Expressing and Enforcing Flow-Based Network Security Policies

Timothy Hinrichs
University of Chicago

Natasha Gude
Stanford University

Martìn Casado
Stanford University

John Mitchell
Stanford University

Scott Shenker
U.C. Berkeley and ICSI

March 17, 2009

Abstract

While traditional network security policies have been enforced by manual configuration of individual network components such as router ACLs, firewalls, NATs and VLANs, emerging enterprise network designs and products support global policies declared over high level abstractions [2, 1, 14, 13]. We further the evolution of simpler and more powerful network security mechanisms by designing, implementing, and testing a flow-based network security policy language and enforcement infrastructure. Our policy language, FSL, expresses basic network access controls, directionality in communication establishment (similar to NAT), network isolation (similar to VLANs), communication paths, and rate limits. FSL supports modular construction, distributed authorship, and efficient implementation. We have implemented FSL as the primary policy language for NOX, a network-wide control platform, and have deployed it within an operational network for over 10 months. We describe how supporting complex policy objectives and meeting the demanding performance requirements of network-wide policy enforcement have influenced the FSL language design and implementation.

1 Introduction

A wide range of modern enterprises rely on their computer networks for day-to-day operations, and rely on their network administrators to configure security components so that the network is not misused by errant individuals or malware. Traditionally, network security policies have been enforced by manual configuration of individual network components such as router ACLs, firewalls, NATs and VLANs. However, emerging enterprise network designs and products increasingly support global policies that can be expressed using high level abstractions [2, 1, 14, 13]. While different network systems may employ different network-level mechanisms, modern designs now allow us to usefully separate policy specification from policy implementation, viewing the implementation mechanism as"compiling" high-level policy statements into the appropriate low-level configurations needed by a network's constituent components. In this setting, a central question that emerges is: what should the policy language for a large-scale high-speed operational network look like?

In this paper, we propose a Flow-based Security Language (FSL) that is designed to provide meaningful, intuitive high-level concepts to network administrators, while at the same time supporting efficient implementation of the resulting policies in modern networks. Our language FSL is based on a restricted form of DATALOG [24, 8, 27, 36, 19, 28, 7], which is widely used in connection with database systems and certain forms of logic programming, enhanced with a carefully-chosen use of negation [31, 8, 24]. As explained further in the body of this paper, the specific design of FSL was guided by three practical issues:

General constraints: Many treatments of access control only consider the binary actions of *allow* and *deny*; however, modern network security requires a more general approach. Operators would like to impose connectivity constraints on the rate, the path (requiring it to either avoid or include certain nodes), and the directionality (such as restricting clients to outbound flows only) of communications.

Distributed and incremental authorship: There are many policy authors in large enterprises, and thus the policy language must cope gracefully with any conflicts between their policies. In addition, operators may need to quickly inject new policies into a system without ensuring the new entries are consistent with all currently-installed policies. Thus, FSL must gracefully and predictably handle two types of policy conflicts: those between administrators, and those within an administrator's own rule set.

Speed: Policy decisions must be enforced on each packet, and on network links this can easily reach millions of enforcement decisions per second.

While there is a large literature on general access policy languages (see, for example, [24, 12, 11, 4]), prior languages are generally geared toward general computing resources or operating system concepts such as user names, host names, groups, services, and so on. As a result, most theoretically-grounded policy languages have never been tested against the gritty realities of operational networks. In particular, the need to process policies at line-speed and the challenges facing operators to declare fully consistent policies, have rarely, if ever, been addressed in previous policy language designs.

FSL is the flow-based network security policy language of an operational system called NOX [20], targeted for widespread deployment in academia and the government. A NOX network involves a set of "dumb" switches that simply implement flow table forwarding, and a centralized controller (replicated as needed for resilience) that guides the switches. When a packet arrives at a switch and there is no corresponding entry in the flow table, the packet is forwarded to the controller for inspection. The controller makes the requisite policy decision, and then inserts the corresponding flow-table entries into the appropriate set of switches. NOX handles the authentication of all users, hosts, and switches using credentials registered by the network operator either directly with NOX or in an external authentication store (such as LDAP or AD). In our current development deployment, we operate a NOX local area network consisting of roughly 50 hosts connected through 4 switches, managed by our FSL implementation running on NOX. The policy used to collect experimental performance results contains 24 rules declared over 64 principals and 11 groups including workstations, servers, printers, and mobile devices. Round trip flow setup latencies (involving two permission checks, route calculations, and flow-entry setups) in this network are generally under 50ms. Once the policy decision has been made for each uniflow, all subsequent packets for that flow are forwarded at line speed (as supported by the switch forwarding hardware). While normal use of this network does not stress our implementation, benchmarks using more demanding generated traffic show that our NOX implementation supports permission checks on over 60,000 flows/s, an order of magnitude higher than any network we have measured.

Following a discussion of the design requirements for the language (Section 2), we introduce FSL (Section 3). We then explain our implementation of FSL (Section 4), discuss our experiences, and analyze its performance. Finally we consider FSL's limitations in the context of other policy languages (Section 5), and conclude (Section 6).

2 Design Requirements

FSL is designed to be practically useful. This requires policies be writable by real operators and enforceable by real networks. Thus, FSL's design constraints come from the requirements of *writing* and *implementing* policies. In this section we first discuss these two requirements in general. Then for specificity we describe the NOX system in which we have implemented FSL, and for generality we define a formal model for our policy problem.

2.1 Writing Policies

Because it is intended for use by real operators, FSL should make it simple to declare and maintain policies. As time goes on, we expect organizations to accumulate a large number of policies. When an operator is faced with a situation requiring a change in policy, it would be unreasonable to require the operator to

modify the entire policy corpus to ensure a newly written (or rewritten) policy has no conflicts with any other policies. Instead, we want to make it possible for operators to incrementally add policy statements as new requirements arise, even if these new statements conflict with older policy statements.

In addition, most large enterprise networks have several systems administrators, each with their own (sometimes overlapping) domain of control. The language needs to accommodate such distributed authorship of policies, with its inevitable conflicts.

Thus, FSL must have a clean model for reconciling conflicts, both within policies written by a single operator, and between policies written by different operators.

2.2 Implementing Policies

Policy decisions must be enforced on each packet, which places a very stringent performance requirement on any policy system. We can relax this requirement by assuming that policies are "flow-based." Operationally, by flow-based we mean that if two packets have the same header, then the policy dictates the same actions for the two packets. In a flow-based system, each time a policy decision is made, the packet header and the resulting policy action is (logically) inserted into a table of $\langle header; action \rangle$ entries, with entries timing out after a period of inactivity. When a new packet arrives, the system first checks for a matching entry in the flow table. If found, the system executes the corresponding action, otherwise it must make a policy decision. The performance requirement of the system is thus measured in terms of the number of new flows per second.

In our experience, large networks (on the order of 10,000 hosts) typically generate fewer than 10^4 flows per second. This is a demanding requirement, but is far smaller than that of a packet-based system where each packet requires an independent policy decision. For instance, a fully loaded 10 Gbps link has over 10^6 packet arrivals per second, with an average packet size of 400 bytes.

Policies dictate the actions that should be taken for a particular set of flows. The actions that operators want to implement include more than just *allow* or *deny*. They want to *rate-limit* flows, and force others to traverse *waypoints* such as proxies or firewalls. Another common policy is to protect clients from inbound connections using NAT, while allowing traffic to public servers within the same network via a DMZ.

Thus, the policy system must be able to deal with a set of various policy actions. As we explain later, the only requirement we place on these actions is that they are partially-ordered with respect to security (e.g., deny is more secure than allow, rate-limited is more secure than not limited, etc.). We say an outcome is consistent with a policy if it is at least as secure as the action dictated by the policy. Thus, a policy's action can be seen as a constraint on that flow, restricting the possible set of outcomes.

2.3 NOX

We have implemented FSL within NOX [20]. NOX is a flow-based architecture utilizing the flow-table like forwarding described above. A NOX network involves a set of "dumb" switches that simply implement flow table forwarding, and a centralized controller (or perhaps several for resilience) that guides the switches. When a packet arrives at a switch and there is no corresponding entry in the flow table, the packet is forwarded to the controller for inspection. The controller makes the requisite policy decision, and then inserts the corresponding flow-table entries into the appropriate set of switches.

Through means we don't discuss here, NOX is able to maintain secure bindings between high-level names, such as user and host names, and low-level packet fields (IP and MAC addresses). For the purposes of our discussion, we assume that NOX associates exactly eight names with each flow. Each flow involves two hosts on the network: a source and a target; likewise, each flow involves two users and two access points (locations where the hosts physically connect to the network). Each flow also plays a part in some protocol, and is either an initial message from source to target or the response to such a message. Every access control decision is based on these eight fields.

2.4 Formal Model

We can formalize the policy system as follows. The fundamental unit upon which the network acts is a *unidirectional flow*[1] which consists of values for the eight fields.

Definition 1 (Unidirectional Flow). *A unidirectional flow (uniflow for short) is characterized by an eight-tuple.*

$$\langle u_{src}, h_{src}, a_{src}, u_{tgt}, h_{tgt}, a_{tgt}, prot, request \rangle$$

- $u_{src}, u_{tgt} \in U$ *(the set of users)*
- $h_{src}, h_{tgt} \in H$ *(the set of hosts)*
- $a_{src}, a_{tgt} \in A$ *(the set of access points)*
- $prot \in P$ *(the set of protocols)*
- $request \in \{true, false\}$

Uniflows constitute the input to the access control decision maker. A security policy for NOX associates every possible uniflow with a set of constraints, and for the remainder of the paper, we suppose that a uniflow can be allowed, denied, waypointed (the route the uniflow takes through the network must include the stipulated hosts), forbidden to pass through certain waypoints (the route for the uniflow cannot include the stipulated hosts), and rate-limited. Formally, NOX policies are functions on uniflows as defined below, and FSL is a language for describing such functions.

Definition 2 (NOX Policy). *A NOX policy is a function from unidirectional flows to two sets of hosts and a natural number. The first set of hosts corresponds to the set of required waypoints, the second set to the set of forbidden waypoints, and the natural number corresponds to the allowed communication rate limit in Mb/s.*

$$U \times H \times A \times U \times H \times A \times P \times \{T, F\} \rightarrow 2^H \times 2^H \times \mathcal{N}$$

3 FSL

Conceptually, FSL is a language for representing NOX Policies, policies that mandate which constraints should be applied to which unidirectional flows. It was designed so that given a uniflow, the constraints on that flow can be computed efficiently (on the order of a tenth of a millisecond) while supporting collaborative policy authorship. After introducing the core of language, this section discusses conflicts and conflict resolution, priorities, computational complexity, and illustrates the versatility of the language by explaining how to enforce NAT and VLAN-based policies.

FSL is based on nonrecursive DATALOG with negation, where each statement represents a simple if-then relationship. It relies on six keywords, five of which appear in the conclusions of rules: *allow*, *deny*, *waypoint*, *avoid*, and *ratelimit*. *allow* and *deny* each take eight arguments corresponding to the eight fields of a uniflow, and allow administrators to instruct NOX to allow and deny, respectively, any uniflow that matches a given set of conditions.

For example, the following four rules say that a superuser has no communication restrictions.

$$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow superuser(U_s)$$
$$superuser(todd)$$
$$superuser(amy)$$
$$superuser(michelle)$$

[1]The term *unidirectional flow* is more appropriate than simply *flow* because the latter is often analogous to a nonce, which consists of two messages: one sent from the source to the destination and another one sent back. A unidirectional flow corresponds to exactly one of those messages.

The arguments to *allow* are all variables (denoted by symbols starting with a capital letter) and correspond to the user source, host source, access point source, user target, host target, access point target, protocol, and whether or not the flow is an initial request, respectively. The remaining three rules make simple declarative statements.

The keywords *waypoint*, *avoid*, and *ratelimit* take one extra argument in addition to the eight fields of a uniflow: the node that should be visited, the node that must not be visited, and the rate limit that should be imposed, respectively.

Administrators write DATALOG security policies using these five keywords to describe the NOX policy they would like enforced on uniflows. Every time a uniflow is initiated in an NOX network, the central controller queries the current FSL policy to determine which constraints should be applied. Each query supplies values for every one of the eight uniflow fields. If one of the fields is not known, because for example the uniflow concerns a machine not in the NOX network, the sixth keyword *unknown* occupies that field. Administrators can then write rules conditioned on unknown uniflow fields.

For example, to allow uniflows initiated by a desktop in the network to machines outside the network, an administrator could write the following rule.

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

In addition to writing self-contained DATALOG policies, FSL allows administrators to write policies that reference external information sources. Those sources may be defined by, for example, SQL queries over local or remote databases, hash tables, or arbitrary procedural code. The definitions for those external sources accompany an FSL policy; we will not discuss them further. In examples, we will indicate external references by underlining the reference.

The formal syntax for FSL rules and FSL policies is given below. We use standard terminology. A predicate is a symbol with an associated 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 $\neg$ applied to an atom. An expression is ground if it contains no variables.

Definition 3 (FSL Rule). *An FSL rule in the context of external references G takes the following form.*

$$h \Leftarrow b_1 \wedge \cdots \wedge b_n \wedge \neg c_1 \wedge \cdots \wedge \neg c_m$$

- *h, the head, and every b_i and c_i, collectively called the body, are atoms.*
- *Every variable in the body must appear in the head.*
- *Predicates allow and deny have arity eight; waypoint, avoid, and ratelimit have arity nine; unknown is an object constant.*
- *When waypoint, avoid, or ratelimit appear in the head, the last argument is an object constant.*
- *h does not contain any predicate $g \in G$.*

For those interested in technical details, notice the second condition: that every variable that appears in the body of a rule must appear in the head. This condition differs from the traditional safety[2] restriction in DATALOG in order to significantly simplify FSL's implementation. See Section 5 for details.

Definition 4 (FSL Policy). *An FSL policy is a set of FSL rules Δ such that the dependency graph $\langle V, E \rangle$ is acyclic.*

V: *the set of predicates occurring in Δ*

E: *contains a directed edge from u to v exactly when there is a rule in Δ where the predicate in the head is u and the predicate v appears in the body.*

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

The semantics of FSL begins with the external references. From a logical perspective, every external reference implementation must ensure that every statement is either true or false—it must represent a model. The actual implementation could be a database or procedural code. Moreover, we assume the external references are aware of the reserved word *unknown*.

Definition 5 (External Reference Implementation). *An implementation for the external reference g_i of arity n is a set of ground atoms of the form $g_i(a_1, \ldots, a_n)$.*

Entailment for FSL is based on the usual stratified semantics from deductive databases and logic programming [33].

Definition 6 (Basic Entailment). *Let Δ be an FSL policy, Γ the external reference implementations, and ϕ a first-order sentence. Let M be the model that the stratified semantics assigns to $\Delta \cup \Gamma$ when the universe consists of the objects mentioned in Δ, Γ, and ϕ. $\models_M$ denotes the usual definition of satisfaction in the model M.*

$$\Delta \cup \Gamma \models \phi \text{ if and only if } \models_M \phi$$

One important feature of FSL is that the order in which rules appear is irrelevant. This makes combining a set of policies straightforward: collect the statements made in all of the policies. In contrast, languages where the order of statements is relevant, e.g. firewall configuration languages [37], make it hard for a computer system to combine independently authored policies automatically. Besides affording the independence administrators are accustomed to today when defining access control policies, embracing collaborative policy authoring has an additional benefit: real-world disagreements can be automatically pinpointed by policy analysis tools.

In FSL, a conflict arises for a given uniflow when a policy makes mutually exclusive decisions about what to do with that uniflow. Unlike traditional access control where all conflicts arise from being both granted and denied access, conflicts in FSL are produced in several different ways. Clearly a uniflow cannot be allowed and denied simultaneously. More interestingly, a uniflow cannot be denied and at the same time forced through an intermediate waypoint. Neither can a flow pass through a waypoint and avoid passing through that same waypoint.

Because NOX cannot enforce a conflicting policy (no uniflow can be both allowed and denied), conflict resolution is built into the language. Since FSL is a language concerned with security, its conflict resolution mechanism errs on the side of caution. A conflicting set of constraints is always resolved to produce the most restrictive constraint set. Deny is more restrictive than a set of required and prohibited waypoints, and waypoints are more restrictive than a simple allow. A set of contradictory waypoints is equivalent to deny. A lower rate limit is more restrictive than a higher rate limit.

Similarly, because NOX cannot enforce an incomplete policy (every uniflow must be either allowed or denied), policy completion is also built into the language, but can easily be overridden. In contrast to conflict resolution, policy completion errs on the side of permissiveness by allowing all unconstrained uniflows. The rationale is a practical one: when installing a new system, it is useful to have all machines able to communicate to ensure the installation is working correctly.

The conflict resolution and policy completion schemes are built into the version of entailment that NOX uses to make authorization decisions. It relies on Basic Entailment (Definition 6) and ensures that every FSL policy corresponds to an NOX Policy (Definition 2).

Definition 7 (FSL Entailment). *Suppose Δ is an FSL policy. FSL Entailment, $\models_{FSL}$ is defined in terms of Basic Entailment, $\models$.*

1. $\Delta \models_{FSL} deny(a_1, \ldots, a_8)$, *if*
   $\Delta \models deny(a_1, \ldots, a_8)$ *or* $\Delta \models \exists x.(waypoint(a_1, \ldots, a_8, x) \wedge avoid(a_1, \ldots, a_8, x))$.
2. $\Delta \models_{FSL} waypoint(a_1, \ldots, a_8, a_9)$, *if* $\Delta \models waypoint(a_1, \ldots, a_8, a_9)$ *and (1) does not hold.*

3. $\Delta \models_{FSL} avoid(a_1, \ldots, a_8, a_9)$, *if* $\Delta \models avoid(a_1, \ldots, a_8, a_9)$ *and (1) does not hold.*

4. $\Delta \models_{FSL} allow(a_1, \ldots, a_8)$, *if (1-3) do not hold.*

5. $\Delta \models_{FSL} ratelimit(a_1, \ldots, a_8, m)$ *if and only if* m *is the minimum of*

$$\{maxrate\} \cup \{r | \Delta \models ratelimit(a_1, \ldots, a_8, r)\}$$

One of the benefits of building a conflict resolution mechanism into the language is that users can leverage it to express certain policies. For example, an open policy allows everything not explicitly denied. The following two rules deny all communication initiated by a blacklisted user but allow everything else.

$$\begin{array}{l} allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \\ deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow \underline{blacklist}(U_s) \end{array} \quad (1)$$

In contrast, the conflict resolution mechanism does not make it easy to express a closed policy: where everything not explicitly allowed is denied. Adding permissive statements will not affect a too-restrictive policy; consequently, FSL provides a different mechanism for relaxing security. An FSL cascade, named for the cascading style sheets popular on the web, is a prioritized series of FSL policies, written $P_1 < \cdots < P_n$. Any policy in the ordering overrides all of the policies less than it.

For example, to define a closed policy, one could construct a cascade with two policies: $P_1 < P_2$. P_2 would describe all of the uniflows 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 8 (FSL Cascade). *An FSL cascade consists of a finite set of FSL 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 uniflow is the only policy that says anything about that uniflow. In other words, policy P_i determines the constraints on a uniflow if P_i constrains the uniflow and there is no other P_j that constrains that uniflow where $j > i$. The formal semantics is based on a precise definition of the constraints imposed on a uniflow by a policy.

Definition 9 (Uniflow Constraints). *Consider a uniflow* $u = \langle a_1, \ldots, a_8 \rangle$. *When* P *is an FSL policy, let* $C_P(u)$ *be the smallest set that includes*

$$\begin{array}{rcl} allow(a_1, \ldots, a_8) & if & P \models allow(a_1, \ldots, a_8) \\ deny(a_1, \ldots, a_8) & if & P \models deny(a_1, \ldots, a_8) \\ waypoint(a_1, \ldots, a_8, a) & if & P \models waypoint(a_1, \ldots, a_8, a) \\ avoid(a_1, \ldots, a_8, a) & if & P \models avoid(a_1, \ldots, a_8, a) \\ ratelimit(a_1, \ldots, a_8, a) & if & P \models ratelimit(a_1, \ldots, a_8, a). \end{array}$$

$C_P(u)$ *denotes the set of constraints imposed by* P *on uniflow* u.

Above we used $\models$, i.e. entailment without conflict resolution, resulting in a definition for Uniflow Constraints that allows conflicts. Replacing $\models$ with $\models_{FSL}$ produces a definition where conflicts have been resolved. The definition we provide loses less information than the alternative, and since resolving conflicts and computing the constraints imposed by a cascade are commutative operations (applying them in both orders produces the same result), the alternate definition can be given in terms of the one above.

Definition 10 (FSL Cascade Semantics). *Consider a uniflow* $u = \langle a_1, \ldots, a_8 \rangle$ *and a cascade* $P_1 < \cdots < P_n$. *If* P_u *is the maximum* P_i *such that* $C_{P_u}(u)$ *is nonempty, then the constraints imposed on* u *by the cascade are exactly the constraints imposed on* u *by* P_u.

$$C_{P_1 < \cdots < P_n}(u) = C_{P_u}(u)$$

FSL was designed to make *writing* authorization policies convenient, but it was also designed to make *implementing* such policies efficient; hence, the complexity of FSL is important. The computational complexity of Basic Entailment turns out to be PSPACE-complete, just like traditional nonrecursive DATALOG with negation. Adding in conflict resolution and cascades increases the complexity by only a linear factor.

Theorem 1. *Let Δ be an FSL policy, Γ the external reference implementations, and ϕ a first-order sentence. Determining whether or not $\Delta \cup \Gamma \models \phi$ is PSPACE-complete.*

Proof. The inclusion in PSPACE follows because an FSL policy is a special case of a nonrecursive logic program with negation and without function constants, which is well-known to belong to PSPACE. The hardness proof demonstrates how to encode an arbitrary quantified boolean formula (QBF) as an FSL policy. The key insight into the proof is that existential variables can be simulated in the body of FSL rules by duplicating the rule for each possible variable instantiation. For example, the DATALOG rule $q(X) \Leftarrow p(X, Y)$, where Y can only be either a or b can be written in FSL as the following two rules.

$$q(X) \Leftarrow p(X, a)$$
$$q(X) \Leftarrow p(X, b)$$

For encoding QBF formulae, this rule duplication technique is only polynomially larger than the usual DATALOG encoding. See Appendix B for full details. □

The PSPACE complexity of FSL appears to destroy any hope of answering queries in the requisite tenth of a millisecond, but fortunately another of the usual DATALOG complexity results holds for FSL policies: restricting the arity of predicates to a constant reduces the complexity of entailment to polynomial time. The implementation discussed in Section 4 employs this constant-arity restriction to guarantee polynomial-time access control decision-making.

Finally, we illustrate how to express two standard security mechanisms for networks in FSL: network address translation (NAT) and virtual networks (VLANs). NAT provides security by disabling incoming connections for certain hosts, e.g. laptops, and disabling outgoing connections for servers.

$$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow \underline{laptop}(H_t) \wedge Req = true$$
$$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow \underline{server}(H_s) \wedge Req = true$$

Virtual networks give the appearance that certain hosts are isolated from the rest of the network. For example, the FSL implementation for a VLAN comprised of hosts a, b, and c is shown below.

$$vlan(a)$$
$$vlan(b)$$
$$vlan(c)$$
$$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow vlan(H_s) \wedge \neg vlan(H_t)$$
$$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow \neg vlan(H_s) \wedge vlan(H_t)$$

4 Implementation

We have implemented FSL as the default policy language for the NOX system [20] and are using it to manage our internal network. Our implementation supports integration with external authentication stores (LDAP and AD), dynamic, incremental policy updates, and runtime group membership changes. All language features are supported with the exception that predicates (except the keywords) are restricted to one argument. Speed critical operations (such as runtime policy checking) are implemented in C++, while compilation and various integration components are written in Python.

In this section we describe our implementation and how it interoperates with other networking functions in NOX. We then describe our use of FSL internally. Finally, to demonstrate how the system performs under load (we are targeting networks of tens of thousands of hosts), we present performance analysis over stress-level workloads.

4.1 Implementation Environment

As mentioned previously, NOX is a flow-based network architecture in which each new flow on the network is sent to a logically centralized controller similar to [13]. Figure 1(a) shows the main components of an NOX network. The controller runs software responsible for authenticating users and hosts, determining the switch-level topology, and calculating network routes. The controller also provides a general programmatic environment in which "applications" written in C++ or Python can gain access to network events and dictate how the network should handle a given flow. We implemented FSL as one such application.

Basically, a NOX application is a set of handlers which are invoked during network events. The events an application can listen for include *new-flow* events, *host join* and *leave* events, and *user authentication* events[3]. Handlers can access the network topology, send traffic out of any switch on the network, and manage switch flow-tables. By adding entries to the switch flow-tables, an application is able to permit a flow on the network, control its path, and specify rate limits.

NOX handles the authentication of all users, hosts, and switches using credentials registered by the network operator either directly with NOX or in an external authentication store (such as LDAP or AD). As we describe below, we use FSL to define the default connectivity needed to bootstrap the authentication process. Once a principal authenticates, NOX creates a mapping between the principal's location and network addresses (*e.g.*, MAC and IP), and its registered name. These mappings allow *new-flow* events to be associated with source and destination identities at a name level (hosts, users, switches).

NOX uses an approach simlar to [14] and [13] to provide a secure network namespace that is resistant to source spoofing attacks from directly connected hosts. It also suffers from the same limitations. Most notably, NOX cannot differentiate between multiple users on the same host. Therefore, for sources and destinations with multiple users, our FSL implementation always uses the least restrictive policy among the set of active users. Admittedly, this policy is the opposite of the conflict resolution policy for FSL (most restrictive). In the future, we plan to extend NOX control to the end host, allowing it to differentiate between multiple users on the same host.

4.2 Operational Overview

Figure 1(b) shows how our FSL implementation integrates with NOX's standard network functions. It relies on other NOX applications to perform topology discovery, routing, authentication, and flow setup.

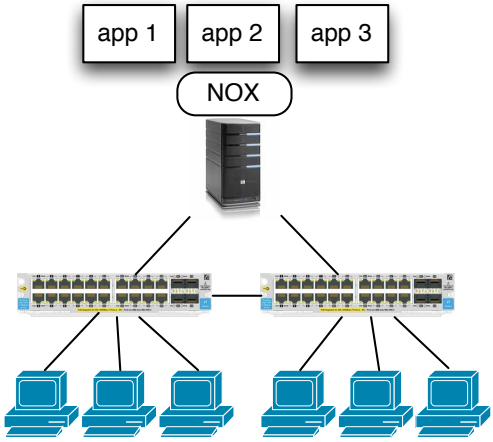
The FSL policy is declared in one or more files which are compiled into a low-level lookup tree. The compilation process checks all available authentication stores to verify that the principal names used in the policy file exist.

Packets received by NOX (for which there is no existing switch entry), are first tagged with all associated names and groups. The binding information between names and addresses happens at principal authentication. If binding information does not exist for the packet, the host and user are assumed to be unauthenticated, and the keyword *unknown* is used in place of the principal names.

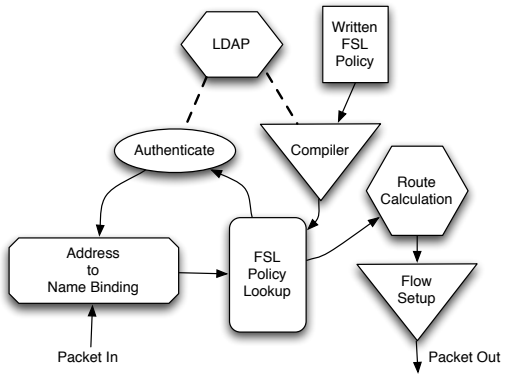
In our implementation, we use an extension of FSL to control authentication policies (admission control). This is done by using the *unknown* keyword in FSL rules, and passing all matching packets to the authentication subsystem. This allows the administrator to specify the required authentication scheme as an action of an FSL condition. We describe this further in Section 4.6.

NOX implements FSL cascades using priorities. Each rule in a single cascade file is associated with the same priority level, with other files utilizing either lower or higher levels depending on their position in the hierarchy. For authenticated packets, the FSL subsystem finds the highest priority matching rule(s) and returns the resulting policy decision. If the decision is not a deny, the constraints are passed to the routing subsystem which will attempt to find a policy compliant route. Finally, NOX will set up the path in the network (specifying a rate limit if necessary) and re-inject the packet at the first hop switch. Once the flow is set up, all subsequent packets from the flow will be forwarded directly by the switch without having to go to the controller.

[3]While a number of other events are supported by the system, they are not relevant to this paper. [20] contains a more detailed description of the NOX software architecture.



(a) In a NOX network, applications provide all network control functions by interposing on all new flows on the network.



(b) Component diagram of our FSL implementation's integration with NOX networking functions

Figure 1: Device and component views of NOX

4.3 Compiler

The FSL compiler performs three tasks: it checks for valid usage of principals and predicates, it detects static conflicts, and it translates rules into a low-level byte format for insertion into a decision tree allowing fast lookup per packet.

Our compiler first parses the policy file and verifies that the principal names used exist in at least one of the available authentication stores. It also finds and reports detectable static rule conflicts; support for dynamic group membership obviates its ability to determine all possible conflicts. Conflicts arising due group membership changes are handled at runtime in the manner described in Section 3.

Once the file is parsed, the compiler stores the rules persistently in a canonicalized internal format. This is used to determine which rules have been added, removed or modified during a policy update, allowing for incremental updates of the policy at runtime. Generally, updates to the policy only require the addition and deletion of a few rules, rather than deleting and re-inserting the full policy.

4.4 Name to Address Bindings

NOX associates high-level names and groups with packet header values at authentication time. When a principle 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 (in the case of standard directory services). On succesful authentication, the names are cached locally at the controller. On each new flow, the source and destination header fields are hashed and used to retrieve the cached name information which is subsequently passed to the FSL lookup tree.

In principle, changes to group membership should be reflected immediately in the internal cache established at authentication time. While this is the case for our built-in authentication store, our implementation currently does not support dynamic updates to group membership in external directories. In this case, modification to the group membership of an authenticated principal may not take effect until it reauthenticates.

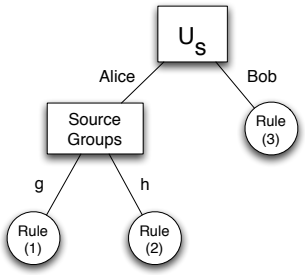


Figure 2: Example decision tree

4.5 Policy Lookup and Enforcement

The policy evaluation engine is built around a decision tree intended to minimize the number of rules that need be checked per flow. The tree partitions the rules based on the eight uniflow fields and the set of groups, resulting in a compact representation of the rule set in a ten-dimensional space. Negative literals are ignored by the indexer and evaluated at runtime.

For example, the rule

$$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow U_s = alice \wedge g(H_s) \wedge h(A_s) \wedge \neg p(A_t)$$

constrains U_s, mentions the source groups g and h, and does not mention any target groups. This rule would belong to the node in the decision tree where $U_s = alice$ and the source is constrained by either g or h.

Each node in the decision tree has one child for each possible value for the dimension that node represents, e.g. a node representing U_s has one child for each value U_s is constrained to 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 above, 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 ten 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.

Recall that group membership is computed at authentication time. We will use G_s to denote all those groups to which the source of a uniflow belongs and G_t to denote the groups to which the target of a uniflow belongs. To find all rules that pertain to any given uniflow, the usual decision-tree algorithm is used with one exception: multiple branches may be followed at any given node. In particular, the ANY branch is always followed, and for branches splitting on source groups and target groups, all children that belong to the uniflow's G_s and G_t respectively, are followed.

For example, consider the policy that includes the following rules and the associated decision tree in Figure 2.

1. $allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow U_s = alice \wedge g(H_s)$
2. $allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow U_s = alice \wedge g(H_s) \wedge h(A_s) \wedge \neg p(A_t)$
3. $allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow U_s = bob \wedge g(A_s)$

The root of the decision tree could either split on U_s or the source groups, since they are equally good at widely segmenting the rules. Suppose it splits on U_s. Then the root has two children: one for *alice* and one for *bob*. (Because every rule constrains U_s there is no need for ANY.) Rule (3) belongs to the *bob* child, which requires no further splitting. The other two rules are assigned to the *alice* child. The *alice* node is split on the source group dimension, where there are two options: g and h. Rule (1) is placed into the g child, and rule (2) is placed into either the g or h child. A uniflow originating from *alice* where the source

fields only belong to group g would evaluate just rule (1), assuming (2) is placed into the h child and both (1) and (2) otherwise. If instead *alice* initiated a uniflow where the source fields belong to both g and h, then rules (1) and (2) are both evaluated, regardless of where rule (2) is placed in the tree.

The cost of answering an FSL entailment query 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 built into the system: actions of all matching rules are collected, and the most secure, potentially composite, action is returned. An FSL cascade is implemented by evaluating rules in priority order; once a match is found, only the remaining rules of the same priority are evaluated.

4.6 Bootstrapping Authentication

Authenticating users and hosts generally requires some default connectivity which is dependent on the admission policy. For example, we are preparing for a deployment in 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 web-servers, and from the web-server to the directory servers.

We have extended our FSL implementation to also support the declaration of admission control policies and the requisite connectivity for authentication. Similar to access controls, the admission policy is a set of FSL-like rules in which the action denotes the authentication type. The predicates used in the rules include access points (*e.g.*, $wireless(A_s)$), protocols, IP prefixes, and special purpose built-in groups such as *all-registered-macs*. For example, the following policy snippet sets up default connectivity for authentication, and redirects all unauthenticated wired hosts to a captive web portal.

$$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = ARP$$
$$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = DHCP$$
$$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow H_s = authentication\_server \wedge Prot = HTTP$$
$$http - redirect(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow H_s = unknown \wedge Prot = HTTP$$

$$cascade()$$

$$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow H_s = unknown$$

In our network, all hosts on wired access points are automatically authenticated if they have a registered MAC address. Whereas users connecting via wireless are required to use a captive web portal. While logically the admission control policy is distinct from the access control policy (since automated conflict resolution is no longer viable), we use the same internal mechanisms to enforce it.

4.7 Experience and Performance

The security policy of our local area network, consisting of roughly 50 hosts connected through 4 switches, is managed by our FSL implementation running on NOX. The policy contains 24 rules declared over 64 principals and 11 groups including workstations, servers, printers, and mobile devices.

We include a subset of the policy in Appendix A. Roughly, our internal policy allows unrestricted ARP and SSH between all hosts. Test servers are not allowed to communicate externally. 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 which allow monitoring and diagnostic traffic between an administrative host and all switches. The lowest priority rule in the policy file matches all flows and denies the traffic.

Round trip flow setup latencies (involving two permission checks, route calculations, and flow-entry setups) are generally under 50ms. Once the policy decision has been made for each uniflow, all subsequent packets for that flow are forwarded at line speed (as supported by the switch forwarding hardware). On a network of this size, there is not enough traffic to stress our implementation (we generally see less than 20

new flow setups/s). However in benchmarks using generated traffic, our implementation running with our internal policy file supports permission checks on over 60,000 flows/s, an order of magnitude higher than any network we have measured. We present more performance tests of the system under load below.

Our experience has been that our simple rule set provides sufficient connectivity for day to day operations without requiring constant maintenance. Furthermore, we find that almost all of the rules are declared over classes of devices that connect to the network (as opposed to individual machines or users) and therefore we expect the policy file to grow slowly with the number of managed principals. We expect to explore this in more detail shortly as we are currently preparing for two much larger deployments of FSL in networks of hundreds and thousands of hosts.

In preparation for broader deployment we have tested our implementation's performance under generated workloads. Unfortunately it is difficult to produce meaningful results without access to sample policies written for real networks. In the degenerate case, enforcement of a policy with many ANY rules can scale linearly as the rule set size grows. Conversely, a large policy containing only exact match rules could force construction of a maximum depth tree, incurring significant hashing overhead.

We test the performance and memory overhead of our implementation with policy files of increasing size (table 1). 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 2 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 [14, 13], 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
500 rules	78,534	1	6
1,000 rules	75,414	2	7
5,000 rules	71,702	9	8
10,000 rules	67,843	56	9

Table 1: Performance and memory overhead of our FSL implementation over policies with increasing rule count. All policies contain exact match rules and average $log_2(\#rules)$ matching rules per flow. 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
500 rules	85,373	1	4
1,000 rules	76,336	2	10
5,000 rules	54,416	9	30
10,000 rules	46,956	38	52

Table 2: 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

FSL is based on a restricted form of DATALOG [24, 8, 27, 36, 19, 28, 7] with negation [31, 8, 24] so that authorization policies can be authored in distributed settings [11, 23, 31, 5]. Because disagreements naturally arise among administrators, FSL was designed so that those disagreements would manifest as conflicts in policies [35, 24, 10, 8, 15, 18, 31, 9, 5]. Conflicts between policy modules entered by different administrators can be detected by automated methods [30, 5, 24] that can be built into policy management tools. At the same time, because FSL policies are used by real systems [12, 17, 34], any conflicts that are not resolved by administrative tools must be resolved automatically at enforcement time [35, 24, 10, 8, 15, 18, 31, 9]. Unlike prior languages, FSL conflict resolution is complicated by the fact that policies can reference external sources [31, 11, 27], which can change independent of the policy. This feature of FSL is essential for large-

scale deployment, since it is impractical to ask large organizations to rewrite their organization charts in a network policy language. Finally, FSL employs priorities (called "overrides" in [11] and "exceptions" in [10]) via FSL Cascades, to help administrators express certain types of policies.

FSL lacks certain features found in the literature. The most common reason a feature was excluded is the performance requirement. Below, we enumerate some of the limitations of FSL and explain why the limitations exist.

Datalog based. Other languages are based on fundamentally different and often more expressive logics [21, 35, 4, 17, 18, 6, 15]. First, DATALOG was chosen because of implementation concerns: the simpler the language, the simpler and faster the implementation. Second, additional expressiveness allows users to state policies that can be difficult to enforce. In first-order logic, a policy might say to allow one of two uniflows without saying which. Third, DATALOG is closer to traditional CS programming languages than other logics, which is important because policy authors are network operators (who may be logic novices).

Existential variables are disallowed. Existential variables (those that occur in the body of a rule but not in the head) are usually included in DATALOG languages because they are useful in a variety of circumstances. For example, Role-based Access Control (RBAC) has been explained very concisely using existential variables [16]. Without existential variables, the implementation is far simpler because there is no search through variable assignments. $q(X) \Leftarrow p(X, Y) \wedge r(Y)$ says that $q(t)$ is true if there is some u such that $p(t, u) \wedge r(u)$ is true. The implementation needs to search through the possible assignments to Y. FSL's design and application ensure that such a search need never occur.

Recursion is disallowed. Policy languages that support recursion without negation [27], stratified recursion [24] and arbitrary [8] recursion occur in the literature. Without recursion, FSL's implementation is both simpler and faster.

Conflict resolution is built into the language. Sometimes the conflict-resolution scheme for a language is built-in [31, 15, 9], but sometimes it is defined by the user [8, 35, 18]. Sometimes, conflicts are only detected but not resolved [5], and other times conflicts are handled by a combination of detection and user-defined resolution [24]. Our choice to fix the conflict resolution scheme comes about because conflicts and conflict resolution in FSL are more complex than usual.

Unlike common access control decisions, which either allow or deny an action, each decision in FSL consists of a set of constraints, e.g. pass through waypoint A but do not pass through B. Conflict resolution consists of mapping one set of constraints to another set; moreover, only certain combinations of constraints are actually enforceable. A system cannot force a uniflow through waypoint A and at the same time avoid waypoint A. It is likely that if conflict-resolution were left to user specification, some conflicts would not actually be resolved, and a built-in resolution mechanism would be necessary anyway.

Delegation is not directly supported. One common concern in authorization logics is the ease with which one principal can delegate rights to another [17, 29]. Such concerns are important when there are certain actions that can change which future actions are allowed and denied. In the setting for which FSL was designed, there are no rights-changing actions. The answer to all authorization requests is the same for all time (except for external reference changes); hence, delegation primitives have been omitted.

FSL policies are timeless. Some policy languages include constructs that reference temporal events [22, 17, 35, 29, 31, 25], e.g. if A communicated with B in the past, then disallow B from communicating with C. History acrues very quickly with 10^4 flows initiated per second. Storing that much information is impractical, and even if it were practical, the system would not be able to answer queries about it quickly enough.

Policy enforcement is centralized. Languages built for trust management are designed so that the authorization policy can be authored by a group of people working independently. But unlike FSL, which enforces that policy at a central location, trust management languages enforce each policy in a distributed fashion [4, 25, 36, 28, 26, 32, 7]. Distributed enforcement is the subject of future work.

Metalevel policy operations are limited. FSL supports two metalevel policy operations: combining a set of policies, and prioritizing a set of policies (FSL cascades). Among the other metalevel operations included in [11], the most germane is scoping, also used in [31]. Scoping is an operation that restricts a given policy to a certain class of objects. For example, in a university setting scoping could restrict a CS

administrator from constraining interdepartmental server communications in the Math department. Adding scoping to FSL is the subject of future work.

Finally, it is worth highlighting the features of FSL that we have been unable to find elsewhere in the literature.

Access control decisions are constraint sets. Instead of either allowing or denying every request, FSL prescribes a set of constraints that apply to that request. Allow and deny are special cases.

Conflict resolution maps one constraint set to another. Because access control decisions result in a set of constraints, a conflict can be attributed to three or more of those constraints. Conflict resolution takes as input one constraint set and outputs another constraint set.

6 Conclusion

Motivated by the opportunities provided by emerging network technologies, and the need for network operators and administrators to express general constraints using distributed and incrementally authored policies enforced efficiently at line speeds, we developed a network security policy framework around a language expressing flow-based network policies. Our policy language FSL allows operators to selectively allow network access, while also imposing constraints on usage rate, allowed network paths (which may be required to avoid or include certain nodes), and directionality (such as restricting clients to outbound flows only). As the name *Flow Security Language (FSL)* implies, the key underlying concept is network flow. In particular, after trying several alternatives, we decided to take unidirectional flows as the basic unit of network access, regarding bidirectional flow as comprising two related uniflows. By associating additional characteristics such as flowrate and initiating party with each flow, we achieve a systematic network flow analog of the traditional access-control matrix (of, say, users, filenames, and read-write-execute permissions). Building on this foundation, FSL allows succinct, structured, high-level specification of allowed flows, freeing network administrators from the drudgery of configuring myriad router ACLs, firewalls, NATs and VLANs to achieve comprehensive and conceptually straightforward network usage policies.

FSL is a declarative policy language based on nonrecursive DATALOG with structured negation. The declarative nature of FSL makes it possible for separate network administrators to separately express their policies and automatically combine those policies. We show how combined policies with potentially conflicting components can be enforced efficiently. In addition, logic-based algorithms that detect and resolve conflicts in predictable and reliable ways may be incorporated into policy development environments. FSL also supports prioritized policy combination, which is a natural way to express many policies and enables incremental policy updates.

We demonstrated FSL's use in practice by implementing it within the NOX network architecture, running it in our internal network for nearly a year, and subjecting the system to additional tests using much more demanding artificially generated loads. We show through performance analysis that our implementation has modest memory requirements and can scale to very large networks while supporting policy files of tens of thousands of rules. We are preparing to further explore FSL's application in operational neworks through much larger deployments in the near future.

References

[1] Consentry networks homepage. `http://www.consentry.com`.

[2] Nevis networks homepage. `http://www.nevisnetworks.com`.

[3] Vernier networks homepage. `http://www.verniernetworks.com`.

[4] M. Abadi, M. Burrows, and B. Lampson. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

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

[6] D. Basin, E.-R. Olderog, and P. E. Sevinc. Specifying and analyzing security automata using CSP-OZ. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 70–81, 2007.

[7] M. Y. Becker, C. Y. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pages 3–15, 2007.

[8] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, 2003.

[9] E. Bertino, E. Ferrari, F. Buccafurri, and P. Rullo. A logical framework for reasoning on data access control policies. In *Proceedings of the IEEE Computer Security Foundations Workshop*, 1999.

[10] E. Bertino, S. Jajodia, and P. Samarati. A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems*, 17(2):101–140, 1999.

[11] P. A. Bonatti, S. D. di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 164–173, 2000.

[12] K. Borders, X. Zhao, and A. Prakash. CPOL: High-performance policy evaluation. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 147–157, 2005.

[13] 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.

[14] M. Casado, T. Garfinkle, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Proc. 15th USENIX Security Symposium*, Vancouver, BC, Aug. 2006.

[15] L. Cholvy and F. Cuppens. Analyzing consistency of security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1997.

[16] J. Crampton. Understanding and developing role-based administrative models. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 158–167, 2005.

[17] J. Crampton, G. Loizou, and G. Oshea. A logic of access control. *The Computer Journal*, 44(1):137–149, 2001.

[18] F. Cuppens, L. Cholvy, C. Saurel, and J. Carrere. Merging security policies: analysis of a practical example. In *Proceedings of the IEEE Computer Security Foundations Workshop*, 1998.

[19] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[20] 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.

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

[22] K. Irwin, T. Yu, and W. H. Winsborough. On the modeling and analysis of obligations. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 134–143, 2006.

[23] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.

[24] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–42. IEEE Press, 1997.

[25] A. J. Lee and M. Winslett. Safety and consistency in policy-based authorization systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 124–133, 2006.

[26] N. Li, B. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. In *Proceedings of the ACM Transactions on Information and System Security*, pages 128–171, 2003.

[27] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the Symposium on Practical Aspects of Declarative Languages*, 2003.

[28] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[29] N. Li and M. V. Tripunitara. On safety in discretionary access control. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 96–109, 2005.

[30] N. Li, M. V. Tripunitara, and Q. Wang. Resiliency policies in access control. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 113–123, 2006.

[31] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An access control language for security policies with complex constraints. In *Proceedings of the Network and Distributed System Security Symposium*, 2001.

[32] R. L. Rivest and B. Lampson. SDSI - a simple distributed security infrastructure. Technical report, Massachusetts Institute of Technology, 1996.

[33] J. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989.

[34] D. S. Wallach and E. W. Felton. Understanding java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63, 1998.

[35] D. Wijesekera and S. Jajodia. Policy algebras for access control - the predicate case. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 171–180, 2001.

[36] M. Winslett, C. C. Zhang, and P. A. Bonatti. PeerAccess: A logic for distributed authorization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 168–179, 2005.

[37] L. Yuan and H. Chen. FIREMAN: A toolkit for firewall modeling and analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 199–213, 2006.

A Example Policy Cascade

The following policies are ordered from highest to lowest: $P_1 < P_2 < P_3 < P_4$. The first policy shown below overrides all those after it—likewise for the other policies.

Policy P_4
allow ARP and DHCP
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = arp$
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = dhcps \wedge H_t = gateway$
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = ssh \wedge \underline{computer}(H_s)$

allow computers to ssh anywhere
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = ssh \wedge \underline{computer}(H_s)$

allow internal monitoring flows: proprietary protocols registered with the system
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1616 \wedge H_s = badwater$
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1717 \wedge H_s = badwater$
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1818 \wedge H_s = badwater$
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1616 \wedge H_t = badwater$
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1717 \wedge H_t = badwater$
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Prot = 1818 \wedge H_t = badwater$

Policy P_3
dissallow testing machines from communicating externally
$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow testing(H_s)$
$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow testing(H_t)$

servers should be inbound-only
$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Req = true \wedge \underline{server}(H_s)$
$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Req = true \wedge \underline{printer}(H_s)$

laptops and mobile devices should be outbound-only
$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Req = true \wedge \underline{mobile}(H_t)$
$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow Req = true \wedge \underline{laptop}(H_t)$

Policy P_2
allow known devices to communicate as long as they abide by the
previous rules.
$allow(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req) \Leftarrow all(H_s)$

Policy P_1
default deny
$deny(U_s, H_s, A_s, U_t, H_t, A_t, Prot, Req)$

B Proofs

Theorem 2 (Without keywords FSL is PSPACE-complete). *Given an FSL rule set Δ and an atom a, checking whether $\Delta \models a$ is PSPACE-complete.*

Proof. Because FSL is a form of function-free, recursion-free logic programming, and entailment for the latter is PSPACE-complete, entailment for FSL belongs to PSPACE. To show entailment is PSPACE-hard, we perform a reduction from QBF.

Consider any quantified boolean formula in prenex form: $Q_1 x_1 \ldots Q_n x_n . \phi(x_1, \ldots, x_n)$. The FSL query that represents the value of this QBF will be written as

$$val_{Q_1 x_1 \ldots Q_n x_n . \phi(x_1, \ldots, x_n)}.$$

The proof demonstrates how to define this predicate in polynomial time. The first step is quantifier elimination.

If Q_1 is $\forall$, then the query is defined as follows.

$$val_{Q_1x_1\ldots Q_nx_n.\phi(x_1,\ldots,x_n)} \Leftarrow val_{Q_2x_2\ldots Q_nx_n.\phi(x_1,\ldots,x_n)}(1) \wedge val_{Q_2x_2\ldots Q_nx_n.\phi(x_1,\ldots,x_n)}(0)$$

Here 1 represents true, 0 represents false, and $val_{Q_2x_2\ldots Q_nx_n.\phi(x_1,\ldots,x_n)}$ is a unary predicate that is true for all those u such that $Q_2x_2\ldots Q_nx_n.\phi(u,x_2,\ldots,x_n)$ is true. Otherwise, Q_1 is $\exists$, and the definition is given by two rules.

$$\begin{aligned} val_{Q_1x_1\ldots Q_nx_n.\phi(x_1,\ldots,x_n)} &\Leftarrow val_{Q_2x_2\ldots Q_nx_n.\phi(x_1,\ldots,x_n)}(1) \\ val_{Q_1x_1\ldots Q_nx_n.\phi(x_1,\ldots,x_n)} &\Leftarrow val_{Q_2x_2\ldots Q_nx_n.\phi(x_1,\ldots,x_n)}(0) \end{aligned}$$

Both cases require constructing definitions for additional predicates. The process is very similar to what is shown above. After constructing k definitions, we need to define

$$val_{Q_{k+1}x_{k+1}\ldots Q_nx_n.\phi(x_1,\ldots,x_n)}(x_1,\ldots,x_k).$$

If Q_{k+1} is $\forall$, the definition is as follows; otherwise, the definition requires two rules as shown above.

$$\begin{aligned} &val_{Q_{k+1}x_{k+1}\ldots Q_nx_n.\phi(x_1,\ldots,x_k)}(x_1,\ldots,x_k) \Leftarrow \\ &\quad val_{Q_{k+2}x_{k+2}\ldots Q_nx_n.\phi(x_1,\ldots,x_n)}(x_1,\ldots,x_k,1) \wedge val_{Q_{k+2}x_{k+2}\ldots Q_nx_n.\phi(x_1,\ldots,x_n)}(x_1,\ldots,x_k,0). \end{aligned}$$

Notice two things. First, in every rule, the variables in the body also appear in the head, making each rule a valid FSL rule. Second constructing the rules corresponding to each quantifier takes time linear in the original sentence, and there are no more than a linear number of quantifiers; thus, the total cost is at most quadratic in the size of the original sentence.

After eliminating all n quantifiers, the rule set requires a definition for

$$val_{\phi(x_1,\ldots,x_n)}(x_1,\ldots,x_n)$$

where $\phi(x_1,\ldots,x_n)$ is quantifier-free. The definition should be true for exactly those assignments to x_1,...,x_n that satisfy the boolean formula $\phi(x_1,\ldots,x_n)$. Suppose that $\phi(x_1,\ldots,x_n)$ is written in conjunctive normal form.

$$((p_{11} \vee \cdots \vee p_{1k_1}) \wedge \cdots \wedge (p_{m1} \vee \cdots \vee p_{mk_m}))$$

Here p_{ij} represents x_l or $\neg x_l$ for l in $\{1,\ldots,n\}$.

Just like the case of quantifiers, the definition for $val_{\phi(x_1,\ldots,x_n)}(x_1,\ldots,x_n)$ is broken into a definitions for a series of predicates: val_1,...,val_m. Predicate val_i corresponds to the evaluation of $\phi(x_1,\ldots,x_n)$ when only considering the first i clauses. The definitions are indicated below. We use two macros. $var(p_{ij})$ is replaced by the variable mentioned in p_{ij}, and $sign(p_{ij})$ is replaced by 1 if the literal p_{ij} is positive and 0 otherwise. Thus, each instance of $var(p_{ij}) = sign(p_{mn})$ is positive equality atom comparing a variable and 1 or 0, e.g. $x = 1$.

$$\begin{aligned} &val_1(x_1,\ldots,x_n) \Leftarrow var(p_{11}) = sign(p_{11}) \\ &\vdots \\ &val_1(x_1,\ldots,x_n) \Leftarrow var(p_{1k_1}) = sign(p_{1k_1}) \\ &val_2(x_1,\ldots,x_n) \Leftarrow var(p_{21}) = sign(p_{21}) \wedge val_1(x_1,\ldots,x_n) \\ &\vdots \\ &val_2(x_1,\ldots,x_n) \Leftarrow var(p_{2k_2}) = sign(p_{2k_2}) \wedge val_1(x_1,\ldots,x_n) \\ &\vdots \\ &val_m(x_1,\ldots,x_n) \Leftarrow var(p_{m1}) = sign(p_{m1}) \wedge val_{m-1}(x_1,\ldots,x_n) \\ &\vdots \\ &val_m(x_1,\ldots,x_n) \Leftarrow var(p_{mk_m}) = sign(p_{mk_m}) \wedge val_{m-1}(x_1,\ldots,x_n) \end{aligned}$$

Because $val_m(x_1, \ldots, x_n)$ corresponds to the truth value of $\phi(x_1, \ldots, x_n)$ when taking all m clauses into account, it is exactly the definition we need.

$$val_{\phi(x_1,\ldots,x_n)}(x_1, \ldots, x_n) \Leftarrow val_m(x_1, \ldots, x_n)$$

Again, every variable in the body also appears in the head. The number of rules is equal to the number of literals in conjunctive normal form, and each rule is linear in the number of variables. Thus, the resulting rules are polynomial in the size of the quantifier-free portion of the formula.

This completes the encoding. The time to produce the encoding is polynomial in the size of the original QBF formula, which completes the proof of PSPACE-hardness. □

Theorem 3 (With keywords FSL languages are polynomial). *Given a policy written in a FSL language and query of the form key*$(t1, \ldots, tn)$*, evaluating the query takes polynomial time.*

Proof. FSL 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. □