Managing the Network with Merlin

Robert Soulé Shrutarshi Basu Robert Kleinberg Emin Gün Sirer Nate Foster
Cornell University
Department of Computer Science
{soule,basus,rdk,egs,jnfoster}@cs.cornell.edu

ABSTRACT

This paper presents the Merlin network management framework. With Merlin, administrators express network policy using programs in a declarative language based on logical predicates and regular expressions. The Merlin compiler automatically partitions these programs into components that can be placed on a variety of devices including switches, middleboxes, and end hosts. It uses a constraint solver and parameterizable heuristics to allocate resources such as paths and bandwidth. To ease the administration of federated networks, Merlin provides mechanisms for delegating management of sub-policies to tenants, along with tools for verifying that delegated sub-policies do not violate global constraints. Overall, Merlin simplifies the task of network administration by providing high-level abstractions for directly specifying network policy.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network operating systems; D.3.2 [**Language Classifications**]: Specialized application languages

Keywords

Software-defined networking, program partitioning, delegation, verification, Merlin.

1. INTRODUCTION

To manage a network today, administrators must carefully configure different pieces of software written in multiple languages executing on a variety of machines. For example, a typical data center network might use end hosts to filter and rate-limit traffic based on packet headers, special-purpose middleboxes to inspect packet payloads for malicious patterns, and a software-defined networking (SDN) controller to install forwarding rules on switches. Network management is further complicated by the fact that networks are expected to accommodate the needs of tenants, who may require quality of service guarantees, specialized security policies, or custom allocations of resources for particular applications.

Consequently, network administration has become increasingly complicated and error-prone. In response, a number of recent projects have attempted to raise the level of abstraction for specifying network functionality, through the use of programmable switches [6, 17, 29], middleboxes [14, 26, 35, 36], or end hosts [7, 24, 38]. Although these approaches have made great strides towards easing the burden on network administrators [23], each offers only a partial solution to the general problem. Because they focus exclusively on a single component of the network, they do not allow administrators to program the network as a unified entity. As a result, there is a disconnect between the high-level properties that administrators want to guarantee (e.g., filter all traffic against a given pattern) and how that functionality is realized on multiple devices (e.g., configure routers and switches so that all traffic traverses at least one middlebox).

The vision advocated in this paper is to provide a unified framework for controlling all of the components in the network—a truly programmable network. As a first step toward this goal, we present the Merlin network management framework. With Merlin, administrators express policies in a high-level, declarative language. The Merlin compiler uses *program partitioning* to transform global policies into smaller sub-policies that are automatically distributed to the different components in the network for enforcement. Management of sub-policies may be *delegated* to tenants, who can modify them to reflect their own custom requirements. Merlin provides mechanisms for *verifying* that a sub-policy modified by a tenant conforms to global policy set down by the administrator. This allows federated networks to be managed within a single administrative framework.

The Merlin language is based on regular expressions (with operators for union, concatenation, and Kleene star)—a natural syntax for describing paths through a network. In addition, using regular expressions provides two important benefits. First, it facilitates encoding Merlin policies as con-

HotNets '13, November 21–22, 2013, College Park, MD, USA.

straint problems whose solutions can be used to determine the placement of the components that enforce policy or provide functionality. Second, it enables verifiable policy delegation as Merlin can determine if a sub-policy modified by a tenant conforms to the original.

Overall, Merlin simplifies network administration by providing high-level abstractions for specifying network-wide policies, freeing administrators from having to worry about low-level implementation details.

2. THE MERLIN LANGUAGE

The Merlin language provides programmers with a rich collection of constructs that allow them to specify the intended behavior of the network at a high-level of abstraction. The following example, which states that all TCP traffic between a given pair of hosts must pass first through a network address translator (`nat`), then a deep packet inspector (`dpi`), and abide by a rate limit of 100Mb/s, illustrates the main features of the language:

```
( ipSrc = 192.168.1.1 and
  ipDst = 192.168.1.2 and
  ipProto = 0x06 )
-> ( .* nat .* dpi .* )
at max(100Mb/s)
```

A Merlin policy is a collection of *statements* that specify the handling of a subset of network traffic. The policy grammar is defined in Figure 1. We assume that statements are disjoint and total—that is, no two statements specify the handling of overlapping sets of packets, and every packet is handled by some statement. In the Merlin implementation, these well-formedness conditions are guaranteed by a simple pre-processor. Each statement comprises several components: the left side of the arrow (`->`) is a *logical predicate* that describes a set of packets, while the right side of the arrow is a *regular expression* that specifies the forwarding paths and transformations which should be applied to matching packets and an optional *bandwidth constraint*.

Logical Predicates. Merlin supports a rich predicate language for classifying network traffic. An atomic predicate of the form $f = n$ denotes the set of packets whose header field f is equal to n. For example, in the policy above, the predicate matches packets with IP source address `192.168.1.1`, IP destination address `192.168.1.2`, and protocol TCP (`0x06`). Merlin provides atomic predicates for fields in a number of standard protocols including Ethernet, IP, TCP, and UDP, and a special field for matching packet payloads. Predicates can also be built up using standard boolean operators such as conjunction (`and`), disjunction (`or`), and negation (`!`).

Location quantifiers. Predicates may be optionally preceded by a location quantifier that determines how traffic from different sources is classified. To illustrate, consider

loc	$\in$ *Locations*	
t	$\in$ *Transformation Functions*	
pol	$::= (s_1; \ldots; s_n)$	Policy
s	$::= q\ p\ \rightarrow\ e$ at r	Statement
q	::= forall | exists	Quantifier
p	$::= m \mid p_1$ and $p_2 \mid p_1$ or $p_2 \mid !\, p_1$	Predicate
m	$::= f = n$	Match
e	$::= . \mid c \mid e\, e \mid e \mid e \mid e^* \mid !\, e$	Path Expression
c	$::= loc \mid t$	Path Character
r	::= max$(n)^?$ min$(n)^?$	Rate

Figure 1: Merlin policy language syntax.

the following policy, which forwards traffic from sources `h1` and `h2` to a destination `h3` with a bandwidth cap:

```
true -> (h1|h2) .* h3 at max(100Mb/s)
```

With the `forall` quantifier, the constraints are applied collectively to traffic from any source by dividing the bandwidth equally among all paths. Effectively, `h1` and `h2` would both be capped by 50Mb/s. With the `exists` quantifier, the constraints are applied to traffic along *some* single path, and all other traffic is dropped. So, either the path starting at `h1` would be capped at 100Mb/s, or the path starting at `h2` would be capped at 100Mb/s. If no quantifier is specified for a statement, the Merlin compiler implicitly treats the statement as being annotated with `exists`.

Regular expressions. Merlin enables programmers to specify forwarding paths for selected classes of network traffic using regular expressions. Regular expressions are a natural and well-studied mathematical formalism for describing paths through a graph (such as a finite state automaton or a network topology). However, rather than matching strings of characters as with ordinary regular expressions, Merlin regular expressions match sequences of network locations as well as names of transformations, as described below. The compiler is free to select any matching path to forward traffic as long as the other constraints expressed by the policy are satisfied. We assume that the set of locations is finite. As with POSIX regular expressions, the dot symbol (`.`) matches a single element of the set of all locations. Note that there is an implicit relationship between certain predicates and expressions. For example, if `h1` has IP address `192.168.1.1`, then the predicate `ipSrc = 192.168.1.1` implies that the associated expression must start at `h1`, even if the programmer has not explicitly written the expression that way.

Transformations. Merlin regular expressions may also contain names of packet-processing functions that transform the headers and contents of packets. Such transformations can be used to implement a variety of useful operations including deep packet inspection, network address translation, load balancing, traffic accelerators, caches, proxies, rate limiters, and many others. The Merlin compiler determines the loca-

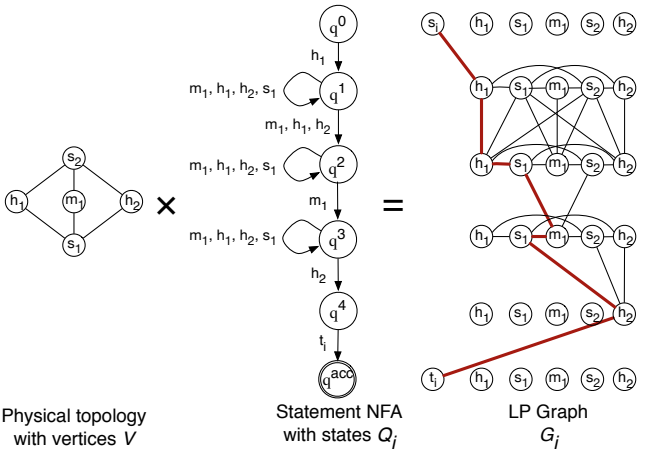

Figure 2: The NFA and graph encoding of the MIP formulation for the example policy. The thick, red path illustrates a solution.

tion where each transformation is enforced, using a mapping from transformation names to possible locations supplied as an auxiliary input. The only requirements on these transformations are: (i) they must take a single packet as input and generate zero or more packets as output, and (ii) they must only access local state. In particular, the restriction to local state ensures that the compiler can freely place transformations inside the network without having to worry about maintaining global state.

Bandwidth. Finally, Merlin policies may specify the network traffic rates with a bandwidth limit (`max`), a bandwidth guarantee (`min`), or both. By convention, policies without a rate clause are unconstrained—policies that lack a minimum rate are not guaranteed any bandwidth, and policies that lack a maximum rate may send traffic at rates up to line speed. Bandwidth limits and guarantees differ from transformations in one important aspect: they represent an explicit allocation of global network resources. Hence, special care is needed when compiling them.

3. COMPILATION

The Merlin compiler performs two main tasks: (i) it maps policies to constraint problems that can be solved using linear programming to determine forwarding paths, transformation placement, and bandwidth allocation; and (ii) it generates the low-level instructions needed to realize the policy using the switches, middleboxes, and end hosts available in the network.

Constraint Problem. In general, the Merlin compiler has substantial flexibility in determining which forwarding paths to use, where to place transformations, and how to allocate bandwidth. However, because Merlin offers a unified framework for programming the entire network, it must balance all of these concerns simultaneously. To do this, the Merlin compiler encodes the input policy and the network topology into a constraint problem whose solution, if it exists, can be used to determine the configuration of each network device.

Placement constraints. Recall that a statement in the Merlin language contains a regular expression which constrains the set of forwarding paths and transformations that may be used to satisfy the statement. We refer to these as *placement constraints*. To facilitate the search for routing paths that satisfy the placement constraints, the compiler represents them internally using a collection of directed graphs $\mathcal{G}_i$ whose paths correspond to paths in the physical network that respect the placement constraints for statement i.

The regular expression e_i in statement i is over the set of locations and transformations. The first step in the construction of $\mathcal{G}_i$ is to map e_i into a regular expression $\bar{e}_i$ over the set of locations (only) using a simple substitution: for every occurrence of a transformation, we substitute the union of all locations associated with that transformation. For example, if `h1`, `h2`, and `m1` can implement network address translation, then the expression `(.* nat .*)` would be transformed into `(.* (h1|h2|m1) .*)`. The next step is to transform the regular expression $\bar{e}_i$ into a non-deterministic finite automaton[1] (NFA), denoted $\mathcal{M}_i$, that accepts precisely the set of strings in the regular language given by $\bar{e}_i$.

Let L denote the set of locations in the physical network and $\mathcal{Q}_i$ denote the state set of $\mathcal{M}_i$. The vertex set of $\mathcal{G}_i$ is the Cartesian product $L \times \mathcal{Q}_i$ together with two special vertices $\{s_i, t_i\}$ that serve as the universal source and sink for paths representing statement i. There is an edge from (u, q) to (v, q') in $\mathcal{G}_i$ if and only if: (i) $u = v$ or (u, v) is an edge of the physical network, and (ii) (q, q') is a valid state transition of $\mathcal{M}_i$ when processing v. Similarly, there is an edge from s_i to (v, q') if and only if (q^0, q') is a valid state transition of $\mathcal{M}_i$ when processing v, where q^0 denotes the start state of $\mathcal{M}_i$. Finally, there is an edge from (u, q) to t_i if and only if q is an accepting state of $\mathcal{M}_i$. Paths in $\mathcal{G}_i$ correspond to paths in the physical network that satisfy the placement constraints of statement i, in a sense made precise by the following lemma.

LEMMA 1. *A sequence of locations $u_1, \ldots, u_k$ satisfies the placement constraint given by $\bar{e}_i$ if and only if $\mathcal{G}_i$ contains a path of the form $s_i, (u_1, q_1), (u_2, q_2), \ldots, (u_k, q_k), t_i$ for some state sequence $q_1, \ldots, q_k$.*

Figure 2 illustrates the construction of the graph $\mathcal{G}_i$ for a statement with the regular expression

```
h1 .* nat .* dpi .* h2
```

in the example network illustrated in the left-hand side of Figure 2. We assume that network address translation (`nat`) can be implemented at `h1`, `h2`, or `m1`, whereas deep packet

[1]For technical reasons, we require a NFA without ϵ-moves. See Hopcroft and Ullman's textbook [21] for a discussion of ϵ-moves and techniques for eliminating them.

inspection (dpi) can only be implemented at m1. Paths matching the regular expression can be "lifted" to paths in $\mathcal{G}_i$; the thick, red path in the figure illustrates one such lifting. Notice that the physical network also contains other paths such as h1, s1, h2, which do not match the regular expression. These do not lift to any path in $\mathcal{G}_i$. For instance, focusing attention on the rows of nodes corresponding to states q^2 and q^3 of the NFA, one sees that all edges between these two rows lead into node (m1,q^3). This, in turn, means that any path that avoids m1 in the physical network cannot be lifted to an s_i-t_i path in the graph $\mathcal{G}_i$.

Path selection. Next, given a set of bandwidth demands and specified link capacities, the compiler finds a satisfying assignment of paths while respecting the placement constraints. The problem bears a similarity to the well-known *multi-commodity flow problem* [1], with two additional types of constraints: (i) *integrality constraints* demand that only one path may be selected for each statement, and (ii) *placement constraints* are specified by regular expressions, as discussed above. To incorporate placement constraints, we formulate the problem in the graph $\mathcal{G} = \bigcup_i \mathcal{G}_i$ described above, rather than in the physical network itself. Incorporating integrality constraints into multi-commodity flow problems renders them NP-complete in the worst case, but a number of approaches have been developed over the years for surmounting this problem, ranging from approximation algorithms [8, 10, 12, 25, 27], to the use of mixed integer programming [4]. Our current implementation adopts the latter technique.

Our mixed integer program (MIP) has one $\{0,1\}$-valued decision variable x_e for each edge e of $\mathcal{G}$. Selecting a route for each statement corresponds to selecting a path from s_i to t_i for each i and setting $x_e = 1$ on the edges of those paths, $x_e = 0$ on all other edges of $\mathcal{G}$. These variables must satisfy the flow conservation equations

$$\forall v \in \mathcal{G} \sum_{e \in \delta^+(v)} x_e - \sum_{e \in \delta^-(v)} x_e = \begin{cases} 1 & \text{if } v = s_i \\ -1 & \text{if } v = t_i \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $\delta^+(v)$, $\delta^-(v)$ denote the sets of edges exiting and entering v, respectively. To express the capacity constraint for a link (u, v) in the physical network, let c_{uv} denote its capacity and let $E_i(u, v)$ denote the set of all edges of the form $((u, q), (v, q'))$ or $((v, q), (u, q'))$ in $\mathcal{G}_i$. The capacity constraints are given by

$$\forall (u, v) \sum_i \sum_{e \in E_i(u,v)} r^i_{\min} x_e \leq c_{uv} \quad (2)$$

where $r^i_{\min}$ denotes the minimum amount of bandwidth guaranteed in the rate clause of statement i.

The compiler allows the user to define an optimization criterion that determines which route assignments should be preferred among those satisfying the placement and bandwidth constraints. The system currently implements three such criteria: (a) *weighted shortest path* minimizes the total number of network hops in the assigned routes, weighted by the amount of guaranteed bandwidth. The user may wish to use this criterion when the goal is to minimize latency, since paths with a greater number of network hops tend to experience increased latency; (b) *min-max ratio* minimizes the maximum fraction of capacity reserved on any link (i.e., $r_{\max}$). This criterion is appropriate when the goal is to balance load across the network links; (c) *min-max reserved* minimizes the maximum amount of bandwidth reserved on any link (i.e., $R_{\max}$). This criterion may be sensible when the goal is to protect against failures, since it limits the amount of traffic that may disrupted by a single link failure.

Code Generation. After the Merlin compiler has determined a placement for the system components, it generates the appropriate code for the specific enforcement mechanism. Below, we briefly identify enforcement strategies for each target type, using readily available mechanisms:

- *Switches.* For basic forwarding, Merlin generates instructions for OpenFlow [29] switches and controllers that install forwarding rules and configure port queues with bandwidth guarantees.
- *Middleboxes.* Richer functions can be implemented by replicating virtual machines [18, 35], or by installing software for middlebox platforms. Merlin currently generates Click [26] modules.
- *End-hosts.* Functionality such as traffic filtering or rate limiting is implemented with standard Linux utilities such as iptables and tc.

Merlin can be configured to use the specific enforcement mechanisms available in a particular network. While the expressiveness of policies is bounded by the capabilities of the available devices, Merlin provides extensible mechanisms for generating target code, and a common interface for programming heterogeneous devices.

4. POLICY TRANSFORMATIONS

One of the main advantages of using a high-level language to specify network functionality is that the same abstractions that allow policies to be mapped to constraint problems (i.e., predicates, regular expressions, and explicit bandwidth reservations and caps), can also be exploited to support transformations on policies, including delegation and distribution. Moreover, the correctness of these transformations can be verified automatically.

Delegation. In many networks, administrators would like to be able to delegate control of a sub-network to tenants. Such a sub-network might be specified in terms of a set of nodes in the physical topology, or a set of packets with specified header fields. At the same time, tenants would often like to impose custom policy requirements. For example, a customer in a cloud data center might want to ensure that every

packet is scanned by a deep packet inspection engine. Or, the customer might want to be able to manage the bandwidth allocated to individual virtual machines—e.g., the web server might get a larger share of the customer's aggregate bandwidth than other machines.

Merlin allows administrators to delegate policies to tenants who may then *refine* them. To delegate a policy, the administrator first intersects the predicates and paths to restrict it to a particular sub-network. For example, suppose the global policy is as follows:

```
( ipSrc = 192.168.1.1 and
  ipProto = 0x06)
-> .* h3 at max(100Mb/s)
```

The tenant could then customize this policy for their particular needs. For example, they might wish to require web traffic to traverse a monitoring middlebox, and allocate additional bandwidth to web and `ssh` traffic:

```
( ipSrc = 192.168.1.1 and
  ipProto = 0x06 and tcpDst = 80)
-> .* dpi .* h3 at max(50Mb/s)

( ipSrc = 192.168.1.1 and
  ipProto = 0x06 and tcpDst = 22)
-> .* dpi .* h3  at max(25Mb/s)

( ipSrc = 192.168.1.1 and
  ipProto = 0x06 and
  !(tcpDst = 22 | tcpDst = 80))
-> .* dpi .* h3  at max(25Mb/s)
```

The tenant could then return this modified policy to the administrator, who would integrate it into the global policy.

Verification. In general, allowing tenants to modify policies would not be safe. For example, a tenant could lift restrictions on forwarding paths, eliminate transformations, or allocate more bandwidth to their own traffic—all violations of the global policy set down by the administrator. The Merlin language is designed to support checking policies for inclusion, which can be used to establish the correctness of policy transformations implemented by untrusted tenants.

Intuitively, a valid refinement of a policy is one that makes it only more restrictive. To verify that a policy modified by a tenant is a valid refinement of the original, one simply has to check that for every statement in the original policy, (i) the set of paths allowed for matching packets in the refined policy is included in the set of paths in the original, and (ii) the bandwidth constraints in the refined policy imply the bandwidth constraints in the original. To decide these conditions, Merlin uses a simple algorithm that performs a pair-wise comparison of all statements in the original and modified policies, (i) checking for language inclusion [21] between the regular expressions in statements with overlapping predicates, and (ii) checking that the sum of the bandwidth constraints in all overlapping predicates implies the original constraint.

Distribution. Another use of delegation in Merlin is for obtaining efficient and scalable enforcement of network-wide policies. As an example, consider a policy that specifies an aggregate bandwidth cap on a pair of hosts managed by different tenants:

```
forall
( ipSrc = 192.168.1.1 or
  ipSrc = 192.168.1.2 )
 -> .* at max(100Mb/s)
```

Enforcing this policy requires maintaining packet counts of traffic from both hosts. An easy option is to use a middlebox that can accurately keep track of the bandwidth usage for both hosts. But a centralized enforcement mechanism can become a bottleneck at sufficiently high traffic loads. On the other hand, distributing enforcement to end-hosts can reduce contention for network devices [13]. To enforce the cap on end-hosts, the Merlin compiler partitions the overall bandwidth allocation to each host—otherwise the individual hosts would be able to generate traffic that exceeds the aggregate. By default, the compiler transforms this policy into one where the location quantifier has been eliminated, and the bandwidth has been distributed evenly between the hosts, as described in Section 2:

```
( ipSrc = 192.168.1.1 )
-> .* at max(50Mb/s)

( ipSrc = 192.168.1.2 )
-> .* at max(50Mb/s)
```

Of course, an equal split may not be the best allocation of bandwidth. For example, allocating more bandwidth to the second tenant may be better:

```
( ipSrc = 192.168.1.1)
-> .* at max(80Mb/s)

( ipSrc = 192.168.1.2)
-> .* at max(20Mb/s)
```

Alternatively, the tenants could cooperate to do fine-grained allocations of bandwidth in response to current traffic conditions, as in systems such as EyeQ [24]. Using Merlin's delegation mechanisms, the tenants can use any strategy of their choosing, as long as the final policies generated by the two tenants are valid refinements of the original policy.

By delegating functionality to tenants, Merlin relocates enforcement of network-wide properties to tenants, and obviates the need for the compiler to maintain distributed state. However, distributed enforcement involves an inherent trade-off: it increases scalability, but risks under utilizing resources if the static allocations do not reflect actual usage. Merlin provides a framework that allows tenants to manage this trade-off themselves, while giving the administrator a means to verify that the distributed policies conform to the original.

5. EXAMPLES

To illustrate the expressiveness of the Merlin language, we present several examples based on realistic network policies.

MapReduce guarantees. MapReduce [11] uses a many-to-many communication pattern during its *shuffle* phase that results in heavy network load. Consequently, MapReduce is sensitive to background traffic in data centers, especially with protocols such as UDP that lack congestion control [5, 24]. Merlin can guarantee a minimum quality of service for MapReduce traffic:

```
( ipSrc = 192.168.1.1/16 and
  ipDst = 192.168.1.1/16 and
  ipProto = 0x06 and tcpDst = 50060 )
-> .* at min(100Mb/s)
```

IP multicast control. Monitoring applications often rely on IP multicast to transmit updates. But switches can only store a limited number of multicast addresses, and must often resort to flooding multicast packets. Instead, the administrator could maintain *multicast groups*—sets of multicast addresses with common subscribers—and compress each group to a single address [39]. Merlin can slot multicast group traffic through such a compression function and enforce a rate limit:

```
forall
( ipDst = 224.0.0.1 or
  ipDst = 224.0.0.2 )
  -> compress .* at max(10Gb/s)
```

Resource isolation. Many regulatory guidelines require corporations to keep different parts of their businesses completely separate. For example, the Sarbanes-Oxley Act requires that the investment side of banks be kept completely separate from the brokerage side of the company. Regulations in healthcare such as HIPAA are similar. Hence, corporate networks must often ensure that traffic from different portions of the network never traverse the same middlebox.

```
forall ( ipSrc = 192.168.1.1/8 )
-> .* m1 .*

forall (!ipSrc = 192.168.1.1/8 )
-> !(.* m1 .*)
```

Defense in depth. A common security practice, known as defense in depth, constructs trustworthy systems by layering less trustworthy components. The following policy captures this approach by routing all traffic through two diverse firewall implementations, without enforcing a particular ordering:

```
forall true
-> ( .* fire1 .* fire2 .*
   | .* fire2 .* fire1 .* )
```

In summary, Merlin allows administrators to specify a diverse set of policies that can enforce bandwidth caps, provide bandwidth guarantees, indicate packet transformations, and dictate forwarding paths.

6. RELATED WORK

A number of systems in recent years have proposed mechanisms for specifying and implementing advanced functionality in a network such as bandwidth caps and guarantees [3, 32, 37], traffic filters [22, 34], or forwarding policies [19, 20, 30]. Merlin builds on these systems by providing a unified interface for controlling many different kinds of devices including switches, middleboxes, and end hosts.

Several other systems have explored programming abstractions for switches [6, 17, 29], middleboxes [14, 26, 35, 36], and end hosts [7, 24, 38]. Notably, languages such as Frenetic [17, 16], Pyretic [31], and Maple [40], provide constructs for programming OpenFlow switches. In particular. Merlin's syntax based on predicates and regular expressions resembles constructs in Frenetic and NetKAT [17, 2]. Compared to these languages Merlin is more general—in addition to configuring switches, it allows administrators to specify middlebox functionality, manage end hosts, control allocation of bandwidth, and delegate policies.

The SIMPLE system [33] uses a linear programming formulation to load balance the network with respect to TCAM space and CPU usage, but does not specify the programming interface to the framework, nor how policies are represented and analyzed. The PANE system [15] allows end hosts to make explicit requests for network resources like bandwidth, but does not provide mechanisms for partitioning functionality across a variety of devices.

Program partitioning has been used in other contexts, including web applications and distributed computing [9, 28].

7. OUTLOOK

In ongoing work, we are investigating how to extend Merlin's forwarding policies to support compositional syntax, as well as new primitives that can encode features such as multicast. We are also investigating uses of authentication logic with Merlin policies in the context of delegation.

Overall, Merlin allows administrators to specify the functionality of an entire network at a suitably high level of abstraction, leaving the low-level details related to the configuration of individual components to the compiler. At the same time, Merlin provides tenants with the freedom to tailor policies to their particular needs, while assuring administrators that the constraints expressed by global policies are enforced. This approach significantly simplifies network administration, and provides a foundation for a variety of future research on network programmability.

Acknowledgments

The authors wish to thank Andrew Ferguson, Sean Ogden, Robbert van Renesse and the HotNets reviewers for helpful comments on earlier drafts of this paper. Our work is supported by NSF grant CNS-1111698, ONR award N00014-12-1-0757, AFOSR grant FA9550-09-1-0100, a Sloan Research Fellowship, and a Google Research Award.

8. REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *POPL*, 2014. To appear.

[3] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, pages 242–253, Aug. 2011.

[4] C. Barnhart, C. A. Hane, and P. H. Vance. Using Branch-and-Price-and-Cut to Solve Origin-Destination Integer Multicommodity Flow Problems. *Operations Research*, 48(2):318–326, Mar. 2000.

[5] D. Bonfiglio, M. Mellia, M. Meo, and D. Rossi. Detailed Analysis of Skype Traffic. *TOMM*, 11(1):117–127, Jan. 2009.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *SIGCOMM*, pages 1–12, Aug. 2007.

[7] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *HotSDN*, pages 85–90, Aug. 2012.

[8] A. Chakrabarti, C. Chekuri, A. Gupta, and A. Kumar. Approximation Algorithms for the Unsplittable Flow Problem. In *APPROX*, pages 51–66, Sept. 2002.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure Web Applications via Automatic Partitioning. In *SOSP*, pages 31–44, Oct. 2007.

[10] J. Chuzhoy and S. Li. A Polylogarithmic Approximation Algorithm for Edge-Disjoint Paths with Congestion 2. In *FOCS*, pages 233–242, Oct. 2012.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, Dec. 2004.

[12] Y. Dinitz, N. Garg, and M. X. Goemans. On the Single-Source Unsplittable Flow Problem. *Combinatorica*, 19(1):17–41, Jan. 1999.

[13] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: A Scalable Fault Tolerant Network Manager. In *NSDI*, pages 7–21, Mar. 2011.

[14] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *SOSP*, pages 15–28, Oct. 2009.

[15] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*, pages 327–338, Aug. 2013.

[16] N. Foster, A. Guha, et al. The Frenetic Network Controller. In *The OCaml Users and Developers Workshop*, Sept. 2013.

[17] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, pages 279–291, Sept. 2011.

[18] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward Software-Defined Middlebox Networking. In *HotNets*, pages 7–12, Oct. 2012.

[19] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica. Pathlet Routing. *SIGCOMM CCR*, 39(4):111–122, Aug. 2009.

[20] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker. Practical Declarative Network Management. In *WREN*, pages 1–10, 2009.

[21] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[22] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a Distributed Firewall. In *CCS*, pages 190–199, Nov. 2000.

[23] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally Deployed Software Defined WAN. In *SIGCOMM*, pages 3–14, Aug. 2013.

[24] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, pages 297–312, Apr. 2013.

[25] J. M. Kleinberg. Single-Source Unsplittable Flow. In *FOCS*, pages 68–77, Oct. 1996.

[26] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *TOCS*, 18(3):263–297, Aug. 2000.

[27] S. G. Kolliopoulos and C. Stein. Approximation Algorithms for Single-Source Unsplittable Flow. *SIAM J. Comput.*, 31(3):919–946, June 2001.

[28] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A Platform for Secure Sistributed Computation and Storage. In *SIGOPS EW*, pages 321–334, Oct. 2009.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 38(2):69–74, Mar. 2008.

[30] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *POPL*, pages 217–230, Jan. 2012.

[31] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *NSDI*, pages 1–13, Apr. 2013.

[32] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, pages 187–198, Aug. 2012.

[33] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*, pages 27–38, Aug. 2013.

[34] M. Roesch. Snort—Lightweight Intrusion Detection for Networks. In *LISA*, pages 229–238, Nov. 1999.

[35] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, pages 24–38, Apr. 2012.

[36] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*, pages 13–24, Aug. 2012.

[37] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *HotCloud*, pages 1–8, June 2010.

[38] A. Shieh, S. Kandula, and E. G. Sirer. SideCar: Building Programmable Datacenter Networks Without Programmable Switches. In *HotNets*, pages 21–27, Oct. 2010.

[39] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, R. Burgess, G. Chockler, H. Li, and Y. Tock. Dr. Multicast: Rx for Data Center Communication Scalability. In *EuroSys*, pages 349–362, Apr. 2010.

[40] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*, pages 87–98, Aug. 2013.