Tri-Modularization of Firewall Policies

Haining Chen, Omar Chowdhury, Ninghui Li, Warut Khern-am-nuai
Purdue University
{chen623, ochowdhu, ninghui, wkhernam}@purdue.edu

Suresh Chari, Ian Molloy, Youngja Park
IBM T. J. Watson Research Center
{schari, molloyim, young_park}@us.ibm.com

ABSTRACT

Firewall policies are notorious for having misconfiguration errors which can defeat its intended purpose of protecting hosts in the network from malicious users. We believe this is because today's firewall policies are mostly monolithic. Inspired by ideas from *modular programming* and *code refactoring*, in this work we introduce three kinds of modules: *primary*, *auxiliary*, and *template*, which facilitate the refactoring of a firewall policy into smaller, reusable, comprehensible, and more manageable components. We present algorithms for generating each of the three modules for a given legacy firewall policy. We also develop ModFP, an automated tool for converting legacy firewall policies represented in access control list to their modularized format. With the help of ModFP, when examining several real-world policies with sizes ranging from dozens to hundreds of rules, we were able to identify subtle errors.

CCS Concepts

•**Security and privacy** → **Access control; Firewalls;**

Keywords

Firewall policies; Modularization; Firewall tool

1 Introduction

A *firewall* is among the first lines of defenses for protecting a network (or, a host) from malicious users. A firewall intercepts network packets, and based on a specific *firewall policy*, decides whether to *allow* or *deny* certain packets to pass through it. As firewalls are developed by many vendors (*e.g.*, Cisco, Check Point), the syntaxes and semantics of firewall policy languages vary. However, at its core, most of the packet filtering rules expressed in these specification languages can be translated into an *access control list* (*ACL*) representation. An ACL firewall policy is specified as an *ordered list of rules*. Each rule has the form "target→action", in which target specifies a set of packets to which this rule is applicable, and action states what should be done with the packet. In an ACL, multiple rules can be applicable to a single packet and the decision of the first rule that is applicable to the packet is imposed on the packet. This is known as the "*first match semantics*".

SACMAT'16, June 05-08, 2016, Shanghai, China

DOI: http://dx.doi.org/10.1145/2914642.2914646

Due to the dynamic nature of a network and its surrounding environment (*e.g.*, addition of new services, discovery of new attacks, a host becoming compromised), the firewall policies must evolve over time, in order to maintain a robust defense against malicious users while allowing legitimate traffic. Hence, it is necessary for firewall policies to be *intellectually manageable*, a term used in the context of programming by Edsger W. Dijkstra in his 1972 ACM Turing Lecture [16]. That is, administrators should be able to understand existing policies, possibly designed by other administrators. They should be able to modify a policy to achieve some intended objectives, mentally assess what the policy does, and "debug" the policy when problems arise.

Regrettably, many firewall policies are not intellectually manageable. For instance, it has been observed that most firewalls on the Internet are poorly designed and have many configuration errors in their rules [38, 39]. As firewalls can only be as effective as their configuration, misconfigurations of firewalls undermine their intended purpose of protecting the networks in question, causing firewalls to offer only a false sense of security.

One characteristic of firewall ACL languages is that two ACL rules may be in *conflict* with each other if they have different decisions (*e.g.*, one allows the packet to pass, and the other drops the packet) but their applicable sets of packets overlap. This means that the semantic of one rule may be changed by other rules that are in conflict with it. Because of this, writing firewall policies has been compared with writing programs with extensive use of goto statements. (See, *e.g.*, [18].) However, as policy rules often have exceptions, more often than not using conflicts is the most succinct way of expressing actual policies.

We argue that (1) the potential for conflicts is only one of three factors causing the difficulty. The other two are (2) policies expressed in ACL-based languages are monolithic; and (3) complex policies require a large number of rules. A monolithic policy can only be understood as a whole. This becomes infeasible as the policy gets large, since most people are unable to put a large amount of information in the working memory.

Since we cannot change the fact that many policies are inherently complex and that conflicts are useful, the only factor we can affect is the monolithic nature of current firewall policies. A notion of modular firewall policy was introduced in [5], where a firewall policy is considered modular if the policy is partitioned into multiple policy components $M_1, \ldots, M_r$ such that each packet is accepted by at most one component. This approach is still inherently monolithic, since one still potentially needs to examine all components when trying to understand what is the decision for one packet.

The goal of our work is to elevate firewall policies from monolithic to modular. The contributions of this paper are as follows.

First, we recognize five requirements for a successful modular-

ization approach (*i.e.*, logical partitioning, isolation among modules, flexible partitioning structure, human-computable policy slicing, and readily deployability), and analyze existing approaches using these requirements to identify their shortcomings.

Second, we introduce our approach of modularizing firewall policies. This includes the concept of a *primary attribute*, which is either the source IP, the destination IP, or the service. The optimal choice of the primary attribute is policy dependent, although for the several dozens of policies we have observed, most of them benefit more from choosing the destination IP as the primary attribute. A policy is partitioned into three kinds of modules: *primary*, *auxiliary*, and *template*. Beyond making policies more modular and easier to understand, our approach also supports policy refactoring, either by distilling templates from recurring patterns, or by breaking up a large module into multiple smaller ones, each covering a subset of the IP range.

Third, to support legacy firewall policies, we have defined a 5-step process and introduced algorithms for converting them into their modularized form. We have also implemented an automated tool called ModFP for this purpose. By utilizing ModFP, we have converted several real-world firewall policies into their modularized form, and found that the process consistently improved the understanding of a policy, and the benefit is much more significant when the policy is large and/or when it has substantial usage of both permit and deny rules. For majority of the real-world firewall policies, their modularized version—translated with ModFP—enjoys a significant number of rule reduction (*i.e.*, 25.3%-68.7%) compared to the original ACL policy. Additionally, the translation from ACL to the modularized version takes a matter of seconds (*i.e.*, 0.26-19.35 seconds). For every large policy we have examined, we have found clear errors (such as redundant rules) as well as irregularities that we conjecture to be errors. For one such policy deployed in a corporate setting, we were able to contact the administrators and confirm that most of our findings are indeed policy errors.

2 Background and Related Work

In this section, we review the ACL representation of firewall policies, and then discuss related work.

2.1 Background on Firewall

A *firewall* typically operates at the gateway of a network to protect the network. The firewall determines whether to allow (resp., deny) certain packets based on some configurable *firewall policy*. Such a policy considers the following fields of a packet while matching it against the rules' target: source IP address (denoted by sIP), source port (denoted by sPort), destination IP address (denoted by dIP), destination port (denoted by dPort), and protocol/service (denoted by protocol). A firewall policy consists of rules, where each rule has the form "target $\rightarrow$ action", where the four actions in Table 1 are possible.

action	**Effect of** action
allow	allow the matched packet to pass
deny	drop the matched packet
chain$Y$	for matched packet, go use chain "Y"
return	resume calling chain

Table 1: Four actions in firewall rules.

Most firewall languages use a *simple list model* where each rule's action (or, decision) is either allow or deny. Linux netfilter uses a *complex chain model*, where a policy consists of multiple chains and all four actions can be used. The chain$Y$ action directs the evaluation to another chain Y, which should include rules using the return action. Similar to a subroutine, the chain Y can be invoked from multiple places.

In a policy, more than one rules may match a packet, and their decisions may conflict. Firewall rule lists use the "*first match semantics*". Hence, the order in which the rules are organized is important in making the decision about whether a packet should be accepted or denied. Most policies have a "catch-all" rule as the last rule, which will match all packets and provide a default decision for any packet that is not matched by any earlier rules. In most cases, this "catch-all" rule has "deny" as the decision.

2.2 Related Work

Wool [38, 39] studied errors in real-world Firewall policies. They define certain characteristics as configuration errors and found that the number of errors in a policy is correlated with the number of rules in a policy. 36 such characteristics are used in [39], including "to any address allow any service" rules, outbound "any" service rules, inbound or outbound instance messaging rules, and so on. While our experience also shows that firewall policies contain many errors, we point out that most of these "configuration errors" as defined in [39] are really irregularities. They may indicate an error, but could also be intended by administrators for some specific reasons. This indicates a fundamental challenge in dealing with firewall configuration errors. Without knowing the original intention of the administrators, it is often impossible to tell whether something in a policy is a feature or a bug.

Analysis and Testing Tools. One line of research aims to identify anomalies in firewall policies [8–11, 13, 24, 40], either in a single policy, or in multiple policies placed on a network. Algorithms and tools were developed to detect anomalies and recommend how they can be fixed. Such techniques resemble static analysis tools for detecting bugs in software programs. They can detect errors manifested as anomalies, but not logical bugs where the policy does not implement what the administrators intend to enforce.

Another approach to deal with firewall policy errors is to develop debugging tools. Some tools generate and send testing packets and check whether they can go through firewalls. Other tools model firewalls using some formal modeling tools (often decision diagrams) and allow administrators to query the policy model [19–21,26,27,29,34,37]. For example, one can issue queries such as "*Which hosts can access the web server at 10.10.2.3*?". With these techniques, administrators need to come up with appropriate queries that provide sufficient coverage and expected answers for these queries.

Several other interesting approaches have been proposed. One is change-impact analysis [23], which takes as input a firewall policy and a proposed change, and outputs the impact of applying the change, such as what packets will have their decisions reversed. Another is classifying the hosts of a network into equivalence classes [28]. Two hosts are equivalent if after changing a packet's source (similarly for destination) IP address from one to the other, the decision remains the same. Techniques to automatically correct errors in firewall policies, when a number of test cases (i.e., packets and the corresponding correct decisions) are given as input, were developed in [15].

Like the case of software development, static analysis and debugging tools are valuable; however, they cannot fully mitigate the problem caused by a primitive programming language lacking support for abstractions and modularization. The work we present in this paper aims at introducing such support.

Automatic policy generation. Instead of specifying firewall policies, in [17], a method is proposed to discover firewall policy rules by first mining the network traffic log using association rule min-

ing, then aggregating the resulting rules, and finally detecting and removing anomalies in the policy using techniques in [11]. In [31], an architecture is proposed for automatically generating conflict-free firewall rules with alert information from network and system logs in multiple-firewall scenarios.

Policy representation. A method to convert an ACL rule list to a textual representation was proposed in [12]. The method first aggregates rules that are similar (*e.g.*, they differ only in one field) together, and then translate them into text. For example, it may produce a rule that reads: "accept all TCP traffic from address $140.192.37.*$ and {to port 80 or to port 21}". A similar approach was proposed by Tongaonkar *et al.* [33], in which given a firewall policy, they first flatten the policy rules into non-overlapping ones using Directed Acyclic Graph (DAG), so that the order of the rules does not affect the policy semantic. Then they merge similar rules to make compound rules such that a complexity metric is minimized. This kind of methods are beneficial to represent legacy policies in a more compact and understandable flavor.

Bartal *et al.* [14] develop a workflow for specifying firewall policies which proceeds in three stages: (i) abstract policy specification, (ii) policy instantiation, and (iii) automatic rule generation. They also develop a rule illustrator that visualizes which traffic between any two hosts are allowed. Their workflow is suitable for a new organization which is setting up their network instead of improving the manageability of legacy firewall policy.

Most commercial firewall policy languages or tools provide similar textual or graphical-based interface, as well as the ability of defining objects that can group multiple hosts into a group, and use these groups in a policy. This provides the functionality of macros at the level of individual fields in firewall policies.

Several efforts exist to specify firewall policies over packet flows between two ranges of IP addresses (which can be implemented by multiple firewalls that are in between the source and destination networks). One example is a firewall specification language for Linux netfilter introduced in [6].

In summary, the languages discussed above provide three kinds of abstractions: (1) named objects that group related IP addresses or port numbers together, similar to macros; (2) defining policies in a global network view instead of the view of a single firewall; and (3) syntactic sugars, *e.g.*, making the rules more like a natural language description. These are orthogonal to the kinds of abstractions we introduce for modularization.

An alternate way of representing firewall policies is by firewall decision diagram (FDD) [18, 25]. An FDD is a decision diagram where nodes are divided into levels, with each level corresponding to one field in a packet. This method is, of course, drastically different from using ACLs. It is unclear whether a policy specified in this form is easier to understand or modify for an administrator.

Policy chains/subroutines. The concepts of policy chains and subroutines exist in firewall products such as Linux netfilter [3] and SRX series firewalls by Juniper [2]. A sequence of rules can be organized into a subroutine and can be invoked from multiple places. This can improve the understandability of policies, especially when the same requirements are repeatedly applied, *e.g.*, the same sequence of rules are applied to multiple hosts.

This, however, does not provide the full advantage of modularization. There is no isolation among subroutines and the full policy, or among different chains specifically in netfilter. Policy chains and subroutines provide the mechanical support for modularization, without the methodology on how to modularize a policy. If one simply divides a long sequence of rules into multiple smaller ones that are chained together, that does not make the policy easier to understand.

Only-one-accept modules. In [5], a notion of modular firewall policy was introduced, where a policy is considered modular if it can be partitioned into multiple policy components $M_1, \ldots, M_r$ such that each packet is accepted by at most one component. In such an approach, a packet is accepted by the overall policy only if it is accepted by one component, and is denied otherwise. This approach is still inherently monolithic because of interactions among different components. As conflicts are still allowed, for example, one module may reject a packet whereas another module may allow it, when trying to understand the decision for a packet, one may still need to examine all components of a policy. There is also no logical basis for partitioning a policy into different modules. Finally, determining the slice of a policy, with respect to a specific packet or packet space, is not easier than in ACL.

3 Tri-Modularization Design Philosophy

In the context of software engineering, modularization signifies the concept of breaking up large, monolithic software source code and organizing them into smaller, reusable units based on the specific tasks these units implement. Modularization hence reduces the size of a program due to reusability, and makes a program easier to understand and debug, making it less error-prone and more reliable. Although the concept of modularization in firewall policies is very appealing, it is not obvious how to most effectively achieve this.

3.1 Requirements

In the context of firewall policies, a modularization approach would divide a policy into smaller pieces, which we call *modules*. To be able to analyze the effectiveness of different approaches of introducing modularization into policies, we identify the following requirements for a successful modularization approach.

Isolation among modules. The modules should be (at least partially) isolated. By isolation, we mean that the interactions among modules are limited and well defined. Only with adequate isolation, would it be possible to understand what each module achieves, without requiring to keep the details of other modules in one's mind. This also makes it possible to make local changes without unintended global side effects.

Logical partitioning. The criteria of partitioning should be simple and logical. That is, it should be easy to identify modules that are relevant to a particular situation. This and the isolation requirements together enable one to first have a global and high-level view of a policy, without understanding each module in depth, and then gradually refine the understanding by understanding the modules one by one.

Flexible partitioning structure. The partitioning should be flexible enough so that when a module becomes too large, one can break it up. This requirement is motivated by the dynamic nature of policies and aims at supporting *policy refactoring*. A policy often needs to evolve over time resulting in large modules which should then be broken up.

Human-computable policy slicing. To help understand policies, it is necessary to support mental policy slicing. In computer programming, *program slicing* [36] is the computation of the set of program statements (*i.e.*, the program slice) that may affect the values at some point of interest. In the context of a firewall policy, slicing can be done not only for a single packet, but also for some natural subspace of the whole space of possible packets. For intellectual manageability of policies, it is desirable that administrators

can mentally calculate relevant slices of a policy with respect to a given packet or packet space.

Deployability. If an approach can only be deployed with a new firewall product that provides specialized support for it, then the benefit of modularization can be exploited by that product's customers alone. On the contrary, if modularization can be adopted by someone who understands the approach when writing a policy with existing products, then it can be adopted widely.

3.2 Two Extremes of Expressing Policies

Our tri-modularization design is the result of our investigation of many real-world firewall policies and the analysis of how to express them in a succinct and intellectually manageable way. However, it is natural to ask what is the philosophy behind tri-modularization and why such a design is useful in practice.

Effectively expressing a function. Abstractly, a firewall policy is a function that maps a tuple of several input attribute values (*e.g.*, IP address, port) to a binary decision (*i.e.*, allow or deny). Our problem is similar to that of how to most effectively represent boolean functions. Standard ways of expressing boolean functions include truth tables, boolean formulas, and circuits. Firewall policies differ in that the input attributes are not boolean. Some of these attributes (such as IP addresses) can take a very broad range of values hence using truth tables for our purpose is infeasible.

The typical approach of using a rule list (or, ACL representation) is close in spirit to using a formula to express a function. The problem is that when the number of rules is large, the formula becomes complicated and difficult to understand.

Another approach that has been proposed is the *firewall decision diagram* (FDD) which partitions the whole packet space, attribute by attribute [18,25]. This is similar to using a circuit. The problem is that the circuits can become large, as it requires a large number of redundancies. In the case of FDD, partitioning is performed using all policy attributes.

Table 2 gives a running example policy. It is an abridged version of an actual policy used in a large-scale US-based IT organization. The complete policy is given in Table 6 in Appendix B and has 209 rules. To fix a misconfiguration error in the policy that we have found and the administrator has confirmed, two rules need to be added, resulting in a 211-rule policy. The added rules are rules 15 and 51 in Table 2.

The FDDs to represent the 211-rule policy have sizes varying from $2,500$ nodes to roughly $22,000$ nodes, depending on the order of the attributes. With the optimal attribute order, the FDD for the given policy has more than $2,500$ nodes. Even though an FDD representation of a firewall policy can contain many more rules than its ACL counterpart due to the partition of the packet space, it has the innate advantage that the following query can easily be calculated by a human user: *Is a particular packet allowed by the firewall policy*? However, neither ACL nor FDD enables a human user to have a global understanding of what the policy achieves. This is highly relevant to the incremental management of firewall policies.

3.3 Tri-Modularization Design

Our tri-modualization approach lies in the middle of the two extremes (*i.e.*, ACL and FDD) discussed above. It combines the advantage of partitioning the packet space by FDD and the advantage of succinctness enjoyed by ACL due to allowing conflicts in the policy. We will use the policy in Table 3–which is equivalent to that in Table 2—as an example when explaining our design. The first column in the table contains the rule numbers, to allow us to refer to them in our discussion whereas the last column contains explanations of rules or modules. We ignore the columns containing sPort and protocol as all rules in the policy have a value of "*" (*i.e.*, wildcard character) in these two fields. In the rest of this Section, when we say lines XX, we refer to Table 3.

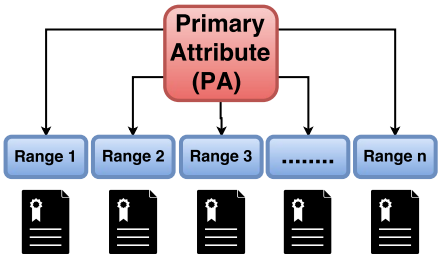

Figure 1: Design Philosophy

Primary attribute and primary modules. One natural approach to achieve isolation and logical partitioning is to require each module to cover a disjoint subset of the possible packets. Each packet is decided by one and only one such module. To ensure that a logical global structure exists and that it is straightforward to figure out which module a packet belongs to, we introduce the concept of ***primary attribute***. In our approach, one can choose either sIP or dIP as the "primary attribute", and a policy is divided into modules in such a way that each module covers a *disjoint range* for the primary attribute. We call such modules ***primary modules***. This is similar to partitioning in FDD but we restrict ourselves to partition along the primary attribute only (see Figure 1).

Choice of primary attribute. Through analyzing real-world policies, we have found that firewall policies essentially have three logical attributes: sIP, dIP, and service. The service is typically defined by the protocol information (*e.g.*, TCP, UDP, or ICMP) along with dPort. The sPort of most rules is essentially "don't care" (contains "*"). Using service as the primary attribute, however, does not support a flexible partitioning structure, because even if one limits to a single service, there are often too many rules to make one primary module difficult to understand. We have further observed that the ideal primary attribute for most policies is dIP, but some policies benefit more from using sIP as the primary attribute.

Representing primary modules. Once the policy is partitioned into disjoint ranges of the primary attribute value, our design exploits the succinctness of ACL due to conflicts in the rules. There are 4 primary modules in Table 3, each of which covers one of the disjoint ranges. PM1 (lines 11-13), PM2 (line 19) and PM3 (lines 20-22) cover single IPs "207.89.182.41", "207.89.182.248" and "207.89.182.57", respectively. PM4 covers a range "$71.121.90.128/26$", so it has rules relevant to "$71.121.90.128/26$" and "$71.121.90.154$" in the primary attribute. Moreover, a primary module may consist of one or more *primary* rules (primary rules cannot have "*" in the primary attribute field) or *instantiation* rules (instantiation rules are relevant to calling template or auxiliary modules). For example, line 13 is an instantiation rule for calling the template module TM1.

Auxiliary modules. One may desire a firewall policy to be fully partitioned into primary modules. For example, a policy may be divided into modules each of which covers a particular range for dIP. While this provides modularization, it can be undesirable, because there are often "*global*" rules that apply across all values in the primary attribute. For example, one often wants to blacklist certain hosts, or block specific ports, etc. When forcing all policies into primary modules, we have to duplicate these global policies in each module. When these rules need to change, one has to make changes to every copy of them. We call these rules that do not

No.	sIP	dIP	dPort	decision
1	71.100.64.0/19	*	*	deny
2	71.240.50.0/26	*	*	deny
3	71.206.182.0/24	*	*	deny
4	71.121.88.84	207.89.182.41	25	allow
5	71.121.92.96	207.89.182.41	25	allow
6	*	*	25	deny
7	*	*	137	deny
8	*	*	445	deny
9	*	*	135	deny
10	*	*	138	deny
11	71.14.116.1	71.121.90.184	1953-1954	allow
12	71.14.116.1	71.121.90.191	1953-1954	allow
13	71.14.116.1	207.89.176.60	1953-1954	allow
14	71.14.116.1	207.89.182.41	1953-1954	allow
15	71.14.116.1	207.89.182.248	1953-1954	allow
16	71.14.116.1	207.89.182.57	1953-1954	allow
17	71.14.116.1	71.121.90.128/26	1953-1954	allow
18	71.87.147.117	71.121.90.184	1950-1951	allow
19	71.87.147.117	71.121.90.191	1950-1951	allow
20	71.87.147.117	207.89.182.41	1950-1951	allow
21	71.87.147.117	207.89.182.248	1950-1951	allow
22	71.87.147.117	207.89.182.57	1950-1951	allow
23	71.87.147.117	71.121.90.128/26	1950-1951	allow
24	71.87.147.117	71.121.90.184	1960	allow
25	71.87.147.117	71.121.90.191	1960	allow
26	71.87.147.117	207.89.182.41	1960	allow
27	71.87.147.117	207.89.182.248	1960	allow
28	71.87.147.117	207.89.182.57	1960	allow
29	71.87.147.117	71.121.90.128/26	1960	allow
30	71.67.95.202	71.121.90.184	1960	allow
31	71.67.95.202	71.121.90.191	1960	allow
32	71.67.95.202	207.89.182.41	1960	allow
33	71.67.95.202	207.89.182.248	1960	allow
34	71.67.95.202	207.89.182.57	1960	allow
35	71.67.95.202	71.121.90.128/26	1960	allow
36	*	71.121.90.184	1953-1954	deny
37	*	71.121.90.191	1953-1954	deny
38	*	207.89.182.41	1953-1954	deny
39	*	207.89.182.248	1953-1954	deny
40	*	207.89.182.57	1953-1954	deny
41	*	71.121.90.128/26	1953-1954	deny
42	*	71.121.90.184	1950	deny
43	*	71.121.90.191	1950	deny
44	*	207.89.182.41	1950	deny
45	*	207.89.182.248	1950	deny
46	*	207.89.182.57	1950	deny
47	*	71.121.90.128/26	1950	deny
48	*	71.121.90.184	1960	deny
49	*	71.121.90.191	1960	deny
50	*	207.89.182.41	1960	deny
51	*	207.89.182.248	1960	deny
52	*	207.89.182.57	1960	deny
53	*	71.121.90.128/26	1960	deny
54	71.0.0.0/8	71.121.90.154	22	allow
55	71.0.0.0/8	71.121.90.154	80	allow
56	71.0.0.0/8	71.121.90.154	443	allow
57	71.0.0.0/8	71.121.90.154	5800-5809	allow
58	71.0.0.0/8	71.121.90.154	5900-5909	allow
59	71.0.0.0/8	71.121.90.154	3690	allow
60	*	71.121.90.154	*	deny
61	71.0.0.0/8	*	*	allow
62	71.67.94.12	207.89.182.27	55555	allow
63	71.121.92.53	207.89.182.179	52311	allow
64	207.89.182.142	207.89.182.57	179	allow
65	207.89.182.143	207.89.182.57	179	allow
66	71.0.0.0/8	*	80	allow
67	71.121.88.50	207.89.182.17	52311	allow
68	71.121.59.54	207.89.182.17	52311	allow
69	*	*	*	deny

Table 2: The original policy in ACL

belong to any primary module "*auxiliary rules*"; they can be easily identified because their primary attribute field contains "*".

We propose to group these auxiliary rules into what we call ***auxiliary modules*** based on the types of the rules. For example, all adjacent rules that block all traffic from some subnet are considered to be in one auxiliary module. This enables one to abstract the meaning of this module as "*some source IPs are blacklisted here*", when trying to form a global understanding of the policy, and dig into exactly which subnets are blacklisted only when necessary. We encourage policy authors to move auxiliary rules of the same type to be adjacent as much as possible, to reduce the number of auxiliary modules as much as possible. Lines 8-10, 14-18, 31, and 32 are examples of auxiliary modules.

Subroutine

	sIP	dIP	dPort	decision	Annotation
1	71.14.116.1	$	1953-1954	allow	**TM1**
2	71.87.147.117	$	1950-1951	allow	
3	71.87.147.117	$	1960	allow	
4	71.67.95.202	$	1960	allow	
5	*	$	1953-1954	deny	
6	*	$	1950	deny	
7	*	$	1960	deny	
-	*	$	*	return	

Main policy

	sIP	dIP	dPort	decision	Annotation
8	71.100.64.0/19	*	*	deny	**AM1**
9	71.240.50.0/26	*	*	deny	
10	71.206.182.0/24	*	*	deny	
11	71.121.88.84	207.89.182.41	25	allow	**PM1** with IP
12	71.121.92.96	207.89.182.41	25	allow	207.89.182.41
13	*	207.89.182.41	*	**TM1**	
14	*	*	25	deny	**AM2**
15	*	*	137	deny	
16	*	*	445	deny	
17	*	*	135	deny	
18	*	*	138	deny	
19	*	207.89.182.248	*	**TM1**	**PM2** with IP 207.89.182.248
20	*	207.89.182.57	*	**TM1**	**PM3** with IP
21	207.89.182.142	207.89.182.57	179	allow	207.89.182.57
22	207.89.182.143	207.89.182.57	179	allow	
23	*	71.121.90.128/26	*	**TM1**	**PM4** with range
24	71.0.0.0/8	71.121.90.154	22	allow	71.121.90.128/26
25	71.0.0.0/8	71.121.90.154	80	allow	
26	71.0.0.0/8	71.121.90.154	443	allow	
27	71.0.0.0/8	71.121.90.154	5800-5809	allow	
28	71.0.0.0/8	71.121.90.154	5900-5909	allow	
29	71.0.0.0/8	71.121.90.154	3690	allow	
30	*	71.121.90.154	*	deny	
31	71.0.0.0/8	*	*	allow	**AM3**
32	*	*	*	deny	**AM4**

Table 3: The modularized version of the example policy in Table 2

Template modules. In many large policies, a sequence of rules may apply to many different IP addresses (*e.g.*, applicable to all webservers). To enable reuse, we allow a third kind of module dubbed ***template modules***. A template module consists of one or more template rules that may be applied to many different IP addresses. For example, lines 1-7 form a template module, with "*TM1*" as its name. This template module is invoked in lines 13, 19, 20, and 23 for IP addresses 207.89.182.41, 207.89.182.248, 207.89.182.57, and 71.121.90.128/26, respectively. Template rules have their primary attribute field being "$", indicating that this is a *formal argument* and can be instantiated when this template module is invoked. We use "$" instead of "*" to differentiate template rules from auxiliary rules.

Putting it all together. As our modularization approach uses three kinds of modules, we call it a tri-modularization design. The high-level idea of our approach is illustrated in Figure 1. The primary attribute is partitioned into disjoint ranges each of which is covered by one primary module. Each primary module is essentially an ACL, and may call auxiliary modules and template modules. Both auxiliary modules and template modules are reusable, and they can be called by multiple primary modules.

In Table 3, there are 4 primary modules, 4 auxiliary modules, and 1 template module. There are no interactions among primary modules, while there are some limited interactions between primary modules and auxiliary/template modules. To evaluate a packet, one only needs to look at the primary module that matches the

packet, template modules called by the primary module, and auxiliary modules, safely ignoring other primary modules. For example, for a packet matching PM3, one may only check the following modules in sequence: AM1, AM2, TM1, PM3, AM3, and AM4. The evaluation will stop whenever the packet's fate can be determined. Thus, a relevant slice of a policy can be easily computable by a human in our design.

3.4 Deployability of Tri-Modularization

The next aspect of tri-modularization we investigate is its deployability. The relevant questions in this regards are: *How deployable the tri-modularization approach is? Can existing firewall products support it?* We observe that some existing products supporting chains/subroutines, such as Linux netfilter/iptables can be used to implement the modules we proposed, especially the reusable auxiliary modules and template modules.

In netfilter, a rule's *target* can be a user-defined chain. When a packet matches a rule whose target is a user-defined chain, the rules in the chain will be evaluated against the packet. If the chain does not deny or allow the packet after the traversal of the chain is done, the next rule in the current chain will be evaluated. Therefore, users can define a new chain for either an auxiliary module or a template module, and then write normal ACL rules whose target is this chain and whose matching conditions are the input arguments when calling the chain. In Table 3, we use similar syntax of chains in netfilter. TM1 can be viewed as a new user-defined chain. There are multiple places where this chain will be jumped to, such as in PM1, PM2, PM3 and PM4. Take PM1 as an example, when the matching condition in line 13 is satisfied, we will jump to TM1 and the rules there will be evaluated.

4 Tri-Modularization of Legacy Policies

Although network administrators can easily use the concept of tri-modularization when writing a new firewall policy, one of the main challenges of tri-modularization's adaptability is the legacy policies. To convert legacy policies to their tri-modularized form and hence enable adaptability, we present an automatic translation procedure, which at a high level has the following steps.

1. **Determining primary attribute**: decide which field (*e.g.*, sIP, dIP) is used as the primary attribute (PA).
2. **Removing redundancies**: identify and remove redundant rules. Removing redundant rules in ACL is straightforward and due to space limitations we do not describe it here.
3. **Creating auxiliary modules**: reorder the rules and assemble auxiliary rules of the same type together.
4. **Creating primary modules**: generate a set of disjoint ranges of PA, each of which will be covered by a primary module; reorder the rules and try to sort primary rules based on the PA values, and then create suitable primary modules.
5. **Creating template modules**: identify frequent rule patterns in the policy and use them to create template modules.

We have developed a tool dubbed ModFP which can help administrators perform the above steps automatically. In the rest of the section, we describe the above steps and the key algorithms.

4.1 Choosing the primary address

The main heuristic in choosing the primary attribute is that we want fewer rules where the primary attribute value is a "*" so that there are fewer auxiliary rules. As primary rules are partitioned into modules that are disjoint, they can be understood independently. As a result, a policy that has many primary rules is not necessarily much more difficult to understand. However, as auxiliary rules apply to all following primary rules (resp., modules), trying to decrease the number of auxiliary rules (resp., modules) can increase the intellectual manageability of a policy significantly. Examining the policy in Table 2, we can see 25 rules have "*" in the sIP field whereas only 11 rules have "*" in the dIP field. We thus choose dIP as the primary attribute. We have observed that—for a dozen or so real-world policies we have converted to their modularized format—dIP turns out to be a better choice as the primary attribute, likely because most of the rules are controlling traffic from outside the network to hosts inside the network, and thus are better grouped by dIP. We also point out that one can always try to modularize a policy first with dIP as primary attribute, then with sIP or some other fields as primary attribute, and compare the results.

4.2 Creating Auxiliary Modules

Recall that in a policy there may be "*global*" rules that do not belong to any specific primary module and instead can apply across primary modules following them. We want to assemble such auxiliary rules of the same type together to form auxiliary modules. This will reduce the number of auxiliary modules, and also make auxiliary modules more manageable. For this purpose, we need to move rules around without changing the policy semantics. We also want to move primary rules that are about the same IP addresses (or the same prefix) together as much as possible to create primary modules, as described in the next section. Therefore, we now introduce how to reorder rules.

4.2.1 Rule reordering

We first introduce the notion of what it means for a rule to be *switchable* with another rule.

Definition 1 (Switchable rules)**.** *For a given policy, we say that rules r_i and r_j are switchable iff r_i and r_j are adjacent rules and switching their order has no impact on the semantics of the policy.*

Two rules r_i and r_j are switchable if and only if either they have the same decision or their sets of applicable packets are disjoint. When two rules have different decisions and their sets of applicable packets overlap, if these two rules are the first two rules in a policy, switching their order will change the decisions on packets that they both are applicable to.

To determine whether certain rules can be moved to be adjacent, we need to know to what extent these rules can be moved around without changing the policy semantics. Given a policy expressed as a list of rules $\mathcal{R}$ where each rule has an index, we use $\mathsf{pre}(r_j)$ to denote the set of rules that should come before the rule r_j, the rule with index j, when we move the rules around. This set can be computed as follows. Going up from r_j, ignore any rule that is switchable with r_j. When we reach the first rule r_i that is not switchable with r_j, if we want to further move r_j up, we need to move r_i together with r_j, we thus add r_i to our set and now check whether they can be moved up together. We can similarly define

Algorithm 1: Creating auxiliary modules in $\mathcal{R}$

```
   Input: A rule set R
1  foreach r_i ∈ R do
2      if r_i is an auxiliary rule then
3          Create an auxiliary module am with r_i only
4          foreach r_j after r_i do
5              if r_j is an auxiliary rule ∧ r_i.type = r_j.type then
6                  if post(am) ∩ pre(r_j) = ∅ then
7                      Merge r_j into am
```

$\mathsf{post}(r_j)$, which is the set of rules that appear after r_j such that

r_j cannot be moved past them without changing the policy semantics. $\mathsf{pre}(r_j)$ and $\mathsf{post}(r_j)$ can be calculated using Algorithms 3 and 4, respectively (see Appendix A). For instance, according to the policy in Table 2, $\mathsf{pre}(r_3) = \emptyset$ (r_3 refers to the rule in line 3), $\mathsf{pre}(r_6) = \{r_4, r_5\}$, and $\mathsf{post}(r_{66}) = \{r_{69}\}$. The definitions of $\mathsf{pre}(\cdot)$ and $\mathsf{post}(\cdot)$ for a rule can be generalized to a sequence to rules. Given a sequence of rules $\mathcal{R}$, $\mathsf{pre}(\mathcal{R})$ denotes the set of rules that should come before all the rules in $\mathcal{R}$, and $\mathsf{post}(\mathcal{R})$ denotes the set of rules that should come after all the rules in $\mathcal{R}$.

Lemma 1. *Given two sequences of rules $\mathcal{R}_1$ and $\mathcal{R}_2$, where $\mathcal{R}_1$ appears earlier than $\mathcal{R}_2$, they can be merged together if there is no rule that belongs both to $\mathsf{post}(r_i)$ for some $r_i \in S_1$ and to $\mathsf{pre}(r_j)$ for some $r_j \in S_2$, i.e., $\mathsf{post}(\mathcal{R}_1) \cap \mathsf{pre}(\mathcal{R}_2) = \emptyset$.*

4.2.2 Merging Auxiliary Rules

According to the primary attribute chosen by users, primary rules and auxiliary rules can be distinguished. Recall that rules with the value of "*" in the primary attribute are auxiliary rules. Further, auxiliary rules can be categorized into different types after the primary attribute is set, see Table 4. "*" means that an auxiliary rule can take any values in the field, while "Specific" means that an auxiliary rule has a specific value in that field, such as a specific IP address, subnet, or service. Algorithm 1 can be applied to generate auxiliary modules based on their types. For each auxiliary rule, we try to merge it with other auxiliary rules with the same type.

4.3 Creating Primary Modules

The objective of creating primary modules is to partition the policy into disjoint sections such that each of the sections can be understood and managed independently with little to no interaction with other portions of the policy. Each primary module contains rules that cover a specific range of primary attribute values. The main challenge is to determine what these disjoint ranges of primary attribute values are. Once such ranges are generated, the next challenge is to group rules that falls into a specific interval together.

4.3.1 Range Generation

Threshold of primary module size. The size of each primary module should not become too large. For example, an administrator may want to have primary modules each of which includes no more than δ rules (*e.g.*, 20). Therefore, users are required to set a threshold δ for how many rules can be in a primary module. If a primary module covering a range has rules more than δ, it means that the range should be further divided. However, in case a range covers a single IP address and cannot be further divided, the above approach is not applicable. In this case, the value of δ should be increased to solve the problem. Therefore, the value of δ needs to be adjusted to the maximal size of primary modules covering single IP addresses, if needed.

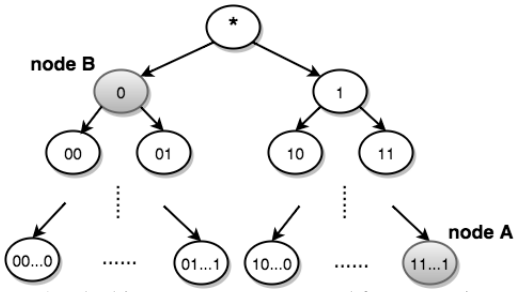

Figure 2: The binary tree structure used for generating ranges

Generating ranges. Algorithm 2 is used to generated a set of disjoint ranges for a rule set with auxiliary modules generated already. The initial input is a rule set and an empty string meaning "*" (*i.e.*, the whole range in the primary attribute). This algorithm uses a tree structure, as show in Figure 2. The left child of a node is obtained by appending one more bit "0" to the node, and the right child by appending "1" to the node. A node in the tree represents either a single value or a range in the primary attribute. In Figure 2, since the primary attribute is dIP, node A represents a single IP (32 one's, *i.e.*, 255.255.255.255). Node B is "0" meaning that the first bit of the total 32 bits is 0 and the other 31 bits can be anything. This node hence represents a range $[0, 2147483647]$. Algorithm 2 uses the following cost function as a utility function.

Given a rule set $\mathcal{R}$ and a range I, the cost function $\mathsf{cost\_func}(\mathcal{R}, I)$ outputs the number of primary modules that are needed to cover the range I. $\mathsf{cost\_func}(\mathcal{R}, I)$ returns ∞ if one of the primary modules is required to have more than δ number of rules while creating primary modules to cover the range I. $\mathsf{cost\_func}(\mathcal{R}, I)$ returns $n \in \mathbb{N}$, otherwise.

Algorithm 2: $getRanges(\mathcal{R}, root)$

Input: A rule set $\mathcal{R}$ with auxiliary modules generated already, and a root range $root$
Output: A set of disjoint ranges $ranges$

```
1  list = ∅
2  if root.length > num_of_bits_in_primary_attr then
3  |  Return list
4  cost = cost_func(R, root)
5  if cost < ∞ then
6  |  if cost > 0 then
7  |  |  if left child's cost equals to root's cost then
8  |  |  |  list = list + getRanges(R, root + "0")
9  |  |  else if right child's cost equals to root's cost then
10 |  |  |  list = list + getRanges(R, root + "1")
11 |  |  else
12 |  |  |  Add root to list
13 |  Return list
14 else
15 |  list = list + getRanges(R, root + "0")
16 |  list = list + getRanges(R, root + "1")
17 Return list
```

4.3.2 Merging Primary Rules

After a set of disjoint ranges are obtained, we can create primary modules by merging together primary rules overlapping with the same range. For each of the ranges, when the first primary rule that is overlapping with the range is found, a primary module is created having only this rule in it. After that, any primary rules that overlap with the range will be appended to this primary module. When trying to move a primary rule into the primary module, there may be primary rules, primary modules, and auxiliary modules lying between them. We can safely ignore those primary rules and primary modules because of the benefit of having disjoint ranges. For auxiliary modules that are switchable with the primary rule to be inserted, we can safely ignore them as well. For other auxiliary modules, however, we need to create instantiation rules for calling those auxiliary modules, put the instantiation rules before the primary rule, and then append them together to the primary module.

A primary rule is overlapping with a range, if the primary rule (1) equals to the range, (2) is a subset of the range, (3) is a super set of the range, or (4) intersects with the range. For the first 2 cases, we just simply append the primary rule to the appropriate primary module. For the last 2 cases, we only add the intersecting parts of the primary rule to the appropriate primary module. In addition to

that, we need to duplicate the rule for the non-intersecting parts of the rule, and keep the duplicate rule(s) in the original rule's place.

4.4 Creating Template Modules

In a policy, some rules may appear multiple times in a primary module or different primary modules with distinct primary attribute values. We want to create templates for those rules and form template modules so that they can be reused. The problem of creating template modules has similarities with the role mining problem [22, 30, 32, 35].

4.4.1 Frequent Rule Pattern Mining Problem

A list of *similar* primary rules may appear in multiple primary modules in a firewall policy, or even appear in a primary module multiple times. We call such a list of rules a *rule pattern* or just *pattern*. Note that *primary* rules that differ only in the primary attribute are regarded as similar rules w.r.t. a pattern. Template modules can be instantiated by invoking it with different values/ranges in the primary attribute field. The notion of template module is similar to the concept of subroutines in programming. Instantiation of a template module is similar to subroutine invocation. We now state the frequent rule pattern mining problem.

Definition 2 (Frequent rule pattern mining problem)**.** *Given a list of primary modules and an argument Ψ_{min} that specifies the minimum number of occurrences of a rule pattern, the frequent rule pattern mining problem is to find all rule patterns whose support (i.e., the number of occurrences) is at least Ψ_{min} from those primary modules. Such patterns are called* frequent rule patterns.

Frequent itemset mining algorithms like Apriori [7] can be applied for mining rule patterns from primary modules. Each primary module is translated to a transaction, and each rule in a primary module is an item. Again, primary rules that differ only in the primary attribute field are regarded as the same rule. In this way, we obtain the input for our modified Apriori algorithm.

We modify the Apriori algorithm from the following two aspects. Firstly, items in an itemset are order-insensitive in the original Apriori algorithm. However, rules in a pattern are order-sensitive, and there may be other rules in between those rules in the pattern. In our modified Apriori algorithm, whenever a potential frequent itemset (*i.e.*, a pattern) is found, we need to go back to the primary modules where the pattern appears and check if the pattern can actually occur in those primary modules. For example, when a candidate itemset $\{a, b, c\}$ is found, we need to go back to a primary module where it appears to check if its support should be increased. Assume that the primary module includes rules $\{A, B, X, Y, C\}$ ($a-A$, $b-B$, and $c-C$ are similar rules). The pattern $\{a, b, c\}$ appearing in the primary module as rules $\{A, B, C\}$. If rules $\{X, Y\}$ in between the pattern can be moved away, the support for the pattern will be increased; otherwise, the support will not be increased.

Secondly, in the original Apriori algorithm, an itemset in a transaction will be counted only once even if it appears more than once in the transaction. Since a pattern may appear in a primary module multiple times, and if a template module is created using this pattern, the primary module should call the template module using an instantiation rule whenever the pattern appears. For this purpose, in our modified Apriori algorithm, all occurrences of an itemset in a transaction will contribute to its support. For example, assume that a candidate itemset is $\{a, b, c\}$, and a primary module where it appears consists of rules $\{A, B, C, A^{'}, B^{'}, C^{'}\}$. This candidate appears twice, so its support should be increased by 2 instead of 1.

The value of Ψ_{min} can be specified by users. For example, if the value is set to 3, it means that a frequent itemset should appear at least 3 times (in different transactions and/or within the same transaction). The value of Ψ_{min} should not be too small or too large. If it is too small, too many frequent itemset will be generated; if it is too large, perhaps no frequent itemsets will be found. We modify the Apriori implementation in SPFM [4] for our purpose.

4.4.2 Template Module Assignment Problem

Given the result of our modified Apriori algorithm which is a list of patterns with lengths from 1 to k, where k is the length of the longest pattern(s) found, we want to find a subset of these patterns optimizing the number of rules reduced by creating template modules for them. We first define the assignment problem.

Definition 3 (Template module assignment problem)**.** *Variables x_i^j is created for each pair of a pattern P_i and a primary module PM_j where the pattern appears. The assignment problem is to assign either 0 or 1 to each of these variables, with 1 meaning that the pattern should be used in the primary module and 0 meaning that it should not, so that the total number of rules reduced by using the patterns assigned with 1's will be maximal.*

The number of rules reduced by using a given pattern P_i can be calculated using the following formula: $|P_i| * sup_i - |P_i| - sup_i$, where $|P_i|$ is length of the pattern, and sup_i is the number of occurrences of P_i. The intuition behind the formula is that $|P_i| * sup_i$ rules can be saved, but some penalties also need to be deducted since an extra template module with $|P_i|$ rules and sup_i instantiation rules are created.

Using the longest pattern(s) will not always yield the optimal result. For example, suppose that the longest pattern is $P_1 = \{a, b, c, d, e, f\}$ with $sup_1 = 3$. However, there are 2 shorter patterns $P_2 = \{a, b, c, d\}$ with $sup_2 = 5$ and $P_3 = \{e, f\}$ with $sup_3 = 4$. Using patterns P_2 and P_3 together is better than using P_1 only, since the number of rules reduced is $(4*5-4-5)+(2*4-2-4) = 13$ in the former case instead of only $6*3-6-3 = 9$ in the latter case. Finding an optimal solution among the patterns found by our modified Apriori algorithm is not trivial. We model the assignment problem as an integer programming problem, and use IBM CPLEX optimizer [1] to solve the problem.

Suppose that m patterns (*i.e.*, potential template modules to be created) are found. In Figure 4 in Appendix A, we show an optimizer model with 4 patterns and 3 primary modules. Each pattern is used by one or more primary modules. This information is obtained by the modified Apriori algorithm mentioned above. Note that the support of a pattern may be different from the number of primary modules where it occurs, since it may appear more than once within a primary module. Each edge between a pattern and a primary module where it appears is represented by a variable $x_i^j \in \{0, 1\}$. $x_i^j = 1$ when pattern P_i will be eventually used by primary module PM_j, and $x_i^j = 0$ otherwise. The goal of the optimizer is to find the assignments of those binary variables to optimize our objective function described in Figure 3.

There are several constraints during the optimization. **(1)** n_i is the actual number of occurrences of the pattern P_i, and it should satisfy $0 \leq n_i \leq sup_i$, where sup_i is the support (*i.e.*, the total number of occurrence) of P_i. If all values of x_i^j ($j = 1 \ldots k$, where k is the number of primary modules where P_i appears) is assigned to be 1, then $n_i = sup_i$. If all x_i^j is assigned to be 0, then $n_i = 0$. **(2)** $h(n_i)$ is a function to decide whether a template module should be created for pattern P_i. When $n_i = 0$, it means that pattern P_i will not be used by any primary modules where it appears, $h(n_i) = 0$ so no template module will be created for this pattern; otherwise $h(n_i) = 1$ so one template module will be created. **(3)** A primary module can not simultaneously use patterns that are overlapping. Therefore, for each primary module PM_j, the

$$\text{maximize} \quad \sum_{i=1}^{m} n_i \times |P_i| - h(n_i) \times |P_i| - n_i$$

$$\text{subject to} \quad n_i = \sum_{j=1}^{k} sup_i^j \times x_i^j, \text{ where } x_i^j \in \{0,1\}$$

$$0 \leq n_i \leq sup_i$$

$$h(n_i) = \begin{cases} 0 & \text{if } n_i = 0 \\ 1 & \text{otherwise} \end{cases}$$

$0 \leq x_s^j + x_t^j \leq 1$ for any pair of patterns that are overlapping and co-exist in the same PM_j

Figure 3: Formulation of template module assignment problem

constraint $0 \leq x_s^j + x_t^j \leq 1$ should be satisfied, which means that at most one of the overlapping patterns P_s and P_t can be eventually used by PM_j.

4.4.3 Generating Template Modules

A template module is created for each pattern P_i that is used by at least one primary module (*i.e.*, at least one variable x_i^j for some j is assigned to be 1) such that the pattern inside every primary module where it appears and for which the corresponding x value is 1 will be replaced by an instantiation rule. Moreover, a pattern may appear in a primary module multiple times, so each occurrence of the pattern will be replaced by an instantiation rule accordingly.

5 Evaluation

Using the tri-modualization approach, we implemented ModFP in Java, and used it to examine a dozen or so real-world policies with sizes from dozens to hundreds. We show the results of the 4 largest policies in Table 5, among which *Policy 3* is the complete and corrected version of the policy we presented in Table 2. Three of these policies belong to an academic institution and have been used in prior work on firewall policies. The remaining policy (*i.e.*, *Policy 3*) belongs to a large-scale US-based IT company.

5.1 Effect on Number of Rules

By utilizing ModFP to convert *Policy 1*, *Policy 3*, and *Policy 4* to their modularized format, the number of rules is reduced by 64.3%, 68.7%, and 25.3%, respectively. For *Policy 2*, the number of rules is increased by 1. For all policies, ModFP only take seconds to convert them into the modularized form. For *Policy 2*, the number of rules increases from 87 to 88 after the conversion because of the following reasons. First, there is only 1 redundant rule in the policy, so removing redundancies does not decrease the number of rules much. Second, by creating a template module for a pattern with a length of 2 and a support of 3, only 1 ($= 2 \times 3 - 2 - 3$) rule is reduced. Third, 3 instantiation rules are created when some rules are merged into the primary module they belong, since there are 3 auxiliary modules that are not switchable. Therefore, eventually the number of rules is increased by 1.

For *Policy 4*, the number of rules does not decrease dramatically like in the cases of *Policy 1* and *Policy 3*. After removing redundancies, the number of rules decreases from 661 to 572. After creating primary modules 24 rules are added, since a destination subnet overlaps with multiple disjoint ranges, and the set of rules related to this subnet needs to be duplicated. And then the number of rules decreases by 102 by using template modules. Therefore, the number of rules decreases by 167 (*i.e.*, 25.3%) in total.

Our tri-modualization approach enjoys additional advantages on top of reducing the number of ACL rules.

5.2 Additional Advantages

Enabling a global understanding of a policy. Several design features of our tri-modualization approach aims at enabling a global understanding of a policy. Primary modules force one to group related rules together. Auxiliary modules group rules of the same type together. With a policy in its modularized form, one can mentally partition a potentially very large number of rules into meaningful modules, to have a global mental picture of the overall policy. One can hence provide a verbal summary of what the policy means and attempts to reason about it.

For example, for the policy in Table 2, we came up with the summary below based on its modularized form in Table 3. First, a list of source IPs are blacklisted. Then for the host 107.89.182.41, beyond "TM1", it also has port 25 open to two other hosts. "TM1" allows traffics to ports 1950-1951, 1953-1954, and 1960 from some specific IPs, and otherwise blocks traffic to ports 1950, 1953-1954, and 1960. Next "AM2" blocks ports 25, 135, 137, 138, and 445. Then the template "TM1" is applied to 3 other IP addresses and subnets. For the host 207.89.182.248, only "TM1" is applied. For the host 207.89.182.57, it has port 179 open to two hosts beyond "TM1". Primary Module "PM4" covers the range "71.121.90.128/26" , so subnet 71.121.90.128/26 and host 71.121.90.154 are covered by this range. "TM1" applies to subnet 71.121.90.128/26. Host 71.121.90.154 is most special: traffics from 71.0.0.0/8 to ports 22, 80, 443, 3690, 5800-5809, and 5900-5909 are allowed, and everything else is blocked. Finally, everything from 71.0.0.0/8 is allowed, and everything else is blocked.

We do not think it is feasible to come up with a similar description from the policy in its original form.

Making policy errors easier to identify. The modular nature of policies make policy configuration errors manifest themselves.

We have converted a dozen or so real-world firewall policies into their modularized form using ModFP. For every large policy we have examined, we have found clear errors as well as strange features that we conjecture to be errors. For the issues we have found with the complete version of the policy in Table 2 (see Table 6 in Appendix B for the complete policy), we have checked with the system administrator, and include the responses here as well.

There are a number of redundancies. For example, lines 202-203 and 206-208 are shadowed by line 201. Line 41 can be removed, as any packet it accepts will reach line 201 and is accepted. Further, since IP addresses 71.121.90.184 and 71.121.90.191 are in the subnet of 71.121.90.128/26, lines 27 and 28 can be removed because of line 50, and 12 other rules are in the same situation. The system administrator's comment on this is that "*The overshadowing is likely due to the number of different people who have access to the firewall policies*."

Less obvious issues can be found as well. Lines 27-193 in Table 6 correspond to a template module. We found that we could not apply the template module to host 207.89.182.248 because two rules (line 1 and line 7 in Table 3) are missing. That is, while lines 1-7 apply to 20+ other IP addresses, only lines 2-6 apply to 207.89.182.248. We found this rather strange, since there is a rule blocking ports 1953-1954 for all traffic, but no rule allowing the ports for certain specific source hosts. And there is a rule allowing port 1960 for certain specific source hosts, but no rule blocking it for all traffic. The system administrator confirms that this IP does not seem to be an active device in the DMZ anymore, and this is likely the result of incomplete cleanup processes.

Another issue is with the template module itself. Overall the

Source IP	Service	Decision
*	Specific	allow
*	Specific	deny
Specific	*	allow
Specific	*	deny
Specific	Specific	allow
Specific	Specific	deny
*	*	allow
*	*	deny

Table 4: Types of auxiliary rules (when dIP is the primary attribute)

Policy	ACL Rules	Modularized Policy				Translation time in seconds				
		PMs	AMs	TMs	Rules	Removing Redundancy	Creating AMs	Creating PMs	Creating TMs	Total
1	42	4	3	1	15	0.046	0.005	0.084	0.121	0.256
2	87	8	6	1	88	0.183	0.017	0.120	0.138	0.458
3	211	17	4	1	66	0.294	0.048	0.226	1.356	1.924
4	661	20	3	10	494	0.551	0.089	0.251	18.460	19.351

Table 5: Experimental Results

intention seems to be that for ports 1950, 1951, 1953, 1954, and 1960, only traffic from a specific host is allowed, and traffic from all other hosts is denied. However, Line 2 in Table 3 allows port 1950-1951 traffic from one specific host, but line 6 blocks only port 1950, and not 1951. A further piece of evidence is that if this is indeed intended, then rule 2 needs to mention only 1950, since as it is, port 1951 will be opened to all hosts in the $71.0.0.0/8$ subnet according to Line 31. Missing 1951 in line 6 was also confirmed to be an oversight. Some of the other comments we have received from the system administrator are:

We don't expend any effort to make the firewall rules easy to read or understand, and in fact we don't generally look at the entire set at all.

If we had software that made it easier to view the rule sets, and make changes to them efficiently, then we would probably have a cleaner set of firewall rules.

Enabling piece-by-piece understanding of a policy. As primary modules cover disjoint ranges, at most one primary module is applicable to each packet. Thus for each packet, one can consider only the auxiliary and template modules (if any), and at most one primary module. To understand the behavior of certain packets, one can quickly decide which primary modules would be applicable and ignore the rest of the primary modules.

Enabling policy refactoring. Our approach enables policy refactoring in two ways. First, template modules enable the definition of reusable templates that can be applied multiple times, similar to reusable subroutines in programming. Second, when a primary module becomes too large and complicated, one can divide it into multiple primary modules, each covering a smaller range.

6 Conclusion

Utilizing the idea of modular programming and code refactoring, we have introduced the tri-modularization design of firewall policies, which consist of three types of modules (*i.e.*, primary, auxiliary, and template). Our approach provides helpful abstraction and makes the policy more understandable and manageable. It also naturally supports policy refactoring in the authoring process. It can significantly reduce the number of rules in a policy and can also make configuration errors stand out and easier to identify. We present algorithms for converting legacy firewall policies in ACL to their tri-modular form, and also present a tool ModFP that automates the conversion. We have shown that using our approach one can understand complex real-world policies as well as identifying subtle errors, which are confirmed by the system administrator.

7 Acknowledgments

This work was supported by the Science of Security Lablet Program of National Security Agency under Grant No. H98230-14-C-0139, and by a gift grant from IBM Research. We also thank Prof. Alex Liu for providing us with some firewall policies.

8 References

[1] IBM CPLEX optimizer. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[2] Juniper. http://www.juniper.net/.

[3] Netfilter. http://www.netfilter.org/.

[4] SPMF: An Open-Source Data Mining Library. http://www.philippe-fournier-viger.com/spmf/.

[5] H. B. Acharya, A. Joshi, and M. G. Gouda. Firewall modules and modular firewalls. In *ICNP'10*, pages 174–182, 2010.

[6] P. Adão, C. Bozzato, G. Dei Rossi, R. Focardi, and F. Luccio. Mignis: A semantic based tool for firewall configuration. In *CSF'14*, pages 351–365, 2014.

[7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB '94*, pages 487–499, 1994.

[8] E. Al-Shaer and H. Hamed. Firewall policy advisor for anomaly detection and rule editing. In *IM'03*, pages 17–30, 2003.

[9] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM'04*, 2004.

[10] E. Al-Shaer and H. Hamed. Modeling and management of firewall policies. *IEEE TNSM*, 1-1, 2004.

[11] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE JSAC*, 23(10), 2005.

[12] E. S. Al-shaer and H. H. Hamed. Design and implementation of firewall policy advisor tools. Technical report, 2002. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.134.3344.

[13] F. Baboescu and G. Varghese. Fast and scalable conflict detection for packet classifiers. *Comput. Netw.*, 42(6):717–735, 2003.

[14] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM TOCS*, 22(4):381–420, 2004.

[15] F. Chen, A. X. Liu, J. Hwang, and T. Xie. First step towards automatic correction of firewall policy faults. *ACM TAAS*, 7(2):27:1–27:24, 2012.

[16] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[17] K. Golnabi, R. K. Min, L. Khan, and E. Al-Shaer. Analysis of firewall policy rules using data mining techniques. In *NOMS'06*, pages 305–315, 2006.

[18] M. G. Gouda and A. X. Liu. Structured firewall design. *Comput. Netw.*, 51(4):1106–1120, 2007.

[19] J. D. Guttman. Filtering postures: Local enforcement for global policies. In *IEEE S&P'97*, pages 120–129, 1997.

[20] J. D. Guttman and A. L. Herzog. Rigorous automated network security management. *Int. J. Inf. Sec.*, 4(1-2):29–48, 2005.

[21] J. Hwang, T. Xie, F. Chen, and A. X. Liu. Systematic structural testing of firewall policies. *IEEE TNSM*, 9(1):1–11, 2012.

[22] M. Kuhlmann, D. Shohat, and G. Schimpf. Role mining - revealing business roles for security administration using data mining technology. In *SACMAT '03*, pages 179–186, 2003.

[23] A. X. Liu. Firewall policy change-impact analysis. *ACM Trans. Internet Technol.*, 11(4):15:1–15:24, 2008.

[24] A. X. Liu and M. G. Gouda. Complete redundancy detection in firewalls. In *DBSec'05*, pages 193–206, 2005.

[25] A. X. Liu and M. G. Gouda. Diverse firewall design. *IEEE TPDS*, 19(9):1237–1251, 2008.

[26] A. X. Liu and M. G. Gouda. Firewall policy queries. *IEEE TPDS*, 20(6):766–777, 2009.
[27] R. Marmorstein and P. Kearns. A tool for automated iptables firewall analysis. In *USENIX ATC'05*, pages 71–81, 2005.
[28] R. Marmorstein and P. Kearns. Firewall analysis with policy-based host classification. In *LISA '06*, pages 41–51, 2006.
[29] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. IEEE S&P'00, pages 177–187, 2000.
[30] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo. Mining roles with semantic meanings. In *SACMAT'08*, pages 21–30, 2008.
[31] A. D. Santis, A. Castiglione, U. Fiore, and F. Palmieri. An intelligent security architecture for distributed firewalling environments. *JAIHC*, 4(2):223–234, 2013.
[32] J. Schlegelmilch and U. Steffens. Role mining with ORCA. In *SACMAT'05*, pages 168–176, 2005.
[33] A. Tongaonkar, N. Inamdar, and R. Sekar. Inferring higher level policies from firewall rules. In *LISA'07*, pages 17–26, 2007.
[34] T. E. Uribe and S. Cheung. Automatic analysis of firewall and network intrusion detection system configurations. *Journal of Computer Security*, 15(6):691–715, 2007.
[35] J. Vaidya, V. Atluri, and Q. Guo. The role mining problem: Finding a minimal descriptive set of roles. In *SACMAT'07*, pages 175–184, 2007.
[36] M. Weiser. Program slicing. In *ICSE'81*, pages 439–449, 1981.
[37] A. Wool. Architecting the lumeta firewall analyzer. In *SSYM'01*, 2001.
[38] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, June 2004.
[39] A. Wool. Trends in firewall configuration errors: Measuring the holes in swiss cheese. *IEEE Internet Computing*, 14(4):58–65, July 2010.
[40] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *IEEE S&P'06*, pages 199–213, 2006.

APPENDIX

A Algorithms of ModFP

Algorithms 3 and 4 show how to calculate $\mathsf{pre}(r_j)$ and $\mathsf{post}(r_j)$, respectively.

Algorithm 3: Calculate $\mathsf{pre}(r_j)$

Input: A rule r_j
$s = \{r_j\}$
for $i \leftarrow j-1$ **to** 0 **do**
  if r_i *is not switchable with some rule in* s **then**
    Add r_i to s
$\mathsf{pre}(r_j) = s \setminus \{r_j\}$

Algorithm 4: Calculate $\mathsf{post}(rule_j)$

Input: A rule r_j
$s = \{r_j\}$
for $i \leftarrow j+1$ **to** $|\mathcal{R}|-1$ **do**
  if r_i *is not switchable with some rule in* s **then**
    Add r_i to s
$post(r_j) = s \setminus \{r_j\}$

Figure 4 shows an optimizer model with 4 patterns and 3 primary modules.

B Complete Version of The Example Policy

In Table 6, we show the complete version of the policy in Table 2.

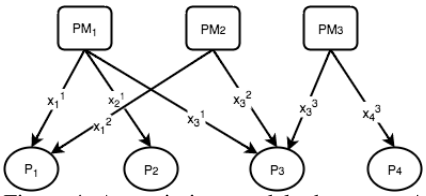

Figure 4: An optimizer model when $m = 4$

No.	sIP	dIP	dPort	decision
1	71.100.64.0/19	*	*	deny
2	71.240.50.0/26	*	*	deny
3	71.206.182.0/24	*	*	deny
4	71.206.190.0/23	*	*	deny
5	71.206.188.0/24	*	*	deny
6	71.206.91.0/24	*	*	deny
7	71.206.88.0/23	*	*	deny
8	71.196.181.0/24	*	*	deny
9	71.196.56.0/22	*	*	deny
10	71.128.0.0/13	*	*	deny
11	71.59.128.0/17	*	*	deny
12	71.59.32.0/19	*	*	deny
13	71.59.11.0/24	*	*	deny
14	71.59.8.0/23	*	*	deny
15	71.59.12.0/22	*	*	deny
16	71.59.16.0/20	*	*	deny
17	71.59.64.0/18	*	*	deny
18	71.121.88.84	207.89.182.61	25	allow
19	71.121.88.84	207.89.182.41	25	allow
20	71.121.92.96	207.89.182.61	25	allow
21	71.121.92.96	207.89.182.41	25	allow
22	*	*	25	deny
23	*	*	137	deny
24	*	*	445	deny
25	*	*	135	deny
26	*	*	138	deny
27	71.14.116.1	71.121.90.184	1953-1954	allow
28	71.14.116.1	71.121.90.191	1953-1954	allow
29	71.14.116.1	207.89.182.37	1953-1954	allow
30	71.14.116.1	207.89.176.14	1953-1954	allow
31	71.14.116.1	207.89.182.57	1953-1954	allow
32	71.14.116.1	207.89.182.61	1953-1954	allow
33	71.14.116.1	207.89.182.41	1953-1954	allow
34	71.14.116.1	207.89.182.27	1953-1954	allow
35	71.14.116.1	207.89.182.26	1953-1954	allow
36	71.14.116.1	207.89.171.57	1953-1954	allow
37	71.14.116.1	207.89.182.251	1953-1954	allow
38	71.14.116.1	207.89.182.250	1953-1954	allow
39	71.14.116.1	207.89.170.190	1953-1954	allow
40	71.14.116.1	207.89.174.60	1953-1954	allow
41	71.14.116.1	207.89.176.60	1953-1954	allow
42	71.14.116.1	207.89.179.185	1953-1954	allow
43	71.14.116.1	207.89.182.107	1953-1954	allow
44	71.14.116.1	207.89.182.179	1953-1954	allow
45	71.14.116.1	207.89.182.198	1953-1954	allow
46	71.14.116.1	207.89.182.50	1953-1954	allow
47	71.14.116.1	207.89.182.143	1953-1954	allow
48	71.14.116.1	207.89.182.142	1953-1954	allow
49	71.14.116.1	207.89.182.17	1953-1954	allow
50	71.14.116.1	71.121.90.128/26	1953-1954	allow
51	71.87.147.117	71.121.90.184	1950-1951	allow
52	71.87.147.117	71.121.90.191	1950-1951	allow
53	71.87.147.117	207.89.182.37	1950-1951	allow
54	71.87.147.117	207.89.176.14	1950-1951	allow
55	71.87.147.117	207.89.182.57	1950-1951	allow
56	71.87.147.117	207.89.182.61	1950-1951	allow
57	71.87.147.117	207.89.182.41	1950-1951	allow
58	71.87.147.117	207.89.182.27	1950-1951	allow
59	71.87.147.117	207.89.182.26	1950-1951	allow
60	71.87.147.117	207.89.171.57	1950-1951	allow
61	71.87.147.117	207.89.182.251	1950-1951	allow
62	71.87.147.117	207.89.182.250	1950-1951	allow
63	71.87.147.117	207.89.170.190	1950-1951	allow
64	71.87.147.117	207.89.174.60	1950-1951	allow
65	71.87.147.117	207.89.179.185	1950-1951	allow
66	71.87.147.117	207.89.182.248	1950-1951	allow
67	71.87.147.117	207.89.182.107	1950-1951	allow
68	71.87.147.117	207.89.182.179	1950-1951	allow
69	71.87.147.117	207.89.182.198	1950-1951	allow
70	71.87.147.117	207.89.182.50	1950-1951	allow

No.	sIP	dIP	dPort	decision
71	71.87.147.117	207.89.182.143	1950-1951	allow
72	71.87.147.117	207.89.182.142	1950-1951	allow
73	71.87.147.117	207.89.182.17	1950-1951	allow
74	71.87.147.117	71.121.90.128/26	1950-1951	allow
75	71.87.147.117	71.121.90.184	1960	allow
76	71.87.147.117	71.121.90.191	1960	allow
77	71.87.147.117	207.89.182.37	1960	allow
78	71.87.147.117	207.89.176.14	1960	allow
79	71.87.147.117	207.89.182.57	1960	allow
80	71.87.147.117	207.89.182.61	1960	allow
81	71.87.147.117	207.89.182.41	1960	allow
82	71.87.147.117	207.89.182.27	1960	allow
83	71.87.147.117	207.89.182.26	1960	allow
84	71.87.147.117	207.89.171.57	1960	allow
85	71.87.147.117	207.89.182.251	1960	allow
86	71.87.147.117	207.89.182.250	1960	allow
87	71.87.147.117	207.89.170.190	1960	allow
88	71.87.147.117	207.89.174.60	1960	allow
89	71.87.147.117	207.89.179.185	1960	allow
90	71.87.147.117	207.89.182.248	1960	allow
91	71.87.147.117	207.89.182.107	1960	allow
92	71.87.147.117	207.89.182.179	1960	allow
93	71.87.147.117	207.89.182.198	1960	allow
94	71.87.147.117	207.89.182.50	1960	allow
95	71.87.147.117	207.89.182.143	1960	allow
96	71.87.147.117	207.89.182.142	1960	allow
97	71.87.147.117	207.89.182.17	1960	allow
98	71.87.147.117	71.121.90.128/26	1960	allow
99	71.67.95.202	71.121.90.184	1960	allow
100	71.67.95.202	71.121.90.191	1960	allow
101	71.67.95.202	207.89.182.37	1960	allow
102	71.67.95.202	207.89.176.14	1960	allow
103	71.67.95.202	207.89.182.57	1960	allow
104	71.67.95.202	207.89.182.61	1960	allow
105	71.67.95.202	207.89.182.41	1960	allow
106	71.67.95.202	207.89.182.27	1960	allow
107	71.67.95.202	207.89.182.26	1960	allow
108	71.67.95.202	207.89.171.57	1960	allow
109	71.67.95.202	207.89.182.251	1960	allow
110	71.67.95.202	207.89.182.250	1960	allow
111	71.67.95.202	207.89.170.190	1960	allow
112	71.67.95.202	207.89.174.60	1960	allow
113	71.67.95.202	207.89.179.185	1960	allow
114	71.67.95.202	207.89.182.248	1960	allow
115	71.67.95.202	207.89.182.107	1960	allow
116	71.67.95.202	207.89.182.179	1960	allow
117	71.67.95.202	207.89.182.198	1960	allow
118	71.67.95.202	207.89.182.50	1960	allow
119	71.67.95.202	207.89.182.143	1960	allow
120	71.67.95.202	207.89.182.142	1960	allow
121	71.67.95.202	207.89.182.17	1960	allow
122	71.67.95.202	71.121.90.128/26	1960	allow
123	*	71.121.90.184	1953-1954	deny
124	*	71.121.90.191	1953-1954	deny
125	*	207.89.182.37	1953-1954	deny
126	*	207.89.176.14	1953-1954	deny
127	*	207.89.182.57	1953-1954	deny
128	*	207.89.182.61	1953-1954	deny
129	*	207.89.182.41	1953-1954	deny
130	*	207.89.182.27	1953-1954	deny
131	*	207.89.182.26	1953-1954	deny
132	*	207.89.171.57	1953-1954	deny
133	*	207.89.182.251	1953-1954	deny
134	*	207.89.182.250	1953-1954	deny
135	*	207.89.170.190	1953-1954	deny
136	*	207.89.174.60	1953-1954	deny
137	*	207.89.179.185	1953-1954	deny
138	*	207.89.182.248	1953-1954	deny
139	*	207.89.182.107	1953-1954	deny
140	*	207.89.182.179	1953-1954	deny
141	*	207.89.182.198	1953-1954	deny
142	*	207.89.182.50	1953-1954	deny
143	*	207.89.182.143	1953-1954	deny
144	*	207.89.182.142	1953-1954	deny
145	*	207.89.182.17	1953-1954	deny
146	*	71.121.90.128/26	1953-1954	deny
147	*	71.121.90.184	1950	deny
148	*	71.121.90.191	1950	deny
149	*	207.89.182.37	1950	deny
150	*	207.89.176.14	1950	deny
151	*	207.89.182.57	1950	deny
152	*	207.89.182.61	1950	deny
153	*	207.89.182.41	1950	deny
154	*	207.89.182.27	1950	deny
155	*	207.89.182.26	1950	deny
156	*	207.89.171.57	1950	deny
157	*	207.89.182.251	1950	deny
158	*	207.89.182.250	1950	deny
159	*	207.89.170.190	1950	deny
160	*	207.89.174.60	1950	deny
161	*	207.89.179.185	1950	deny
162	*	207.89.182.248	1950	deny
163	*	207.89.182.107	1950	deny
164	*	207.89.182.179	1950	deny
165	*	207.89.182.198	1950	deny
166	*	207.89.182.50	1950	deny
167	*	207.89.182.143	1950	deny
168	*	207.89.182.142	1950	deny
169	*	207.89.182.17	1950	deny
170	*	71.121.90.128/26	1950	deny
171	*	71.121.90.184	1960	deny
172	*	71.121.90.191	1960	deny
173	*	207.89.182.37	1960	deny
174	*	207.89.176.14	1960	deny
175	*	207.89.182.57	1960	deny
176	*	207.89.182.61	1960	deny
177	*	207.89.182.41	1960	deny
178	*	207.89.182.27	1960	deny
179	*	207.89.182.26	1960	deny
180	*	207.89.171.57	1960	deny
181	*	207.89.182.251	1960	deny
182	*	207.89.182.250	1960	deny
183	*	207.89.170.190	1960	deny
184	*	207.89.174.60	1960	deny
185	*	207.89.179.185	1960	deny
186	*	207.89.182.107	1960	deny
187	*	207.89.182.179	1960	deny
188	*	207.89.182.198	1960	deny
189	*	207.89.182.50	1960	deny
190	*	207.89.182.143	1960	deny
191	*	207.89.182.142	1960	deny
192	*	207.89.182.17	1960	deny
193	*	71.121.90.128/26	1960	deny
194	71.0.0.0/8	71.121.90.154	22	allow
195	71.0.0.0/8	71.121.90.154	80	allow
196	71.0.0.0/8	71.121.90.154	443	allow
197	71.0.0.0/8	71.121.90.154	5800-5809	allow
198	71.0.0.0/8	71.121.90.154	5900-5909	allow
199	71.0.0.0/8	71.121.90.154	3690	allow
200	*	71.121.90.154	*	deny
201	71.0.0.0/8	*	*	allow
202	71.67.94.12	207.89.182.27	55555	allow
203	71.121.92.53	207.89.182.179	52311	allow
204	207.89.182.142	207.89.182.57	179	allow
205	207.89.182.143	207.89.182.57	179	allow
206	71.0.0.0/8	*	80	allow
207	71.121.88.50	207.89.182.17	52311	allow
208	71.121.59.54	207.89.182.17	52311	allow
209	*	*	*	deny

Table 6: The complete version of the policy in Table 2