Intentional Network Monitoring: Finding the Needle without Capturing the Haystack

Sean Donovan, Nick Feamster
Georgia Institute of Technology

Abstract

Monitoring network traffic serves many purposes, from security to accounting, yet current mechanisms for collecting network traffic are typically based on low-level features of network traffic (*e.g.*, IP addresses, port numbers), rather than characteristics that more closely map to intent (*e.g.*, people, applications, or devices). In this paper, we present the case for intentional network monitoring—the practice of capturing the minimal set of traffic that satisfies the operator's monitoring intent or goal—and a preliminary design and implementation for NetAssay, a system that enables intentional monitoring. A significant challenge in developing NetAssay is developing a runtime that can maintain a mapping between stable abstractions that an operator or programmer might use to express intent (*e.g.*, a username) and the dynamic, heterogeneous data that establishes these associations (*e.g.*, information from a login server or DNS record). We present examples that show how the NetAssay runtime can perform late binding between these mappings and network flow space and discuss the research and technical challenges associated with establishing more general late-binding mechanisms.

Categories and Subject Descriptors: C.2.3 [Computer-Communication Networks] *Network Operations*: Network Monitoring

General Terms: Algorithms; Design; Measurement

Keywords: Software-defined networking (SDN); network monitoring

1 Introduction

Network operators need better ways to express the traffic they are interested in capturing at a higher level of abstraction. To address this need, we introduce a concept called *intentional network monitoring*, which allows network operators to monitor subsets of traffic of higher-level principals, such as who or what the traffic is associated with, in service of a particular *intent*. Much as previous work on intentional naming allows end systems to route traffic according to intent (*e.g.*, "send this document to the nearest printer") [1], intentional monitoring allows operators to monitor traffic according to similar goals (*e.g.*, "monitor all voice traffic from this user"). Network operators should be able to express an intentional specification to capture network traffic without regard to the lower-level details concerning how or where that traffic is captured.

Recently exposed network surveillance and data collection activities highlight the potentially far-reaching consequences of our current inadequate network monitoring capabilities. Consider a search warrant that permits monitoring of all voice traffic of a particular citizen. In the case of conventional telephony, executing the warrant entails "wiretapping" a phone line, an operation that captures no more and no less than the warrant specifies. Today, however, a user might make a voice call using a variety of applications and devices, ranging from video chats on laptops, phones, and tablets to phone calls through VoIP switching devices (*e.g.*, Vonage). A warrant might also provide authority to monitor other forms of communication, such as online chats or emails. In addition to coping with the increasing diversity of communications, monitoring infrastructure needs to cope with dynamism: for example, a typical user is mobile and may initiate such a call using different upstream Internet service providers in different locations (*e.g.*, home, work, coffee shops, hotels). More broadly, network monitoring and management can in general benefit from the ability to perform intentional monitoring. For example, a network billing system could associate packet or byte counts to specific applications and users, enabling much more fine-grained accounting. A network monitoring system could express interest in traffic from a particular user, application, or device. Unfortunately, today's mechanisms provide no way to perform monitoring that precisely satisfies intent; as a result, finding the "needle" often requires capturing the "haystack", compromising user privacy.

In this paper, we present the case for intentional network monitoring and the preliminary design and implementation of NetAssay, a system that enables some forms of intentional network monitoring. The primary challenge in implementing NetAssay is developing a runtime that supports stable

HotNets-XIII October 27 - 28 2014, Los Angeles, CA, USA

http://dx.doi.org/10.1145/2670518.2673872

abstractions based on a variety of external data that is continuously changing. The first aspect of this challenge involves *dynamism*: Expressing traffic in terms of users, applications, or services does not map to static flow space: users move, they change the devices that they use, networks are renumbered, routes change, and so forth. This dynamism requires implementing a *late binding* mechanism that remaps higher-level abstractions to portions of flow space in real-time, as these associations change and mechanisms for efficiently performing incremental recomputation of policies and recompilation of flow table entries.

A second challenge involves *heterogeneity*: Establishing mappings between higher-level entities and rules in flow space requires incorporating associations from potentially many data sources, including (but not limited to) DNS, BGP, RADIUS, and 802.1X. In some cases, maintaining these mappings requires on-path monitoring of existing traffic flows. An additional challenge thus involves developing scalable and transparent mechanisms for establishing and tracking these mappings, efficiently and with minimal disruption to existing network traffic.

Inspired by earlier work on intentional naming [1], which combines name resolution with routing, NetAssay controls routing to establish and adapt these mappings in real-time. For example, mapping a domain name to flow space can be done by diverting a copy of the DNS responses that establish these mappings to the control runtime that ultimately "compiles" higher-level principals to flow space. The runtime thus exploits its control over network traffic to transparently establish these mappings, without the programmer's knowledge of how these mappings are established. Currently, NetAssay allows network operators to express the intent to monitor based on the domain names and autonomous systems. In principle, NetAssay allows network operators to express intent for *any* higher-level primitive that can be mapped to flow space. A significant research challenge involves developing runtime mechanisms to cope with the heterogeneous sets of dynamic data sources that establish real-time associations between higher-level entities and flow space.

The rest of the paper is organized as follows. Section 2 describes related work in intentional naming, network monitoring, and SDN. Section 3 presents the case for intentional network monitoring by presenting several examples of monitoring tasks that are difficult to do with today's technology. Section 4 presents the design and preliminary implementation of NetAssay, a system for intentional network monitoring. We conclude in Section 5 with a research agenda.

2 Related Work

Early work on intentional naming in networks [1] inspired some design aspects of NetAssay. The Intentional Naming System (INS) [1] allows systems to route traffic to network endpoints based on intent (*i.e.*, printing to the closest printer) as opposed to a more ephemeral network identifier such as an IP address (or even a domain name). NetAssay applies a similar concept to network monitoring.

NetAssay's runtime must support dynamic updates to policies as network conditions (*e.g.*, the set of IP prefixes corresponding to a particular AS, the set of authorized hosts). Supporting this level of dynamism at scale is part of our ongoing work, but previous work on functional reactive network control [15] and dynamic specialization [6] may ultimately prove useful in helping NetAssay scale in light of dynamic network conditions.

NetAssay's programmatic interface is inspired by Pyretic [13] and NetKAT [2], which use the notion of a *virtual packet header* to allow a programmer to express policies in terms of queries on network traffic. A virtual packet header includes all of a packet's header fields, as well as metadata associated with that packet (*e.g.*, the switch where the packet is located). NetAssay extends the notion of a virtual packet header to support additional metadata that is associated with higher-level network principals, such as a user, device, or application. The NetAssay abstractions and runtime extend Pyretic to support queries based on a broader range of metadata; currently, the NetAssay extensions are wrappers; we are working to incorporate many of these functions into the native language.

NetAssay uses an SDN controller to manipulate and monitor traffic flows; in this regard, NetAssay relates to other systems that have used SDN to facilitate monitoring. OpenSAFE [3] helps a network operator monitor traffic without affecting the performance of other production traffic, but its abstractions focus on routing traffic through existing monitoring devices, as opposed to developing abstractions for monitoring. Ethane [5] provides some support for monitoring, but the monitoring it performs is focused on authorization in enterprise networks; it does not provide a general framework for monitoring arbitrary sets of traffic flows. OpenSketch [16] provides a mechanism for performing specific types of measurements in switch hardware. In contrast to NetAssay, OpenSketch aims to provide a software interface to switch hardware for performing specific types of measurement (*e.g.*, hashing and counting); it does not operate on higher-level network entities such as user names, network names, devices, or domains.

Several other applications have developed security-inspired applications for network monitoring. CloudWatcher [14], an application developed for NOX [7], requires network devices to register device IDs and types with the controller and routes packets corresponding to corresponding traffic flows through the appropriate network devices to facilitate monitoring. Like OpenSAFE, CloudWatcher focuses on routing traffic through monitoring devices, not on creating and maintaining a mapping between higher-level abstractions and flow-table entries that perform the monitoring. NetAssay focuses on mapping higher-level monitoring abstractions (*e.g.*, sets of devices corresponding to a user) to constructs that can be installed on switches (*i.e.*, flow table entries); in this sense, NetAssay is

complementary to these efforts and can ultimately enable the types of rerouting that these systems rely on.

3 The Case for Intentional Network Monitoring

We present a case for intentional network monitoring by offering examples of network monitoring tasks that are difficult to implement using today's network infrastructure. Today's primary modes of expressing network monitoring tasks involve packet filters (as in pcap), flow statistics (as in NetFlow), or interface byte or packet counters (as in SNMP). Unfortunately, these expressions involve low-level network primitives (*e.g.*, switch port, IP address). Network operators should rather be able to express network monitoring queries based on higher level questions such as a user, application, or device that is initiating or receiving the traffic, the network or domain with the traffic is associated, or perhaps more sophisticated characteristics such as where the traffic has been or where it is going. The rest of this section expands on these possibilities with possible use cases for intentional network monitoring.

Monitoring or billing usage based on service or application. Suppose that an Internet service provider (ISP) wants to bill a customer based on the volume of streaming video traffic that the user sends to a particular service such as Netflix. Implementing such a policy is rather difficult to do today: it involves establishing a mapping between a service called "Netflix" and the corresponding parts of flow space.

Simply mapping a domain name such `netflix.com` to the IP addresses corresponding to its DNS name may not be sufficient because much of the content may be hosted on a content distribution network (CDN), whose IP addresses are shared with other services, and whose domain names may not actually be the higher-level domain name but instead that of the CDN. Furthermore, not all traffic to a domain may be associated with the application in question, so a query for streaming traffic may also need to establish associations with flows based on network application identification. Performing this type of monitoring today is simply not possible. Although it is possible to establish certain mappings between applications and flow space, and between CDNs and applications [4], these mappings continually change, making it difficult to integrate them into a monitoring system with higher-level programmatic abstractions.

Monitoring traffic based on network paths and properties. Suppose that a network operator wants to subject all traffic that originates from a particular autonomous system (AS) to more thorough monitoring. For example, an AS may have a reputation of hosting malware [9]; a network operator may wish to capture all traffic that originates from a certain AS for further inspection (*e.g.*, by an intrusion detection system). In this case, intentional monitoring must support an abstraction that maps AS numbers to regions of flow space (in this case, IP prefixes).

In addition to queries based on AS numbers an operator might also wish to write a query based on a dynamic list of "malicious" ASes, such as the one that Hostexploit maintains [9]. The runtime would then either need to (1) proactively update the both its own mapping and existing flow-table entries every time the reputation list was updated or (2) reactively query such a reputation list each time a new traffic flow arrived. Either of these approaches has performance tradeoffs that deserve further attention. We envision a runtime that could ultimately support monitoring based on more complex policies, such as the path that the traffic is taking through the network. Recent work on path queries [12] may help support for this type of monitoring.

Monitoring traffic based on user or device type. Suppose that a network operator or law enforcement agency wants to monitor the traffic for a particular user or device. For example, the operator of a corporate enterprise network might want to monitor all chat traffic from a particular employee to assess flight risk (from our discussions with network operators at a major financial institution, this is a common application of network monitoring). Unfortunately, capturing the traffic flows that are specific to this task are difficult, as a user may have multiple devices on the network, and the IP addresses associated with those devices may also change over time, particularly from day-to-day and as a user moves around the network.

Authentication and directory systems such as LDAP [17] and 802.1X [10] associate MAC addresses and IP addresses with user names, and device fingerprinting techniques may ultimately be able to create mappings between device MAC addresses and more meaningful characterizations of a device (*e.g.*, phone, tablet, laptop), but these mappings have typically not been integrated into a general network monitoring framework, to allow operators to extract traffic flows for a specific user name or user device. Ultimately, the runtime of a network monitoring system could enable real-time mapping of network monitoring queries based on these entities to the appropriate MAC addresses, which could subsequently be translated into flow rules.

4 NetAssay

We first present our design goals and the types of programming interfaces that we aim to expose to network operators and programmers. We then describe a prototype implementation of NetAssay, which realizes a preliminary runtime system that supports and maintains these mappings.

4.1 Design Goals

We provide a brief overview of the design goals of NetAssay and how the design attempts to achieves these goals.

- **Intuitive and intent-based.** Operators should be able to express policies that correspond to the goals they are trying to achieve in an intent-based, high-level language (*e.g.*, "capture all streaming video traffic to Netflix", "capture all

Web traffic from Alice's iPad") rather than having to write filters in terms of low-level protocol headers.

- **Dynamic.** Programming abstractions should remain stable, even as certain aspects of the network state change. For example, a network operator might want to capture traffic from certain ports for hosts that an intrusion detection system deems to be infected. In this case, the operator should be able to write a traffic capture expression based on "infected hosts", without having to worry about whether the IP addresses in the set of infected hosts might change.
- **Extensible.** Programmers and network operators may want to perform network monitoring tasks based on higher-level attributes that are difficult to predict in advance. We can predict that operators may want to express network monitoring polices based on certain attributes (*e.g.*, devices corresponding to a particular user, application, or device), but there may be other traffic attributes that depend on the network's design (*e.g.*, a network that has an authentication service might wish to express queries in terms of authenticated hosts). Ultimately, NetAssay must provide an interface that allows programmers and operators to extend the programming abstraction with new, custom attributes.
- **Fast.** Certain mappings in NetAssay may require monitoring network traffic in real-time (*e.g.*, the mapping between a domain name and flow space may require monitoring DNS responses). When mappings between abstractions and flow space require monitoring traffic on-path, NetAssay should do so in such a way that does not degrade network performance.
- **Accurate.** NetAssay's runtime must accurately map higher-level programming constructs to the corresponding portions of flow space. For example, a query for a user's streaming traffic should yield all of the corresponding flows, without additional extraneous flows. Achieving this goal scalably and in response to changing network behavior may be difficult in practice, particularly given switch table sizes and update rates.

4.2 Programming Abstractions

We design NetAssay's programming abstractions by extending the Pyretic language and runtime [13]. Pyretic allows a network operator to write a network control program as if a logically centralized SDN controller were seeing every packet in the network. The programming abstraction is that operators write queries on "located packets", which are packets with additional virtual packet header fields that specify the switch and port where the packet is currently located. For example,

```
match(srcport=80)
```

returns all packets whose source port is 80, and

```
match(srcport=80,dstip='192.0.2.28')
```

returns only packets with source port 80 whose destination IP address is `192.0.2.28`. NetAssay's extension to this is conceptually simple: the system aims to allow a programmer to express match statements that are based on a broader array of metadata, such as those from Section 3.

Ultimately, NetAssay should support expression of the following policy, which would return only packets that were originating from or destined to `example.com`:

```
match(domain='example.com')
```

Extending Pyretic's syntax is straightforward enough: Pyretic allows programmers to specify values for certain "virtual" packet headers such as the location of a packet within a network or the switch port where the packet is located (or where it is to be forwarded). Ultimately, Pyretic views packets as key-value dictionaries, where there is a key-value pair for each header field in the packet. Because Pyretic already provides clean interfaces for querying and writing policies on packets with virtual headers, we can develop a query language for NetAssay simply by extending the sets of keys that a programmer can use in a Pyretic query.[1]

The extended set of virtual packet headers that we have described above are challenging to implement because, unlike the rest of Pyretic's virtual packet headers, those virtual headers that NetAssay must support do not map naturally to switch hardware. In Pyretic, the keys that map to network headers that switches understand must be translated into flow table entries. If the virtual header does not directly map to flow table entries in hardware, software must translate these headers to a set of rules that can be represented in hardware. NetAssay's runtime must thus establish and maintains a mapping between NetAssay's higher-level virtual packet headers and the portions of flow space that correspond to those higher-level descriptions. Establishing such a mapping—and maintaining it as network conditions change—is challenging. We explain these challenges and our initial solutions in the next section.

4.3 Runtime

NetAssay's programming interface requires a runtime that translates higher-level metadata about traffic flows to lower-level rules that correspond to flow space. In this section, we describe the NetAssay runtime components; we then explain how NetAssay currently establishes and maintains mappings between certain abstractions and flow space and the research challenges associated with maintaining these mappings in general.

4.3.1 System components

The NetAssay assay runtime has two components. The first component is a *control application*, which a network operator writes in a Pyretic-style language that we have augmented to support NetAssay's extensions to support metadata. Second, the *metadata engines* (MEs) translate a NetAssay policy into the equivalent Pyretic policy and update these policies dynamically as conditions change. Different mappings will

[1] The current implementation of NetAssay operates using syntax that is slightly less well-integrated with Pyretic: `matchURL('example.com')`, but we are currently modifying the implementation so that it operates within Pyretic's current syntax.

NetAssay policy	`match(authusers='true', domain='example.com')`
Pyretic policy	`(match(srcip=203.0.113.42) + match(srcip=203.0.113.43) + match(srcip=203.0.113.44)) >> (match(dstip=98.41.100.2) + match(dstip=98.41.100.3) + match(dstip=98.41.100.4) + match(srcip=98.41.100.5))`

Table 1: *Translating a NetAssay policy into a Pyretic policy. NetAssay policies effectively involve metadata that specify set membership, so anytime set membership (*e.g.*, in* `authusers`*) changes, the NetAssay runtime must generate a new Pyretic policy.*

require different MEs. For example, a NetAssay query that says `match(authusers=true)` requires a corresponding ME that returns a Pyretic policy that can be mapped to flow space, such as `match(srcip=203.0.113.43) + match(srcip=203.0.113.44) + ...` depending on the set of IP addresses (or some other flow space characteristic, such as host MAC addresses) that correspond to the set of flows in `authusers`.

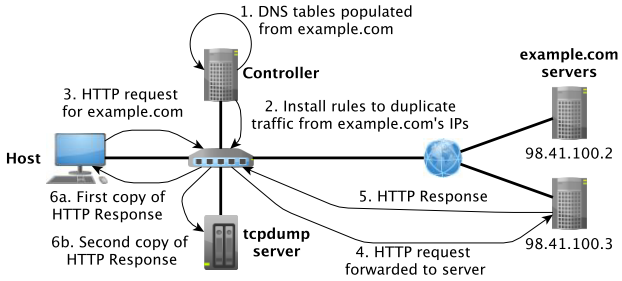
Table 1 shows an example of a NetAssay policy that combines input from two different MEs: one that handles `authusers`, and one that handles mapping to a specific `domain`. The corresponding Pyretic policy, which NetAssay generates, shows how the runtime must both invoke multiple metadata engines to determine which parts of flow space correspond to the query and use Pyretic's composition operators to generate a single Pyretic policy corresponding to the original NetAssay query.

Maintaining mappings such as the one shown in Table 1 is challenging. The mappings themselves may change over time, so the runtime must adapt the corresponding Pyretic policies as network conditions change. Pyretic does provide a `DynamicPolicy` class that allows policies to be updated while a control program is running, but Pyretic only provides a mechanism for updating a policy; it does not specify how or when those updates take place.

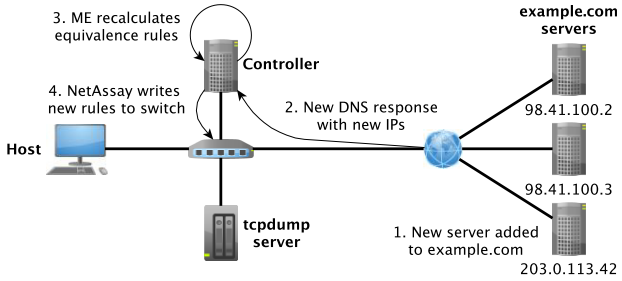
4.3.2 Mapping abstractions to flow space

The NetAssay runtime must maintain mappings higher-level abstractions and the low-level policies that can be expressed in terms of flow space that hardware switches can support. Each ME keeps the lower-level policy up-to-date updates the switch flow table entries. To maintain these mappings, the ME may incorporate data from considerably different sources and involves several important design requirements:

Reactive functional sets. Set membership changes (*e.g.*, changes to the set of authorized hosts, newly compromised hosts, changes in DNS mappings) should trigger the appropriate MEs to recompile the higher-level abstractions into new flow table entries. For example, in Table 1, if the DNS A record mapping for `example.com` changes, the corresponding DNS ME must produce a new Pyretic policy for that corresponding code block. Procera's support for functional reactive programming allows it to handle these types of



(a) Deriving mappings. *In Step 1, the metadata engine at the controller receives DNS responses that are redirected from switches across the network. In Step 2, the NetAssay control module uses the resulting mapping to generate flow table entries that copy and redirect traffic for the appropriate portion of flow space.*



(b) Dynamic updates. *When a new server IP address is added to* `example.com`*, the metadata engine must dynamically regenerate the appropriate Pyretic policy, and the control module must recompile the corresponding flow table entries.*

Figure 1: *The DNS metadata engine maintains a mapping between domain names and IP addresses and subsequently installs monitoring rules for the corresponding flows.*

policy updates [15] and could be a useful as a native Pyretic abstraction.

Reactive specialization. In addition to supporting dynamic updates to the Pyretic *policies* based on dynamic changes to set membership, NetAssay (and ultimately Pyretic) must manage how these policies compile into flow table entries that are ultimately installed in the network switches. Naïvely installing all flow table entries that correspond to some higher-level abstraction would typically exhaust forwarding table memory, often for rules that would never match against any incoming traffic. Frenetic supports "reactive specialization" of flow table rules for "group by" queries (*i.e.*, histograms), where the controller would only reactively install a flow table entry for a group after it had seen at least one packet from the group [6]. When considering metadata that defines set membership (*e.g.*, all prefixes owned by AS 7018), NetAssay will need to exploit this feature to make efficient use of limited switch memory. A development branch of Pyretic supports reactive specialization; we plan to use this feature to support metadata queries in NetAssay.

Coordination between forwarding and monitoring. Certain MEs can maintain mappings between higher-level abstractions and flow space based on external data sources (*e.g.*, BGP routing tables can map AS numbers to IP prefixes, 802.1X

servers can map usernames to MAC addresses). In other cases, however, MEs must derive these mappings from on-path data-plane traffic (*e.g.*, establishing mappings between domain names and IP addresses by capturing DNS responses). Figure 1a shows a process where a DNS ME might intercept DNS responses (Step 1) and install the corresponding rules to generate a copy of traffic coming from `example.com`'s IP addresses (Step 2). In this case, the DNS ME must first install rules to capture the DNS responses and then install an additional set of rules once the mappings are observed. Furthermore, the ME must monitor the responses for changes in DNS records, as in Figure 1b where an additional server is added to `example.com`'s cluster, and update the corresponding Pyretic policies. This particular example requires the ME to intercept subsets of data-plane traffic and implement the reactive functional sets for `example.com`.

Deriving mappings. NetAssay can map high-level primitives to flow space either by monitoring in-network traffic flows that contain these mappings (*e.g.*, DNS records) or by incorporating external information directly (*e.g.*, via a BGP feed). As one example of in-network traffic analysis, the NetAssay DNS ME parses DNS responses for A records that establish mappings between domain names and the corresponding sets of IP addresses. To support the DNS ME, NetAssay installs a forwarding table entry in the switch to divert a copy of all such responses to the DNS ME, which analyzes this traffic off-path to maintain the corresponding mappings. On the other hand, NetAssay's BGP ME can parse an external BGP update feed to maintain mappings between an IP prefix and the origin AS for that prefix (as well as other properties in the BGP update).

5 Summary and Research Agenda

Network operators need better ways to express the traffic they are interested in monitoring, at higher levels of abstractions than current mechanisms provide. Inspired by previous work on intentional naming, we have proposed a notion called *intentional network monitoring*, which allows network operators to specify subsets of traffic they want to monitor based on higher-level entities that more closely match their intent and goals. We have presented a preliminary design and implementation for NetAssay that performs intentional network monitoring for domain names and ASes. Our implementation is by no means complete; rather, our initial prototyping exercise has highlighted many of the research challenges involved in realizing intentional network monitoring, which we now discuss in more detail.

Supporting functional reactive sets in the face of dynamic network conditions introduces significant research challenges. As sets of hosts specified by a certain dimension of metadata change (*e.g.*, ASes announce new prefixes, domain names incorporate new IP addresses, users authenticate and leave) the NetAssay runtime must continually update: (1) the mapping between these sets and the corresponding Pyretic policies; (2) the sets of flow table entries that are ultimately installed in the switch, as part of reactive specialization. Fortunately, previous work on functional reactive network control (*e.g.*, Procera [15]) and reactive specialization (*e.g.*, [6]) provide some possible starting points for solving these problems, but these features are yet to be part of any single language or runtime.

Supporting a diverse, heterogeneous set of metadata constructs is also extremely challenging, as different metadata is represented and updated in different ways. For example, metadata corresponding to domain names requires on-path monitoring of DNS response traffic, metadata corresponding to AS numbers requires incorporating a BGP routing updates, and metadata corresponding to usernames requires incorporating data from an authentication server. To be efficient and scalable, the NetAssay runtime will ultimately need to devise clever filtering mechanisms to avoid recomputation of policies (and recompilation of flow table entries) every time metadata is updated. There are additionally scalability concerns related to number of rules that can be installed into an individual switch. Recent work on judiciously populating switch flow table entries based on active traffic flows and creating multi-layered caches [11] may ultimately provide some helpful optimizations that could be adapted for NetAssay.

Supporting dynamism and heterogeneity introduces challenges related to both scalability (*i.e.*, storing all rules in the flow tables) and performance (*i.e.*, the rate at which flow table entries can be updated, and forwarding performance during updates). NetAssay rules introduce new problems of scale, as a single `match` rule for a domain name or AS number can result in tens of flow table entries (or more), many of which may never match on any traffic. Achieving good performance while supporting dynamism is also challenging: whenever any metadata engine receives a change to a dynamic set, it recomputes all corresponding Pyretic rules, and the underlying Pyretic runtime compiles the policies to flow-table entries. Yet, certain types of auxiliary information are always in a state of flux (*e.g.*, BGP updates experience continual churn, DNS records continue to change, the set of authorized hosts may change over time); such continual change could induce the NetAssay runtime to be continuously pushing new rules to switches. Optimizations such as rule caching [11], memoization during compilation [8], and reactive specialization [6] should ultimately be able to help us realize these design goals.

Although we have designed NetAssay with network monitoring in mind, the types of `match` primitives that NetAssay enables could be coupled with a broader set of *actions*. For example, a network operator might implement a security policy that automatically drops or re-directs traffic that has passed through disreputable ASes, or rate-limits traffic for a certain user or application. We are exploring these possibilities in our ongoing work.

Acknowledgments

This work was supported by NSF awards CNS-1018021, CNS-1409076, and a Google Focused Research Award. We thank Josh Reich for his help integrating with Pyretic.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 186–201, Kiawah Island Resort, SC, Dec. 1999.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, volume 49, pages 113–126, San Diego, CA, 2014.

[3] J. R. Ballard, I. Rae, and A. Akella. Extensible and Scalable Network Monitoring Using OpenSAFE. *Proceedings of USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN)*, Apr. 2010.

[4] I. N. Bermudez, M. Mellia, M. M. Munafò, R. Keralapura, and A. Nucci. DNS to the Rescue: Discerning Content and Services in a Tangled Web. In *ACM SIGCOMM Internet Measurement Conference*, pages 413–426, Boston, MA, Nov. 2012.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM*, Aug. 2007.

[6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. *International Conference on Functional Programming (ICFP)*, pages 279–291, 2011.

[7] N. Gude, T. Koponen, B. Pfaff, J. Pettit, T. Koponen, M. Casado, N. McKeown, S. Shenker, and M. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, July 2008.

[8] A. Gupta, L. Vanbever, M. Shahbaz, S. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. SDX: A Software Defined Internet Exchange. In *ACM SIGCOMM*, Chicago, IL, Aug. 2014.

[9] Hostexploit, June 2011. `http://hostexploit.com/`.

[10] *IEEE Standards for Local and Metropolitan Area Networks: Port-based Network Access Control, IEEE Std 802.1x*, June 2001.

[11] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite CacheFlow in Software-Defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks (HotSDN)*, Chicago, IL, Aug. 2014.

[12] S. Narayana, J. Rexford, and D. Walker. Compiling Path Queries in Software-Defined Networks. *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks (HotSDN)*, pages 1–6, Aug. 2014.

[13] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular SDN Programming with Pyretic. *USENIX ;login,*, 38(5):128–134, Oct. 2013.

[14] S. Shin and G. Gu. CloudWatcher: Network Security Monitoring using OpenFlow in Dynamic Cloud Networks. *IEEE International Conference on Network Protocols (ICNP)*, Oct. 2012.

[15] A. Voellmy, H. Kim, and N. Feamster. Procera: A Language for High-level Reactive Network Control. In *ACM SIGCOMM Workshop on Topics in Software Defined Networks (HotSDN)*, pages 43–48, Helsinki, Finland, Aug. 2012.

[16] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–42, Lombard, IL, Apr. 2013.

[17] K. Zeilenga. Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map. RFC 4510, June 2006.