Practical Declarative Network Management

Timothy L. Hinrichs
University of Chicago
Computer Science
Chicago, IL USA
thinrich@cs.uchicago.edu

Natasha S. Gude
Stanford University
Computer Science
Stanford, CA USA
ngude@cs.stanford.edu

Martin Casado
Stanford University
Computer Science
Stanford, CA USA
casado@cs.stanford.edu

John C. Mitchell
Stanford University
Computer Science
Stanford, CA USA
mitchell@cs.stanford.edu

Scott Shenker
U.C. Berkeley and ICSI
Electrical Engineering and
Computer Science
Berkeley, CA USA
shenker@icsi.berkeley.edu

ABSTRACT

We present Flow-based Management Language (FML), a declarative policy language for managing the configuration of enterprise networks. FML was designed to replace the many disparate configuration mechanisms traditionally used to enforce policies within the enterprise. These include ACLs, VLANs, NATs, policy-routing, and proprietary admission control systems. FML balances the desires to express policies naturally and enforce policies efficiently. We have implemented FML and have used it to manage multiple operational enterprise networks for over a year.

Categories and Subject Descriptors

D.4.6 [**Software**]: Security and Privacy Protection; D.1.6 [**Programming Techniques**]: Logic Programming

General Terms

Design, Languages

Keywords

Network, Policy, Security, Performance

1. INTRODUCTION

In recent years, high-level declarative management languages have gained traction in the commercial world. Examples include XACML [16] for declaring access-controls over middleware and webservices, and P3P [8, 9] for declaring privacy policies in web environments. In addition, many e-mail readers use declarative languages for message filtering.

WREN'09, August 21, 2009, Barcelona, Spain.

These and other successes provide evidence for the utility of declarative management techniques. However, the adoption of these languages has been almost entirely at the application layer and above. In contrast, enterprise networks continue to be managed through a number of disparate low-level mechanisms, including the use of VLANs and subnetting for isolation, ACLs for access control, NAT for client protection, and policy routing for source-based policies and the integration of middleboxes. As has been frequently lamented in the literature [7, 19, 14], these traditional approaches for network configuration result in networks whose connectivity is dictated by thousands of lines of brittle, low-level configuration code that grows stale as the network evolves. Thus, enterprise networks provide an ideal example of where declarative management techniques could provide substantial benefits.

In this paper, we present Flow-based Management Language (FML), a high-level declarative language for expressing network-wide policies about a variety of different management tasks within a single, cohesive framework. While there have been numerous authorization languages proposed in the literature (*e.g.*, [1, 4, 6, 12, 15, 18] among many others), we believe FML has a unique position in the academic design space as it was purpose-built to replace existing network configuration practices, and has been extensively tested in practice. More specifically, the contributions of this work are as follows:

- With FML we present a simple language that can be used to express many common configurations used in networks today. We demonstrate its expressibility with a series of example applications and by presenting our experiences running FML in multiple operational networks over the last year.
- FML was designed to admit efficient implementation, suitable for large enterprise networks. In particular, its implementation is polynomial time in general, and the fragment discussed in this paper requires linear time, thus supporting network-speed decisions. We present performance numbers from our operational implementation that demonstrate its scaling properties.

In what follows, we describe FML and provide some of

the logical foundations of the language. We then describe how FML can be used to express common network configurations. In the subsequent sections, we describe our implementation of FML, and provide an analysis of its performance as well as our experiences deploying it within two operational networks. Finally we present related work and conclude.

2. FML

2.1 Background

The granularity on which FML operates is a unidirectional network flow (hereafter when referring to flows, we refer specifically to unidirectional flows). This means that a resulting policy decision applies equally to all packets within the same flow, and a policy may be constructed that treats each flow differently (and thus the policy must be consulted at least once per flow).

We have designed and built FML as the underlying policy language for NOX [10], a network-wide control plane that enforces policies on every flow in the network. Similar to its predecessor, Ethane [7], NOX checks the first packet of every flow against the network policy before admitting the flow onto the network. The result of the check may deny or otherwise dictate how the flow is to be handled by the network.

In addition to performing per-flow policy checks, NOX authenticates all network principals including switches, users, and hosts. This is done via standard mechanisms such as 802.1x. NOX then maintains name to address bindings of all authenticated principals, and thus given a flow, it can map the flow to its sending and receiving host and user names.

For concreteness, the descriptions of FML in this paper are derived from our specific implementation environment within NOX. However, we believe the principles generalize to any flow-based network architecture. We assume the network policy engine can derive the associated high-level names for all flows on the network. As shown in Table 1, these include the sending and receiving host name, the sending and receiving user, the sending and receiving access points (physical ports), and the protocol. In addition, we assume that the policy engine is invoked for each flow on the network and can determine whether the flow is a request or a response.

Property	Description
u_s and u_t	Source and Target Users
h_s and h_t	Source and Target Hosts
a_s and a_t	Source and Target Access Points
prot	Protocol
request	Whether or not flow is a request.

Table 1: The properties characterizing a unidirectional flow.

2.2 Overview

FML is a language for specifying policies about flows. FML is based on nonrecursive DATALOG with negation. A FML policy is a set of statements, each representing a simple if-then relationship. The statements dictate that a specified constraint should be applied to the matching flows.

For example, the following three statements say that *todd* and *michelle* are superusers, and a superuser has no communication restrictions.

$$\begin{aligned}&allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow \\ &\quad superuser(U_s) \\ &superuser(todd) \\ &superuser(michelle)\end{aligned}$$

The arguments to *allow* are all variables (denoted by symbols starting with a capital letter) and correspond to the eight fields of a flow. The remaining two rules make simple declarative statements. The constraints (such as *allow*) are named by keywords. Each application of FML can define its own set of keywords.

Throughout this section, we use four keywords from the access control application discussed in Section 3: *allow*, *deny*, *waypoint*, and *avoid*. Intuitively, *deny* blocks a flow; *waypoint* requires a flow to pass through a particular node in the network; and *avoid* forbids a flow from passing through a particular node in the network.

A basic FML policy consists of statements like those seen above, and while such a language can easily represent a wide variety of information, sometimes not all the information pertinent to a policy can or should be represented within a single file. For example, user names and associated groups are often maintained in an external authentication store (such as LDAP or AD), and maintaining a copy within a policy file may be impractical. Consequently, FML policies can include external references to, for example, SQL queries over local or remote databases, hash tables, or arbitrary procedural code.

One of the benefits of FML is that it natively supports distributed authorship, making it easy for disparate parties to collaborate on policy decisions.

As with any network configuration language, FML must assume the possibility of distributed authorship within a single policy domain. This can lead to policies with conflicts (simultaneously allowing and denying a uniflow); consequently, FML includes two conflict resolution mechanisms. One mechanism is under the control of application developers and resolves conflicts at the level of keywords, while the other (a FML cascade) is under the control of policy writers and is built into the language itself. Below we formalize the design of our language. We leave a more detailed treatment to [13].

2.3 Formal definition

The formal syntax for FML rules and FML policies is given below. We use standard terminology. A predicate is a symbol associated with a fixed number of arguments: its arity. A term is a variable (starting with an upper-case letter) or an object constant (starting with a lower-case letter). An atom is a predicate of arity n applied to n terms. A literal is an atom or its negation (indicated by $\neg$). An expression is ground if it contains no variables.

DEFINITION 1 (FML RULE). *A FML rule in the context of external references G takes the following form.*

$$h \Leftarrow [\neg]b_1 \wedge \cdots \wedge [\neg]b_n$$

- *h, the head, and every b_i, collectively called the body, are atoms.*
- *Every variable in the body must appear in the head.*
- *When a keyword with more than eight arguments appears in the head, the extra arguments are object constants.*

- *h does not contain any predicate $g \in G$.*

It is worth noting that the second condition differs from the traditional safety[1] restriction in DATALOG and significantly simplifies FML's implementation. See Section 4 for more information. In addition, it guarantees polynomial-time evaluation. We include the proof in Appendix A.

A FML policy is simply a collection of FML rules that are nonrecursive: if predicate A's definition depends on B, then B's definition does not depend on A.

DEFINITION 2 (FML POLICY). *A FML policy is a set of FML rules Δ that is nonrecursive. (See, for example, [17] for a formalization.)*

The formal semantics of FML is straightforward and can be defined using the usual stratified semantics [17] of logic programming or database theory. For the sake of brevity we omit the definitions. It suffices to say that the obvious if-then intuition for each rule is exactly right, with the only complication arising in the context of *sets* of statements in which a single flow can match multiple rules. In an FML policy, the order of statements is irrelevant—reorder the statements in any policy, and the meaning of the policy is entirely unchanged.

Order irrelevance in a policy language offers two benefits over languages in which statement order matters, *e.g.*, firewall configuration languages [20]. First, combining a set of independently authored policies is straightforward: collect all statements made in the policies. By contrast, it is not immediately clear how to automatically combine policies in which statement order matters, thus requiring an algorithm that will likely be unintuitive to the user. Second, large ordered policies can be difficult to understand because a rule only applies to a flow if none of the previous rules apply. Conversely, every statement in an FML policy applies to every flow; finding a single statement that *allow*s a flow guarantees that one of the constraints on that flow is *allow*.

The practical consequence of order-irrelevance is that a policy can impose multiple constraints on the same flow. For example, an access control policy can force a flow to pass through two different intermediate waypoints while requiring it to avoid yet another waypoint. While this feature makes expressing certain policies more natural, it also admits policies that are unenforceable. For example, a policy could dictate that a flow should be both allowed and denied. No real system can simultaneously allow and deny an access control request; consequently, FML must support schemes for resolving conflicts. Formally, conflict resolution acts as a layer of semantics that is defined on top of the core (stratified) semantics of FML.

2.4 Conflict Resolution

FML conflict resolution varies depending on the application, a necessity because the keywords differ from application to application. In the access control example, the conflict resolution scheme prioritizes the keywords: *deny* takes precedence over everything; *waypoint* and *avoid* are of equal priority and take precedence over *allow*. *allow* has the lowest priority. The resolution mechanism throws away all constraints except those with the highest priority. For example, if a flow were allowed and denied, the resolution mechanism would throw away the *allow* constraint.

[1] All variables must occur in a positive literal in the body.

One benefit of permitting conflicts is that administrators can leverage the resolution scheme to write more concise policies. For example, in the access control setting, it is easy to write an open authorization policy—one that allows everything not explicitly denied.

$$\begin{aligned} &allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \\ &deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow \\ &\quad blacklist(U_s) \end{aligned} \tag{1}$$

These two rules deny all communication initiated by a blacklisted user and allow everything else. To further restrict access, an administrator need only include additional *deny* rules.

While open policies are easy to represent, closed policies (those where everything not explicitly allowed is denied) are problematic. Adding *allow* statements does not affect a policy where every flow is denied because *deny* takes precedence over *allow*. Closed policies thus present a scenario in which ordered semantics would make declaration easier. Consequently, FML supports an ordering mechanism. Named for the cascading style sheets used on the web, an FML cascade is a prioritized series of FML policies, written $P_1 < \cdots < P_n$. Intuitively, any policy in the ordering overrides all of the policies less than it. A conflict between two policies is resolved to what the higher priority policy dictates.

For example, to define a closed authorization policy, one could construct a cascade with two policies: $P_1 < P_2$. P_2 would describe all of the flows that should be allowed, and P_1 would contain a single rule:

$$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req).$$

DEFINITION 3 (FML CASCADE). *A FML cascade consists of a finite set of FML policies $\{P_1, \ldots, P_n\}$ and a total ordering $<$ over those policies. We denote a cascade with $P_1 < \cdots < P_n$.*

In a cascade $P_1 < \cdots < P_n$, the highest ranked policy that says anything about a particular flow is the only policy that says anything about that flow. In other words, policy P_i determines the constraints on a flow if P_i constrains the flow in any way and there is no other P_j that constrains that flow where $j > i$. Formal definitions for this intuition can be found in [13].

While conflict resolution could arguably serve to be as confusing as ordered policy semantics, it provides the user with a straightforward means to define how differing constraints should be merged. Forced order significance precludes this flexibility. Furthermore, any ordered set of FML rules, declarable here using cascades, can be equivalently expressed as an unordered set of FML rules in linear time, eliminating the need for conflict resolution in environments better-suited to ordered semantics. Appendix B details algorithms for eliminating both cascades and keyword conflicts from a policy.

3. EXAMPLES

To demonstrate how FML is used in practice, in the following section we apply it to several common network management tasks: access control, quality of service, NAT administration, and admission control. Each FML application will introduce a set of keywords, describe their meanings, provide examples, discuss conflict resolution, and explain policy enforcement issues at a high-level.

To simplify the presentation of the example policy statements, rather than explicitly including the usual variables for the eight flow fields as shown below,

$$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow \ldots$$

we will denote those eight variables with a single vector variable $\overline{Flow}$, and write rules as $allow(\overline{Flow}) \Leftarrow \ldots$.

These applications assume that the network architecture is such that the policy engine can intercept every flow on the network, and can dictate its path and properties (such as QoS parameters). FML could equally well be used to manage each switch individually (assuming the switch is flow-based), in which case the application would be limited to switch-local rather than network-wide functions.

3.1 Access Control

As briefly discussed in Section 2, in the access control application of FML, administrators dictate whether flows are allowed on the network, and if so, whether there are any constraints on the routes. The keywords are described in Table 2.

Keyword	Description
$allow_0$	Allow flow.
$deny_0$	Deny flow.
$waypoint_1$	Route flow through network node.
$avoid_1$	Route flow to avoid network node.
$ratelimit_1$	Limit flow's maximum Mb/s.

Table 2: Keywords for access control policies. Subscripts denote the number of arguments in addition to the eight flow fields.

For example, to force wireless guest users to send all requests through an IDS, one could write the following rules. Note that below, *guest* is externally defined.

$$\begin{aligned} &waypoint(\overline{Flow}, ids) \Leftarrow \\ &\qquad guest(U_s) \wedge wireless(A_s) \\ &wireless(wap1) \\ &wireless(wap2) \end{aligned}$$

The keywords for this application admit a variety of conflicts: *deny* and any other keyword, *waypoint* and *avoid* if they constrain a flow's route to both include and exclude the same intermediate node, and any keyword together with *allow*.

To resolve conflicts, a most-restrictive scheme is sensible given the desire for security. *deny* is more restrictive than *waypoint*, *avoid*, and *allow*. If a flow is both required to pass through a node via *waypoint* and forbidden to pass through that same node via *avoid*, the flow is denied. Otherwise, the *waypoint* and *avoid* constraints are enforced, and if there are no *deny*, *waypoint*, or *avoid* constraints imposed on a flow, it is allowed.

For a realistic example of an FML access control policy, see Appendix C.

3.2 Quality of Service

For quality of service, administrators dictate how resources should be allocated to different flow classes by specifying the relative importance of certain properties: latency, jitter, and bandwidth. The keywords are described in Table 3.

Keyword	Description
$latency_1$	Set maximum packet delay.
$jitter_1$	Set maximum variance in packet delay.
$band_1$	Set minimum available kbits/s.

Table 3: Keywords for QoS policies. Subscripts denote the number of arguments in addition to the eight flow fields.

For example, VOIP requires low latency and jitter, while bandwidth is less important. Conversely, data backups require high bandwidth, but jitter and latency are less important. The following policy snippet uses protocol number 1233 to signify a data-backup flow.

$$\begin{aligned} &latency(\overline{Flow}, 100) \Leftarrow Prot = voip \\ &jitter(\overline{Flow}, 5) \Leftarrow Prot = voip \\ &band(\overline{Flow}, 3000) \Leftarrow Prot = 1233 \end{aligned}$$

Conflicts arise when a single flow is assigned two or more values for the same property, *e.g.*, a latency of 100 and 200. To resolve such conflicts, we use the most-demanding scheme, which given multiple values for a single property chooses the one that is most difficult to enforce: the lowest latency and jitter and the highest bandwidth.

Because all application statements follow the same logical model, multiple applications can be used to manage the network from the same policy file (provided a sane conflict resolution strategy exists). For example, if both QoS and access controls statements are included in a policy, the lookup could be performed as follows. Each time a flow is initiated, the QoS and access control policies are queried to determine the constraints on the flow. Max-flow algorithms could then be used to choose a route through the network that satisfies the access control constraints and is most likely to achieve the QoS requirements (assuming the policy controller has a reasonably accurate snapshot of the network state).

3.3 NAT

Network address translation (NAT) maps between two pools of IP addresses and is generally used to allow multiple machines within a private IP range to share a single, public address. Implementing such a translation requires altering the IP and port number of each packet leaving and entering the private network. NAT thus differs from applications previously discussed in that each packet in the flow must be modified, therefore requiring the network switches to support this functionality.

NAT also differs from the other applications by requiring the maintenance of state between the request and response flows of a connection. In particular, if a flow's source IP and port number have been overwritten, the return flow must be modified to the original values for the originating host to correctly receive the response. The mapping between the new and old values can be stored in the same location as the policy engine, which will be consulted upon initiation of the return flow. The keywords for NAT are listed in Table 4.

For example, the following statements cause every packet from private IP address 10.0.0.1 connected at the wireless access point *patio* to appear to the rest of the network as though they are from the public address 170.70.70.1 with a modified port. The statements force switch *sw* to perform

Keyword	Description
$srcNAT_2$	Set source IP and port.
$dstNAT_2$	Set destination IP and port.
$unSrcNAT_0$	Set destination to match original source.
$unDstNAT_0$	Set source to match original destination.
$noChange_0$	Do not change flow.

Table 4: Keywords for NAT administration policy. Subscripts denote the number of arguments in addition to the eight flow fields.

the address translation.

$$srcNAT(\overline{Flow}, 170.70.70.1, sw) \Leftarrow \\ A_s = patio \wedge IP_s = 10.0.0.1$$
$$unSrcNAT(\overline{Flow}) \Leftarrow \\ A_d = sw \wedge IP_d = 170.70.70.1$$

The *sw* argument corresponds to the NAT switch for the private network—the switch that partitions the private IP addresses from the public IPs. Since many private subnetworks may exist within a network, the same private IP may be used in different portions of the network, and the *sw* argument serves to differentiate which private network each rule applies to.

This example includes a rule conditioned on the source IP address of a flow, a field that was not included in the definition of a flow (Table 1); however, that definition was intended to be instructive, not exhaustive. Our actual implementation includes several other flow properties not mentioned in Table 1, *e.g.*, source and target IP addresses.

In this application, conflicts arise between *noChange* and any other keyword, and when two changes require modifying the same field, *e.g.*, assigning two different addresses to the same source IP. Our resolution mechanism takes the most conservative action by prioritizing undoing of NAT actions over everything else, and otherwise performing no translation if conflicts exist. In particular, *unSrcNAT* and *unDstNAT* take precedence over all other keywords, while *noChange* does so over *srcNAT* and *dstNAT*. Conflicts among *srcNATs* or *dstNATs* resolve to *noChange*.

3.4 Admission Control

Admission control allows the administrator to specify the authentication requirements for hosts and users to gain access to the network. Admission control can be achieved through FML by using it to define what default connectivity is allowed (for unauthenticated hosts), and which authentication mechanisms are to be used.

For example, we have recently deployed an admission control policy at a large university in which all users with private addresses must authenticate via a captive web portal before being given access to the network. Therefore we have to provide default access from hosts to the web-servers, and from the web-servers to the directory servers.

For example, the following policy cascade sets up default connectivity for authentication, and redirects all unauthenticated users to a captive web portal. Here we have $P_3 > P_2 > P_1$.

Policy $\mathbf{P_3}$
$allow(\overline{Flow}) \Leftarrow Prot = arp$
$allow(\overline{Flow}) \Leftarrow Prot = dhcp$
$allow(\overline{Flow}) \Leftarrow H_t = auth\_server \wedge Prot = http$
$allow(\overline{Flow}) \Leftarrow H_s = auth\_server \wedge Prot = http$
Policy $\mathbf{P_2}$
$httpRedirect(\overline{Flow}, 307, auth\_server) \Leftarrow$
$\quad U_s = unknown \wedge Prot = http$
Policy $\mathbf{P_1}$
$deny(\overline{Flow}) \Leftarrow U_s = unknown$

The only reserved constant in FML, *unknown*, is used for any flow field that has no known value. The policy above allows ARP and DHCP messages, as well as HTTP connections to and from the authentication server. In addition, an HTTP flow with an unknown source user is redirected with the status code 307 to the host *auth_server*. Finally, all other flows are denied.

This example extends the access control application with the keyword *httpRedirect*. For conflict resolution, *deny* overrides everything, and *httpRedirect* overrides everything except *deny*. Multiple *httpRedirect* target hosts causes a multiple target redirect (status code 300), and multiple *httpRedirect* status codes are resolved to a temporary redirect (status code 307).

4. IMPLEMENTATION

In this section we describe our implementation of FML. All language features are supported with the exception that predicates (other than keywords) are restricted to one argument and are strongly typed (apply to hosts, users, or access points). This restriction enables linear run-time evaluation, an improvement over polynomial time for arbitrary FML. Speed critical operations (such as run-time policy checking) are implemented in C++, while compilation and various integration components are written in Python. Our implementation is roughly 10,000 lines.

We have implemented (and deployed) all of the applications mentioned in the previous section except for QoS (including access controls, NAT, and admission controls). In what follows, we briefly describe our implementation and its environment, our real-world experiences, and provide some performance results.

4.1 Implementation Environment

As mentioned previously, we have implemented FML within NOX. NOX is a general control platform on top of which network-wide control applications can be built. NOX provides applications with a global view of the network topology (links, nodes, and hosts) and the ability to be notified on significant network events (*e.g.*, a new flow has entered the network, or a host has joined the network). In addition, through NOX, applications can control the behaviour of the network by setting up flows in the network switches (as the result of a policy decision, for example), and sending packets.

FML operates within NOX by intercepting all new flows on the network and checking them against the current policy. If a flow is permitted, FML uses the NOX routing library to calculate a compliant route (which may include waypoints) and then adds the flow to the flow-table of each switch along

the path.

NOX builds the policy namespace, containing name to address bindings, by handling the authentication of network principals. When a principal authenticates, its name is bound to the source access point, source MAC address, and source IP address (if it exists), used during the authentication exchange. The names and user credentials are generally retrieved from a remote authentication store (our implementation supports both LDAP and AD). On successful authentication, the names are cached by the policy engine and indexed by address for quick retrieval on each new flow.

4.2 Policy Lookup and Evaluation

Since policy consultation occurs on every flow, the evaluation engine has been optimized for performance and is comprised of two components: a decision tree intended to minimize the number of rules evaluated per flow, and an algorithm for evaluating a given set of rules. Given a flow's eight properties, the decision tree identifies the subset of the original policy that needs to be evaluated, and the evaluation algorithm computes the set of constraints imposed by the selected rules on the given flow. Since the evaluation algorithm is standard, we focus on the decision tree indexing structure.

The tree partitions the rules by placing all rules that constrain the eight flow fields the same way into the same partition. The result is a compact representation of the rule set in an eight-dimensional space. Negative literals are ignored by the indexer and evaluated at runtime.

Each node in the decision tree has one child for each possible matching value for the dimension that node represents, *e.g.*, a node representing U_s has one child for each value assigned to U_s in the subtree's policy rules. In addition, because some of a subtree's rules may not constrain the dimension a node represents, *e.g.*, $Prot$ in the rule below, each node includes an ANY child for such rules to be placed in. Each node in the decision tree is implemented using a hash table with chaining to ensure that each of its children can be found in near constant time. The decision as to which of the eight attributes to branch on at any point in the tree is made by finding the dimension that most widely segments a subtree's rule set. In particular, we select the dimension that minimizes the average number of rules at each child node plus the number of ANY rules in the subtree.

For example, the rule

$$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow \\ U_s = alice \wedge U_t = bob$$

mentions the source user U_s and the target user U_t. This rule would belong to the node in the decision tree where U_s is *alice* and U_t is *bob*. Figure 1 shows one possible tree (branch), where the rule above is located in the node marked with an asterisk. (Our decision tree also handles rules that require a uniflow field to be true of a user-defined predicate, but our simple example conveys the basic premise. See [13] for more details.)

Notice that the size of the tree is dependent only on the size of the policy, not on the number of users or hosts in the network. For example, the source user node only has as many children as there are constants c where the constraint $U_s = c$ occurs in some rule body. Thus FML evaluation is dependent only on the size of the policy (assuming appropriate indexing for external references).

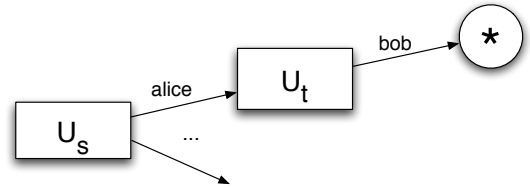

Figure 1: Example decision tree

The total cost of FML evaluation is the cost of finding the pertinent rules in the tree index plus the cost of evaluating those rules. Rule evaluation is performed in the usual way. Conflict resolution is handled by the system directly: actions of all matching rules are collected, and the conflict resolution scheme is applied. A FML cascade is implemented by evaluating rules in priority order; once a match is found, only the remaining rules of the same priority are evaluated.

4.3 Deployments

FML is currently being used to manage two operational networks. The first is a small business with roughly 70 hosts connected through 4 switches. The second deployment is at a large medical university where FML is being used to enforce policy over a roughly 200 host network. We describe each of these deployments in more detail below.

Within the small business network, FML is being used to enforce both admission and access control policies. The admission policy requires users on workstations and laptops to authenticate via a captive web-portal (against an LDAP authentication store) before being permitted access to the network. The access policy is roughly as follows. Application servers are allowed unrestricted access, while test servers are unable to connect to the Internet. Servers and printers should only allow inbound connections (except for outgoing SSH sessions on server), providing protection similar to a DMZ. Laptops and mobile devices (such as mobile phones supporting Wi-Fi) are not allowed inbound connections (similar to NAT). We also have a few rules that allow monitoring and diagnostic traffic between an administrative host and all switches. The policy file for this network is just over 40 lines.

Like the small business network, the deployment at the medical university enforces basic admission and access policies. However, it also makes use of NAT and waypointing. The policy dictates that all laptops must be NAT'd (protecting them from inbound traffic). It also requires that all HTTP traffic traverse a proxy before leaving the campus network.

Flow setup latencies (involving two permission checks, route calculations, and flow-entry setups) are generally under 20ms. In our deployments, there is not enough traffic to stress our implementation (we generally see less than 100 new flow setups/s even in the larger network). However in benchmarks using generated traffic, our implementation running with our internal policy file supports permission checks on over 30,000 flows/s, over three times that of any network we have measured. We present more performance tests of the system under load below.

4.4 Performance

We tested the performance and memory overhead of our

implementation with policy files of increasing size (Table 5). All policies (except those with 0 rules) forced a maximum depth tree once constructed. For each incoming flow, on average $log_2(\#rules)$ matched and had to be evaluated by the system. Table 6 shows the same test using policies in which 10% of the rules contain ANY fields (the number of fields containing ANYs is evenly distributed between 1 and the maximum number of fields). In this case, the increased number of rules matching a given flow due to the number of ANYs causes a performance degradation in larger policy files.

The goal of this analysis is not to provide an exhaustive investigation of the performance of our implementation, but rather to gain some insight into its handling of load under various rule sets. As shown in [7], even large enterprise networks of tens of thousands of hosts generally have less than 10,000 flow requests per second, far below the performance capabilities of our implementation for the rule sets tested.

	flows/s	Mbytes	avg. matches
0 rules	103,699	0	0
100 rules	78,808	0	4
1,000 rules	75,414	2	7
10,000 rules	67,843	56	9

Table 5: Performance and memory overhead of our FSL implementation over policies with increasing rule count. The rightmost column contains the average number of matching rules per flow.

	flows/s	Mbytes	avg. matches
0 rules	103,699	0	0
100 rules	100,942	1	2
1,000 rules	76,336	2	10
10,000 rules	46,956	38	52

Table 6: Performance and memory overhead of our FSL implementation over policies declared over 1000 principals in which 10% of the rules contain ANYs. The rightmost column contains the average number of matching rules per flow.

5. RELATED WORK

Declarative languages have been proposed for numerous applications by the security and artificial intelligence communities. Here we summarize which language features FML includes; other features were left out most often because of the severe performance requirement (10^{-4} seconds per query).

FML is based on a restricted form of DATALOG with negation, *e.g.*, [15], allows conflicts, *e.g.*, [18], admits automated detection of conflicts, *e.g.*, [3], and adheres to a fixed conflict resolution scheme, *e.g.*, [18, 15]. As is standard in database applications, FML policies reference external sources, *e.g.*, [5]; it also employs a sequential semantics operator (called "overrides" in [5] and "exceptions" in [4]) via FML Cascades. For a more comprehensive treatment of related work, see [13].

The highest-level design decision was to base FML on DATALOG instead of, for example, first-order logic [12], modal logic [1], or linear logic [6]. Besides efficient implementation, two features of DATALOG stand out. First, its similarity to traditional programming languages makes it accessible to our target audience: network operators (who may be logic novices). Second, its semantics guarantees that true disjunction is inexpressible. If true disjunction were expressible, a FML policy could deny one of two flows without specifying which. Such expressiveness has questionable utility and makes enforcement especially problematic.

There have been a number of approaches proposed for making firewall configuration more manageable. These include the use of entity relationship modeling [2] and high-level language design (*e.g.*, [11]). Our work has similar objectives, yet we broaden the scope of the configuration beyond filtering policies for firewalls to include other common network configurations such as QoS, route control, NAT, and broadcast isolation.

6. CONCLUSION AND FUTURE WORK

In this paper we described FML a new langauge for declaring network connectivity policies as a set of ordered constraints. FML allows succinct, structured, high-level specification of various management tasks, freeing network administrators from the drudgery of configuring myriad router ACLs, firewalls, NATs and VLANs to achieve comprehensive and conceptually straightforward network usage policies.

FML is a declarative policy language based on nonrecursive DATALOG with structured negation. The declarative nature of FML enables administrators to focus on policy decisions instead of implementation details. In addition, logic-based algorithms that detect and resolve conflicts in predictable and reliable ways may be incorporated into policy development environments. FML also supports prioritized policy combination, which is a natural way to express many policies and enables incremental policy updates.

FML is not merely a paper design. We have used it in several operational networks, and have subjected it to additional tests using much more demanding artificially generated loads. This operational experience demonstrates that our implementation has modest memory requirements and can scale to very large networks while supporting policy files of tens of thousands of rules.

In the future we plan to investigate how FML policies can automatically be decomposed to support distributed enforcement, thereby lessening the impact of a centralized controller. Such a scheme would enable centralized, authoritative policy authoring and analysis yet retain the robustness of today's networks. While our current infrastructure suffices to address small and medium networks, automatic decomposition could enlarge the applicability of FML to large, international networks as well.

We also plan to develop a tool suite for analyzing FML policies. For example, FML supports both developer and administrator supplied forms of conflict resolution, both of which can make understanding large FML policies difficult. While the algorithms in Appendix B demonstrate that conflicts can be removed automatically, the resulting policies are far from user-friendly. A suite of sophisticated, user-friendly tools would help operators debug and feel confident about their FML policies.

Finally, FML relies heavily on developer-chosen keywords that encode all the possible constraints that may be imposed on a flow for a given application. To implement a new application, a developer must write custom software that implements each of the keywords, as well as conflict resolution.

It would be preferable if FML allowed administrators to invent new keywords by writing additional policy statements axiomatizing the meaning of those keywords, and the FML implementation would compile those statements to imperative code. This would enable administrators to impose constraints on flows not anticipated by the NOX developers.

7. ACKNOWLEDGEMENTS

We would like to thank Michael Genesereth and Jad Naous for their helpful feedback during the FML design process. We would also like to thank the Nox development team, particularly Dan Wendlandt and Teemu Koponen for significant contributions towards the implementation. Finally, we would like to thank the anonymous reviewers for their comments.

8. REFERENCES

[1] M. Abadi, M. Burrows, and B. Lampson. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4), 1993.

[2] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22:381–420, 2004.

[3] A. Barth, J. C. Mitchell, and J. Rosenstein. Conflict and combination in privacy policy languages. In *Proc. of the ACM Workshop on Privacy in the Electronic Society*, 2004.

[4] E. Bertino, S. Jajodia, and P. Samarati. A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems*, 17(2), 1999.

[5] P. A. Bonatti, S. D. di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *In ACM CCS*, 2000.

[6] K. D. Bowers, L. Bauer, D. Garg, F. Pfenning, and M. K. Reiter. Consumable credentials in logic-based access-control systems. In *Proc. of the NDSS*, 2007.

[7] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. ACM SIGCOMM Conference*, Kyoto, Japan, Aug. 2007.

[8] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle. The platform for privacy preferences 1.0 (P3P1.0) specification, 2002.

[9] S. Egelman, L. F. Cranor, and A. Chowdhury. An analysis of p3p-enabled web sites among top-20 search results. In *ICEC '06: Proceedings of the 8th international conference on Electronic commerce*, New York, NY, USA, 2006.

[10] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards and operating system for networks. In *ACM SIGCOMM Computer Communication Review*, July 2008.

[11] J. D. Guttman. Filtering postures: Local enforcement for global policies. In *In Proceedings, 1997 IEEE Symposium on Security and Privacy*, pages 120–129. IEEE Computer Society Press, 1997.

[12] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *Proc. of the IEEE Computer Security Foundations Symposium*, 2003.

[13] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker. Expressing and enforcing flow-based network security policies. Technical report, University of Chicago, 2008.

[14] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, 2000.

[15] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proc. IEEE Symposium on Security and Privacy*, 1997.

[16] B. Parducci, H. Lockhart, R. Levinson, J. B. Clark, and M. McRae. eXtensible Access Control Markup Language (XACML) specification, 2005.

[17] J. Ullman. *Principles of Database and Knowledge-Base Systems*. 1989.

[18] D. Wijesekera and S. Jajodia. Policy algebras for access control - the predicate case. In *Proc. ACM CCS*, 2001.

[19] G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, and G. Hjalmtysson. Routing design in operational networks: A look from the inside. In *Proc. ACM SIGCOMM '04*, New York, NY, USA, 2004.

[20] L. Yuan and H. Chen. FIREMAN: A toolkit for firewall modeling and analysis. In *Proc. of the IEEE Symposium on Security and Privacy*, 2006.

APPENDIX

A. PROOFS

THEOREM 1 (FML LANGUAGES ARE POLYNOMIAL). *Given a policy written in a FML language and a query of the form $key(t_1, \ldots, t_n)$, evaluating the query takes polynomial time.*

PROOF. FML is a syntactic variant of nonrecursive DATALOG with negation where the variables in the body of every rule must appear in the head. It is simple to see by induction on the number of extension (backward-chaining) operations needed to evaluate the query that the number of arguments to every predicate participating in that evaluation is no greater than n (the number of arguments to *key*). This is important because n is a constant (independent of the policy), and a well-known result of DATALOG guarantees that if all of the predicates take no more than a constant number of arguments, evaluation is polynomial. Thus, evaluation of the query is polynomial because the fragment of the policy used during evaluation obeys the constant-argument assumption.

Here we sketch a proof of the well-known result: if all of the predicates in the rule set take no more than c arguments, then evaluation is polynomial in the size of the rule set and data. Consider any rule.

$$p(\bar{t}) \Leftarrow [\neg]b_1(\overline{t_1}) \wedge \cdots \wedge [\neg]b_m(\overline{t_m})$$

The number of ground instances of this rule is at most $|U|^c$, where U is the universe, because once every variable in the head of the rule is bound, every variable in the body is bound also. Because c is a constant, $|U|^c$ is a polynomial; consequently, grounding a rule set takes polynomial time. To determine whether a ground set of rules entails a ground atom, one can use a variant of the context-free grammar (CFG) marking algorithm for determining whether a given grammar is empty.

If negation does not occur in the rules, the CFG marking algorithm is exactly the right algorithm to use. This algorithm runs in time polynomial in the size of the input, which is polynomial in the size of the original sentences.

If negation does occur, the marking algorithm needs to be altered so that it only marks negative literals once all the positive consequences have been marked. The result is an algorithm that runs the CFG algorithm no more times than the number of the original sentences, which again is a polynomial algorithm. □

B. ALGORITHMS

The algorithms presented in this section demonstrate that both keyword and cascade conflicts can be removed automatically from FML. For the sake of brevity, the algorithms are illustrated using just the keywords *allow* and *deny*; generalizing to an arbitrary set of keywords is straightforward. The first section introduces a special form for FML policies that simplifies the presentation of the subsequent algorithms. In the examples that follow, we use $\overline{Flow}$ to represent the usual sequence of variables:

$$U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req.$$

B.1 Inline Normal Form

For the upcoming algorithms, it is convenient if a FML policy mentions no intermediate predicates. An intermediate predicate is any predicate that is not an external reference, a keyword, or =. We say that a policy where the only predicates are non-intermediaries is in Inline Normal Form (INF).

For example, the following policy uses the intermediate symbols *host* and *private*.

$$\begin{array}{l} allow(\overline{Flow}) \Leftarrow host(H_s) \\ host(X) \Leftarrow private(X) \\ host(X) \Leftarrow server(X) \\ private(X) \Leftarrow desktop(X) \\ private(X) \Leftarrow laptop(X) \end{array}$$

An equivalent policy, written in INF is shown below.

$$\begin{array}{l} allow(\overline{Flow}) \Leftarrow server(H_s) \\ allow(\overline{Flow}) \Leftarrow desktop(H_s) \\ allow(\overline{Flow}) \Leftarrow laptop(H_s) \end{array} \tag{2}$$

Transforming a policy into INF is straightforward. We say that the *definition* for a predicate p is the disjunction of all the rule bodies where p occurs in the head (after canonicalizing the head of each rule). To compute INF, repeat the following procedure until all of the intermediate predicates in rule bodies have been removed: each time an intermediate predicate occurs in the body of a rule, inline the definition for that predicate. Afterwards, every predicate in the body of a rule is an external reference or =.

In the above example, it takes two iterations of inlining before all of the intermediate predicates have been removed. Concentrating on just the *allow* rule, the result of replacing *host* with its definition is

$$allow(\overline{Flow}) \Leftarrow (private(H_s) \vee server(H_s))$$

The result of replacing *private* with its definition is:

$$allow(\overline{Flow}) \Leftarrow ((desktop(H_s) \vee laptop(H_s)) \vee server(H_s))$$

This procedure produces rules with arbitrary boolean formulae in the body. Such rules are sometimes convenient, even though they do not change the expressiveness of the language. A set of rules where the bodies can include arbitrary boolean formulae and the only predicates mentioned are the keywords, external group symbols, and =, is in Extended Inline Normal Form (EINF).

To convert EINF rules into INF requires converting to conjunctive normal form (CNF) (without introducing intermediate predicates). The standard recursive-descent clausal form conversion algorithm does just this.

B.2 Flattening a FML Cascade

FML was designed so that administrators could independently author pieces of a security policy for a network and then automatically combine those pieces. FML cascades make expressing and editing certain policies easier; consequently, we expect that sometimes administrators will express their policies as cascades. Thus, despite the order-relevance of a cascade, we need algorithms that automatically combine them while preserving their semantics. Our approach transforms each FML cascade into an equivalent FML policy and then combines the resulting policies in the usual way.

To illustrate how a FML cascade can be flattened into a single FML policy, consider a simple example with two policies, each with a single rule. Policy P_1 says that wireless users cannot connect to the Human Resources (HR) server, and policy P_2, which overrides P_1, says that the CEO can do as he pleases. The ordering is $P_1 < P_2$.

$$\begin{array}{ll} P_1: & deny(\overline{Flow}) \Leftarrow \underline{wireless}(A_s) \wedge H_t = hrserver \\ P_2: & allow(\overline{Flow}) \Leftarrow U_s = ceo \end{array}$$

This cascade says that every wireless guest user except for the CEO is denied from accesssing the HR server, and the CEO can access everything. To write this cascade as a single policy, it is sufficient to negate the conditions under which the rule in P_2 applies and add the result to the conditions in the P_1 rule.

$$\begin{array}{l} deny(\overline{Flow}) \Leftarrow wireless(A_s) \wedge H_t = hrserver \wedge U_s \neq ceo \\ allow(\overline{Flow}) \Leftarrow U_s = ceo \end{array}$$

In general, suppose the two policies are written in Extended Inline Normal Form with exactly one rule for *allow* and one rule for *deny*. In policy P_1, there are exactly two rules.

$$\begin{array}{l} deny(\overline{Flow}) \Leftarrow \phi_1(\overline{Flow}) \\ allow(\overline{Flow}) \Leftarrow \psi_1(\overline{Flow}) \end{array}$$

Likewise, policy P_2 contains two rules, denoted by subscripting with a two instead of a one. In the cascade $P_1 < P_2$, a rule in P_1 only applies if none of the rules in P_2 apply. The rules in P_2 apply when either $\phi_2(\overline{Flow})$ or $\psi_2(\overline{Flow})$ is true; the rules in P_2 do not apply when neither of those sentences is true. Such a policy can be constructed by adding constraints to each rule in P_1, the result of which is shown below, while including the rules in P_2 as they are.

$$\begin{array}{l} deny(\overline{Flow}) \Leftarrow \phi_1(\overline{Flow}) \wedge \neg\phi_2(\overline{Flow}) \wedge \neg\psi_2(\overline{Flow}) \\ allow(\overline{Flow}) \Leftarrow \psi_1(\overline{Flow}) \wedge \neg\phi_2(\overline{Flow}) \wedge \neg\psi_2(\overline{Flow}) \end{array}$$

Generalizing to n policies is straightforward, and it turns out that by using intermediate predicates, there is an algorithm that flattens a FML policy in linear time.

B.3 Conflict Conditions

One of the features of FML, introduced to facilitate collaborative policy authoring, is the ability to express conflicts using keywords. One administrator may want to deny a class of flows, and another administrator may want to allow that class. Algorithms for detecting conflicts are important because people who contributed conflicting statements can be made aware of their conflicting intentions.

Certain types of conflicts can always be detected at compile-time (static conflicts), but other types of conflicts can only be detected at run-time because they depend on the external references. In general, the best we can hope for are the conditions on the external references (expressed as logical formulae) that characterize the conflicts.

Consider an example. The following rules rely on the external references q and r.

$$allow(\overline{Flow}) \Leftarrow q(U_s)$$
$$deny(\overline{Flow}) \Leftarrow r(U_s)$$

These rules conflict only when there is some user that belongs to both the groups q and r. Since we do not know at compile-time whether there is such a user, the best we can do is state the conditions under which there is a conflict between these two rules: $q(U_s) \wedge r(U_s)$.

To construct the conditions under which a policy is conflicting, suppose the policy has been written in EINF with one rule for *allow* and one rule for *deny*.

$$deny(\overline{Flow}) \Leftarrow \phi(\overline{Flow})$$
$$allow(\overline{Flow}) \Leftarrow \psi(\overline{Flow})$$

The conditions under which a conflicts occurs is given by the following expression.

$$\phi(\overline{Flow}) \wedge \psi(\overline{Flow})$$

Admittedly, such an expression needs to be simplified before being presented to a real person, but conceptually, computing the conditions under which conflicts occur is straightforward.

B.4 Conflict-free Normal Form

The fact that FML policies allow keyword conflicts is a powerful property because it enables the system to understand more fully the users' intentions; however, because a real system can only enforce a conflict-free policy, those conflicts are resolved automatically. The downside to this approach is that a person looking at the current policy may have a hard time understanding why the system constrains a given flow the way it does. Finding a single rule that says a flow is allowed is insufficient for concluding that the system will allow that flow. There may be another rule that says to deny that flow, and the deny rule takes precedence.

The algorithm outlined next transforms a given policy P into a new policy P' such that P' is conflict free and P and P' are equivalent when *deny* overrides *allow*. When examining P', finding a rule that allows a flow is sufficient for determining that policy as a whole allows that flow. We say that a policy without conflicts is in Conflict-free Normal Form (CFNF).

Suppose the policy has been written in Extended Inline Normal Form with exactly one rule for *allow* and for *deny*.

$$deny(\overline{Flow}) \Leftarrow \phi(\overline{Flow})$$
$$allow(\overline{Flow}) \Leftarrow \psi(\overline{Flow})$$

The conflict resolution policy says that *deny* overrides *allow*; thus, the *allow* rule only applies when the *deny* rule body is false. The transformation to CFNF is accomplished by conjoining the negation of the *deny* rule body to the *allow* rule body. The *deny* rule need not be altered.

$$allow(\overline{Flow}) \Leftarrow \psi(\overline{Flow}) \wedge \neg\phi(\overline{Flow})$$

C. EXAMPLE POLICY

The policy cascade shown in Figure 2 is the one used for access control in our internal network. The policies are shown from highest to lowest priority: $P_1 < P_2 < P_3 < P_4$.

Policy P_4
allow ARP and DHCP
$allow(\overline{Flow}) \Leftarrow Prot = arp$
$allow(\overline{Flow}) \Leftarrow Prot = dhcps \wedge H_t = gateway$

allow computers to ssh anywhere
$allow(\overline{Flow}) \Leftarrow Prot = ssh \wedge \underline{computer}(H_s)$

allow internal monitoring flows:
proprietary protocols registered with the system
$allow(\overline{Flow}) \Leftarrow Prot = 1616 \wedge H_s = badwater$
$allow(\overline{Flow}) \Leftarrow Prot = 1717 \wedge H_s = badwater$
$allow(\overline{Flow}) \Leftarrow Prot = 1818 \wedge H_s = badwater$
$allow(\overline{Flow}) \Leftarrow Prot = 1616 \wedge H_t = badwater$
$allow(\overline{Flow}) \Leftarrow Prot = 1717 \wedge H_t = badwater$
$allow(\overline{Flow}) \Leftarrow Prot = 1818 \wedge H_t = badwater$

Policy P_3
dissallow external communications for
testing machines
$deny(\overline{Flow}) \Leftarrow testing(H_s)$
$deny(\overline{Flow}) \Leftarrow testing(H_t)$

servers should be inbound-only
$deny(\overline{Flow}) \Leftarrow Req = true \wedge \underline{server}(H_s)$
$deny(\overline{Flow}) \Leftarrow Req = true \wedge \underline{printer}(H_s)$

laptops and mobile devices
should be outbound-only
$deny(\overline{Flow}) \Leftarrow Req = true \wedge \underline{mobile}(H_t)$
$deny(\overline{Flow}) \Leftarrow Req = true \wedge \underline{laptop}(H_t)$

Policy P_2
allow known devices to communicate as long as
they abide by the previous rules.
$allow(\overline{Flow}) \Leftarrow all(H_s)$

Policy P_1
default deny
$deny(\overline{Flow})$

Figure 2: Cascaded policy for internal network.