation on four-tuples. Let $P_1 = (A_1, D_1, EA_1, ED_1)$ and $P_2 = (A_2, D_2, EA_2, ED_2)$. Then $P' = (A',$

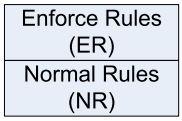

Figure 4: Format of Low level rules generated by algorithm

$D', EA', ED')$ represents the result of P_1 `extends` P_2, where

$$ED' = ED_2 \cup [ED_1 \backslash EA_2]$$
$$EA' = EA_2 \cup [EA_1 \backslash ED_2]$$
$$D' = [D_1 \backslash EA_2] \cup [D_2 \backslash A_1] \cup [ED' \backslash EA']$$
$$A' = [A_1 \backslash ED_2] \cup [A_2 \backslash D_1] \cup [EA' \backslash ED']$$

The rest of the FLIP constructs are semantically straightforward. Thus, we can associate a semantics as a 4-tuple (A, D, EA, ED) with every FLIP program.

3.3 Rule Translation Algorithm

FLIP performs conflict discovery before translation. The high-level policies are translated into low-level rules only when all conflicts have been resolved [1].

The top-level rule generation algorithm is shown in Algorithm 1. After the rule translation for each policy group, FLIP analyzes the assignment between policy groups and domains. For those domains or subdomains which are applied with more than one policy groups, FLIP generates rules based on their priorities.

In order to translate high-level policy into low-level common format filtering rules, FLIP generates rules for each policy group, as described in Algorithm 2.

Each policy group is then translated into low-level rules, as described in Algorithm 3.

Because of the constraints imposed in section 2.2, there can be no overlapping between high level rules in the same policy group. So, in each policy group, we can process each collection of high-level rules separately and modularly. The disjointness also permits us to construct convenient orderings of the low-level rules. In our representation, we always put the low-level rules corresponding to the enforced rules at the top (of each policy group) — see the diagram in figure 4.

The algorithm works modularly with respect to the constructs of FLIP. Figure 5 shows an example of how the low level rules from parent policy group are inherited by child policy group. On the sides of the picture, we have pictorial representations of the low-level rules that implement the enforce rules (ER) and normal rules (NR) of the parent and the child. The picture in the middle illustrates the result of inheritance. The resulting rule order is consistent with the hierarchical relation between policy groups — the enforce rules (ER) from parent group are put on top of the rule list, then the local enforce rules. After that, the normal rules (NR) from local policy group and normal rules form parent group are put into rule list sequentially.

3.4 Results

We prove that our translation of the inheritance combinator of FLIP preserves equivalence.

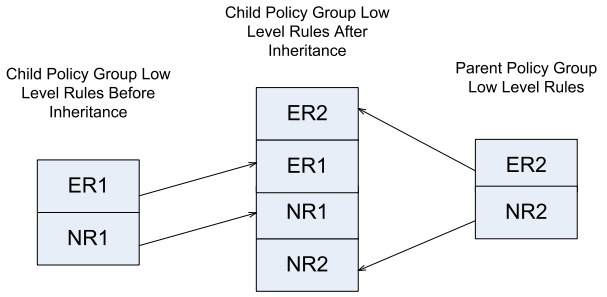

Figure 5: Illustration of translation algorithm for inheritance

LEMMA 1. *Let the translation of policy group P_1 (resp. P_2) be equivalent to P_1 (resp. P_2). Then the policy group P_1 be extended from policy group $P_2$is equivalent to its translation given by the algorithm.*

PROOF. The proof is based on boolean algebra manipulations. We sketch the operational intuitions underlying the equivalence by tracing the accept/deny decisions.

When policy group P_1 extended from P_2 algorithm 3 adds enforced rules of P_2 to the top of low level rules list. Then it adds the enforce rules of P_1 into the rules list. The rule order guarantees that the enforce rules from parent group will be matched first. So the `enforce allow rules` of P_1 after translation will be the union of `enforce allows rules` with the `enforce allow rules` of $P_1$from P_2 that are not already denied by the `enforce deny rules` of P_2, i.e.

$$T(EA_1') = EA_2 \cup [EA_1 \backslash ED_2].$$

Similarly, we can get the required equivalence for $T(ED_1')$.

Based on the rule order, the `normal allow rules` defined in P_1 will be hit if the combined enforce rule can not match the packet. And the `normal allow rules` defined in P_2 can only be hit if both the combined enforce rules and the normal rules defined in P_1 can not match the packet. So the total allow rules will be the union of the `enforce allow rules`$(EA_1' \backslash ED_1')$, `normal allow rules` in P_1 (A_1) that are not denied in P_2 (ED_2) and the `normal allow rules` in P_2 (A_2) that are not already denied in P_1 (D_1). So we get

$$T(A_1') = [A_1 \backslash ED_2] \cup [A_2 \backslash D_1] \cup [EA_1' \backslash ED_1'].$$

Similarly, we can get the required equivalence for $T(D_1')$. □

This is the key case of the proof since the priorities between multiple policy groups on a domain is also resolved in a fahsion analogous to inheritance of policies. So, we have the desired theorem:

THEOREM 1. *Any FLIP program is equivalent to the IL rules generated for that program by the translation algorithm.*

4. A CASE STUDY

Through the following example, we show how FLIP can be used to define security policy in a large-scale environment with different security requirements. We use the School of Computer Science, Telecommunications and Information Systems (CTI) of DePaul University as our case study. The

CTI network is composed by a set of departments, administration office, research labs, servers (mail, FTP, Web, etc.), and the desktop of faculties and staffs. Based on the information sensitivity, research and administration requirements, each office and lab may have different security policies. We use a fraction of these policies to demonstrate the usability and expressiveness of FLIP, these policies by no means cover all the security requirements in real life deployments. The security requirements are outlined as follows:

- All labs must not allow SSH [19] access from machine other than the network administrators' machine.
- It is recommended (but not enforced) to prevent students in labs to access `www.yahoo.com`.
- If it does not violate the CTI or research lab general policy, `mnlab` would like to allow incoming multicast traffic and to allow student access any web page.
- The `admin` group may disallow student access from remote campuses through remote desktop. However, faculty are allowed remote desktop access from remote campuses.
- NFS should not be allowed through a firewall in `mnlab`.

The domains and services used in this example can be defined as follows:

```
 domain CTI = [140.192.*],
        remote_campus = [140.192.8.*],
        Labs = [140.192.35.128 -140.192.37.255],
        mnlab = [140.192.37.128-140.192.37.143],
        faculty = [140.192.34.*],
        CTIadmin = [140.192.34.224-140.192.35.15],
        Blacklist = [207.115.*];
        multicast = [224.0.0.0 -239.255.255.255],
service http = tcp.[port =80],
        telnet = tcp.[port =23],
        remote_desktop = tcp.[port=3389],
        ssh = tcp.[port =22]
        NFS = tcp.[port=2049]
```

First we define the policy group `CTI_Policy` which is the general policy for CTI domain:

```
policy_group CTI_Policy {
  incoming:
    tcp.[port = *] { enforce deny BlackList }
}
```

Then we extend `CTI_Policy` to define the general policy for research lab.

```
policy_group Lab_Policy extends CTI_Policy {
  incoming:
    enforce ssh { deny * except CTIadmin }
    enforce udp.[port=*] { deny multicast }
  outgoing:
    http { deny to yahoo.com }
}
```

Now we can define the policy for multimedia networking lab (`mnlab`). Please be aware that the policy of http traffic to yahoo defined in `Lab_Policy` is overwritten by the policy defined in `mnlab_Policy`. Although, the policy in `mnlab_Policy` try to allow multicast traffic, it conflict with the enforce rule inherited from `Lab_Policy`. So multicast traffic can not reach `mnlab`. Through this example, we can see how the enforce rule is kept during policy group inheritance.

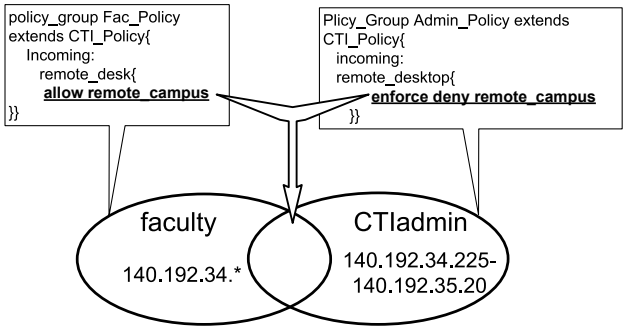

Figure 6: Conflicts between Fac_Policy and Admin_Policy

```
policy_group mnlab_Policy extends Lab_Policy {
  incoming:
    udp.[port= any] { allow multicast }
    NFS { deny * }
  outgoing:
    http { allow * }
}
```

The policy for CTI admin and faculty ca be defined by extending the general global policy group (CTI_Policy)as follow:

```
policy_group Admin_Policy extends CTI_Policy{
  incoming:
    remote_desktop { enforce deny remote_campus }
}


policy_group Fac_Policy extends CTI_Policy{
  incoming:
    remote_desktop { allow remote_campus }
}
```

After define all the policy and domain, now we can apply the policy groups on domains.

```
apply CTI_policy on CTI;
apply Lab_policy on Labs;
apply mnlab_Policy on mnlab;
apply Admin_Poliy, Fac_Policy on faculty* CTIadmin;
apply Fac_Policy on faculty;
apply Admin_Policy on CTIadmin;
```

Because there is an intersection between `faculty` and `CTIadmin` domain, the common part of these two domains will be applied with two policy groups. There is a conflict between policies for remote desktop service, as shown in Figure 6. For the common part we explicitly apply both `faculty` and `CTIadmin`, and give `CTIadmin` high priority; thus the intersection between `faculty` and `CTIadmin` will not support remote desktop access. Due to space limitation, we only show the low level rules in IL format generated by FLIP for `mnlab` which is applied with `mnlab_Policy`:

```
1 tcp         207.115.*:any 140.192.37.128/28:any ->deny
2 tcp 140.192.34.224/27:any 140.192.37.128/28:22  ->allow
3 tcp   140.192.35.0/28:any 140.192.37.128/28:22  ->allow
4 tcp               any:any 140.192.37.128/28:22  ->deny
5 udp       224.0.0.0/3:any 140.192.37.128/28:any ->deny
6 udp       224.0.0.0/3:any 140.192.37.128/28:any ->allow
7 tcp               any:any 140.192.37.128/28:2049->deny
8 tcp 140.192.37.128/28:any                any:80  ->allow
9 tcp 140.192.37.128/28:any    69.147.114.210:80  ->allow
```

Rule 1 is the enforce rule for `BlackList` defined in `CTI_Policy`. The enforce rules for ssh defined in `Lab_policy` is translated into rule 2,3 and 4. Rule 5 is the enforce rule for multicast

Experience	Time Man-written(m)	Time With FLIP(m)	Man-written Conflicts	FLIP Conflicts
Expert	30	17	7	0
Intermediate	51	24	13	0
Beginner	75	32	17	0

Figure 7: Average finishing time and number of anomalies of the policy definition experiment

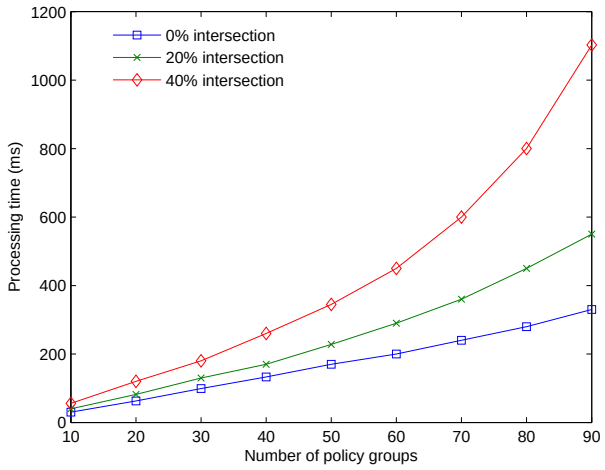

Figure 8: Processing time for conflict discovery.

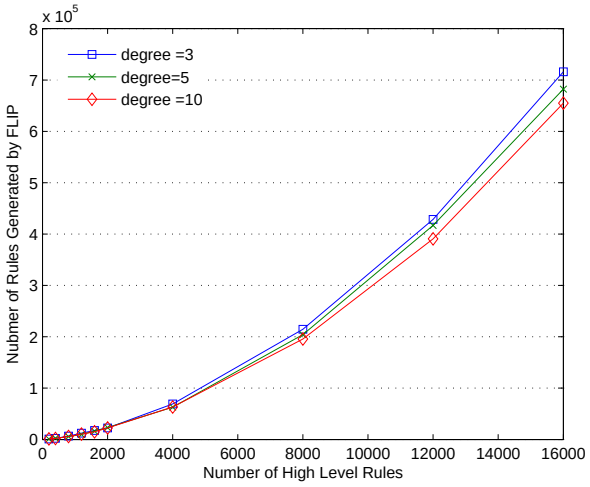

Figure 9: The generated rules number changes as degree changes

traffic defined in `Lab_policy`. The rule for allowing multicast traffic defined in `mnlab_Policy` is translated into rule 6. Rule 7 is for NSF traffic. The rule of allowing Web traffic from `mnlab_Policy` is shown in 8, which overwrites the rule of deny Web traffic to yahoo.com from `Lab_policy` (rule 9).

5. IMPLEMENTATION AND EVALUATION

We implemented the techniques and algorithms described in this paper in a software module called FLIP tool. The implemented tool, FLIP, performs conflict detection and low-level rule generation. The implementation is built in Java. In this section, we present our evaluation study of the usability and the performance of the FLIP.

5.1 FLIP Usability

To assess the practical value of our techniques, we used the FLIP to design some real firewall rules based on different security requirements. FLIP has shown to be effective by reducing the policy development time without introducing conflicts. We then attempted to quantitatively evaluate the practical usability of FLIP by conducting a experiments that consider the level of network administrator expertise, the developing time and number of anomalies in the final result. In this experiment, we created a firewall policy exercise and asked 12 network administrators with varying level of expertise in the field to complete the exercise. The network administrators are separated into two groups, each group has equal numbers of administrators in same level. The exercise is to write firewall rules based on a given security policy requirements. The total number of rules was around 60 in a network having only three firewalls. The results of this experiment are shown in Figure 7. We can see that with the help of FLIP, the beginner can complete the experiment in almost half the time needed by the expert with no conflicts.

5.2 Scalability and Performance

In second part of our evaluation study, we conducted a number of experiments to measure the performance and the scalability of FLIP. We produced three sets of policy groups and domains definition. In first set, there is no intersection between target domains on which the policy groups are applied. In second set, 20% of target domains have overlapping, and in third set, 40% of target domains have overlapping. Our simulation target a B class IP network, we assume each policy group explicitly defines 20 high level rules, and each policy group is applied to a different domain. We use FLIP to analyze these policy groups to find conflicts with various number of policy groups. The processing time needed to finish the complete analysis is shown in Figure 8. The result shows clearly that as the percentage of intersecting domains increase, the time needed to detect the conflicts increase. This is because each domain intersection is applied with more than one policy group, the rules from different policy groups need to be analyzed together to detect the conflicts. Our experiments were performed on a Pentium PIII 800 MHz processor with 256M Byte of RAM.

5.3 The Number of Rules Generated by FLIP

The number of rules generated by FLIP is crucial to the firewall performance, since running time to find a match increases linearly in the number of rules [9]. Our goal is to avoid generating unnecessary or redundant rules. FLIP translate the rules into general firewall configuration format, like the format in IPTables [14]. That means the source and destination can not be defined as IP Range. So the

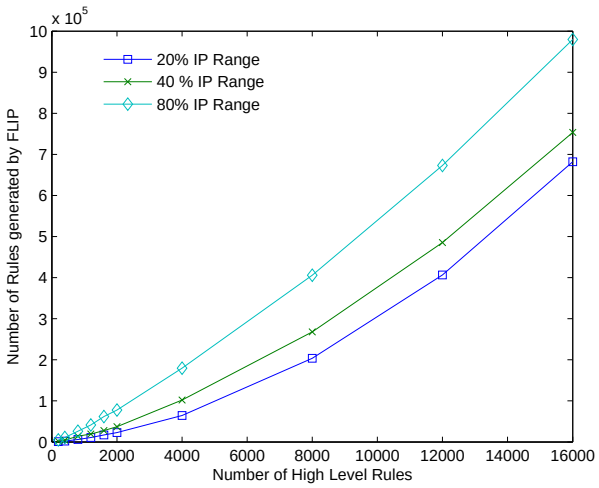

Figure 10: The generated rules number changes as percent of domains defined by IP Range changes

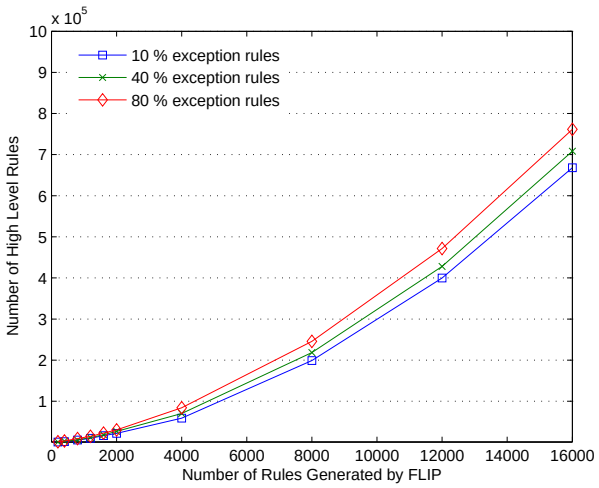

Figure 11: The generated rules number changes as percent of exception rules change

domain defined by IP Range have to be break in to subdomain which can be defined with network address and subnet masks. Due to the policy group inheritance and domain (IP range) separation, the number of low level rules generated by FLIP is larger than the explicitly defined rules in high level policy groups. In the example in Section 4, we define 8 domains, 6 network services and 6 policy groups, after translation, FLIP generates 48 low level rules.

In this section, we evaluate the rules generated by FLIP under different scenario and try to identify the key factors which influence the rule numbers. There are three factors which contribute to the number of rules generated by FLIP engine: policy group inheritance, domain definition, and number of exception rules. We evaluate their influence separately. For this study, we continue use the previous setting (each group has 20 high level rules).

The Impact of Policy Group Inheritance.

We control the inheritance relationships between groups by define the maximal child group each parent can possible has (degree). The smaller the degree, the larger the depth of the tree. This means child groups inherit more rules from parent groups. In this experiment, we assume there are 20% domains defined by IP range, and 20% of high level rules have exception. We generate a set of policy groups and a set of domains. We fix the relation between the policy group and domain, conduct the experiment with different degree values. The result of number of low level rules is shown in Figure 9. For this result, we can see that the change of degree does not influence the final rules number much.

The Impact of Domain Definition With IP Range.

In this study, we fix the degree =5, exception rules ratio equal 20%. We control how many percentage of domain are defined with IP Range, and study its impact on the rule generation. The result is shown in Figure 10. From this figure, we can see that as the percentage of domains defined with IP Range increase, the rules number increase evidently. This is because the more the domains defined with Ip Range, the high the possibility that these domains have overlapping. More overlapping domains may introduce more conflicts. Since our approach breaks large overlapping domains into small disjoint subdomains to resolve the conflicts for each subdomains, as the overlapping domains increase, the number of rules generated by FLIP also increases.

The Impact of Exception Rules.

Exception rules need be translated specially in our approach, it need more than one rules to represent the user's objectives. We fix the degree equal to 5, 20% of domains defined with IP range, and study the number of rules changes as the percentage of exception rules change. The result is shown in Figure 11. From this figure, we can see that with exception rules increase, the total low level rules does not increase much.

6. RELATED WORK

In this section we will review some representatives of high level firewall languages and works about firewall policy deployment. We evaluate these languages based on following aspects: conflict free, rule-order dependency, easy to use, expressiveness, modularity and reusability.

The high-level firewall language (HLFL) [12] project is an approach to translate the high level firewall rules into useful rules for IPChains, Netfilter, Cisco and many others. HLFL adopts a simple grammar to define firewall rules:

```
proto src operator dst [interface][keywords]
```

The user can use this language to define the rules for an entire network. The major contribution of this approach is the automated translation, but it lacks important features such as detecting and preventing the conflicts in the firewall rules. Also the rule order is important in this language: in order to update the firewall policy, the user must find the appropriate position to insert the rules. This is very difficult when there are large number of rules in a firewall.

Firmato [4] is a complete firewall management toolkit, based on entity-relation model. With Firmato, the user can define host group, service group and role group and the relation between these groups. The Firmato entity-relation model can be viewed as the application of a role-based model to the firewall policy area. Firmato support the separation

of policy definition from network topology to increase the language modularity and reusability. One of the major limitation of Firmato is that the user can only define `allow` rules. `deny` rules are supported only by the default rule: all traffic not explicit allowed will be denied. Although this make Firmato conflict-free and rule order independent, it greatly limits its expressiveness and usability. Filtering postures adopts the same idea of only supporting `allow` rules, so it shares the same limitation.

The Inspect language [13] is an object-orient, high level script language to define the packet handling rules. Inspect supports features (such as functions, macros, and dynamic data structures) which are usually supported only by full programming languages. Inspect lacks conflict detection and rules are order dependent. The language also requires detailed knowledge of TCP/IP.

Netspoc [16] is a description language for security policy and topology. In netspoc, policy is a set of related rules, network objects and service definitions are used in rules to describe network traffic. Rules in a single policy refer to the same network objects. Netspoc resolves the conflicts between rules by giving the deny rules higher priority, automatically overwriting allow rules. Also this is a safe approach, it make the debugging of firewall policy more difficult. If desired traffic is blocked by the firewall, it's hard to find the cause of the problem.

Security policy specification language (SPSL) [10] is an attempt by IETF to create a policy specification language for use with IPSEC. The syntax was derived from the routing policy specification language. SPSL use objects, each object has attributes to store the data. SPSL can define node based or domain based policy. The order of rules in SPSL is important, it will enforce the first matching policy. It also lacks the conflict detection and and other features that is important to large-scale policy management like inheritance and restriction.

Although all these approaches propose some high level language to define firewall policy, each has its own advantages and limitations. FLIP offers novel features that are important for firewall policy management but missing in most of these solutions.

Safety in firewall policy deployment has been studied in [21]. A deployment is safe if it does not cause the firewall to drop legal traffic or permit illegal traffic during deployment. This work introduces formal definition and theoretical analysis of safe deployment problem and proposes algorithms to discover the smallest possible number of editing commands to safely deploy firewall policy. Our work can benefit from the algorithms proposed in this work to find the safe and efficient ways to update firewall rules.

7. CONCLUSION AND FUTURE WORK

Managing security policy in large-scale enterprise network requires a fixable policy definition, conflicts resolution and rule distribution. Simply having firewalls on the network boundaries or between sub-domains may not necessarily make a network any more secure. Misconfiguration of security devices due to rule conflicts or policy inconsistency may increase network vulnerabilities. In this paper, we proposed a new high-level firewall policy language (FLIP) to manage distributed firewalls globally and transparently. FLIP is a service-oriented language that focuses on the service security requirements, unlike rule-oriented languages that impose traffic detail in the configuration. FLIP allows network administrators to define their high-level security goals that are then translated to rules and distributed in the firewall devices automatically. The generated rules are order-independent and conflict-free for intra- or inter-firewall configuration. The FLIP modularity and reusability exhibited by the policy group definition greatly reduce the time and effort to define firewall policies and generate rules in a large-scale enterprise networks. The hierarchical structure of policy groups improves the understanding of the interaction between the global and local policies. We also show the soundness and completeness of our rule translation algorithm. So, FLIP can guarantee the consistency between the security policy definition groups and the enforcement points

In our evaluation study, we have shown that using FLIP enables the network administrators with little experience to design the firewall policies faster and more accurately than experts in the field. In regards to performance, although the policy group analysis are parabolically dependant on the number of policy groups and the number of rules in each group, our experiments show that the average processing time for conflicts discovery is very reasonable for practical applications. Using our Java implementation of the FLIP engine, our results indicate that, in case of 80% of domains are intersected with each other, it takes 56 to 1103 ms of processing time to analyze 10 to 90 (200-1800 rules) policies groups. We also show that the number of rules generated rules by FLIP is very comparable to the rules defined manually in IPTables format.

Our future research plan includes extending FLIP to support other security devices: IPSec, IPS/IDS, and extending FLIP to accommodate business-level objectives/abstraction. We also want to improve the rules translation and distribution algorithms, optimize the rules order, combine and aggregate rules to improve firewall performance.

8. ACKNOWLEDGEMENTS

This research was supported in part by National Science Foundation under Grant Cybertrust 0430175 and Career 0347542. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

9. REFERENCES

[1] Ehab Al-Shaer and Hazem Hamed, Discovery of Policy Anomalies in Distributed Firewalls,In Proceedings of IEEE INFOCOM'04, March 2004

[2] Ehab Al-Shaer and Hazem Hamed, Taxonomy of Conflicts in Network Security Policies, IEEE Communications Magazine, Vol. 44, No. 3, March 2006

[3] BitTorrent `http://www.bittorrent.com/`

[4] Y. Bartal., A. Mayer, K. Nissim and A. Wool. Firmato: A Novel Firewall Management Toolkit. *Proceedings of 1999 IEEE Symposium on Security and Privacy*, May 1999.

[5] Ian Foster The Grid: Blueprint for a New Computing Infrastructure *Morgan Kaufmann*, 2004.

[6] M. Greenwald, S. Singhal, J. Stone, and D. Cheriton. Designing an Academic Firewall. Policy, Practice and Experience with SURF. *Proc. of Network and*

Distributed System Security Symposium (NDSS), pages 79"C91, February 1996.

[7] Greg Graham, Richard Cavanaugh, Peter Couvares, Alan De Smet, and miron Livny Distributed Data Analysis: Federated Computing for High-Energy Physics *The Grid: Blueprint for a New Computing Infrastructure*, 2004.

[8] Hazem Hamed, Ehab Al-Shaer and Will Marrero, Modeling and Verification of IPSec and VPN Security Policies. *In Proceedings of IEEE ICNP'2005,* November 2005.

[9] Hazem Hamed and Ehab Al-Shaer, Dynamic Rule-ordering Optimization for High-speed Firewall Filtering, *ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS'06),* March 2006.

[10] D. Harkins, D. Carrel, The Internet Key Exchange (IKE) *RFC 2409*,1998

[11] J. D. Howard. An Analysis Of Security On The Internet 1989 - 1995. *PhD thesis, Carnegie Mellon University,*, April 1997

[12] High Level Firewall Language `http://www.hlfl.org/`

[13] The INSPECT Language guide `http://www.security-gurus.de/papers`

[14] IPtables `http://www.netfilter.org/`

[15] S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith. Implementing Distributed Firewall. *Proceedings of Computer and Communications Security (CCS), pages 190"C199,* November 2000.

[16] NetSPoC: a Network Security Policy Compiler *http://netspoc.berlios.de*

[17] D. Nessett and P. Humenn. The Multilayer Firewall. *Proc. of Network and Distributed System Security Symposium (NDSS), pages 13"C27,* March 1998.

[18] Squid Web Proxy Cache `http://www.squid-cache.org/`

[19] Tatu Ylönen SSH — secure login connections of the internet. *In Proceedings of the Sixth USENIX Security Symposium, San Jose, California, USA,* July 1996.

[20] World of Warcraft `http://www.worldofwarcraft.com/`

[21] Charles C. Zhang, Marianne Winslett and Carl A. Gunter. On the Safety and Efficiency of Firewall Policy Deployment *Proc. of IEEE Symposium on Security and Privacy,* May 2007.