2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Athena: A Framework for Scalable Anomaly Detection in Software-Defined Networks

Seunghyeon Lee[†], Jinwoo Kim[†], Seungwon Shin[†], Phillip Porras[‡], and Vinod Yegneswaran[‡]
[†]KAIST [‡]SRI International
{seunghyeon, jinwoo.kim, claude}@kaist.ac.kr, and {porras, vinod}@csl.sri.com

***Abstract*—Network-based anomaly detection is a well-mined area of research, with many projects that have produced algorithms to detect suspicious and anomalous activities at strategic points in a network. In this paper, we examine how to integrate an anomaly detection development framework into existing software-defined network (SDN) infrastructures to support sophisticated anomaly detection services across the entire network data plane, not just at network egress boundaries. We present *Athena* as a new SDN-based software solution that exports a well-structured development interface and provides general purpose functions for rapidly synthesizing a wide range of anomaly detection services and network monitoring functions with minimal programming effort. *Athena* is a fully distributed application hosting architecture, enabling a unique degree of scalability from prior SDN security monitoring and analysis projects. We discuss example use-case scenarios with *Athena*'s development libraries, and evaluate system performance with respect to usability, scalability, and overhead in real world environments.**

I. Introduction

Tracking and responding to network traffic anomalies is an ubiquitous and significant challenge faced by all network operators [1]. Sudden and massive deviations in the volume and mix of data flowing through the network infrastructure, due to congestion, outages, network probes, and flooding attacks, are so frequent that few networks would fail to benefit from integrated network anomaly detection services. Even anomalies that arise from benignly-motivated incidents, such as *flash crowds*, may threaten the ability an enterprise to maintain reliable network operations.

Software-Defined Networking (SDN) offers the potential to explore new efficient strategies for instrumenting networks, integrating software detection services, and responding to anomalies that would otherwise impede network operations. For example, by leveraging the centralized control plane to track new flow requests and extracting data plane statistics, one can analyze live network data flows without requiring the insertion of third-party devices (e.g., a network monitoring middlebox). Indeed, a survey of recent research proposals illustrates the increasing exploration of SDN strategies to integrate network misuse and anomaly detection services [2], [3], [4], [5], [6], [7], [8], [9], [10].

However, while these proposals offer a diverse exploration of SDN-based monitoring strategies, several significant technical challenges remain. First, most SDN-based threat monitoring proposals that implement monitoring services as SDN applications, do not consider the issue of network-wide large-scale data extraction and management across a distributed data plane, with many switches and controller instances. Second, prior research has primarily focused on applications designed for *specific* suspicious or anomalous activity. Prior SDN monitoring projects have employed focused subsets of features from SDNs for tracking specific phenomena rather than defining a feature framework for building SDN-enabled network anomaly detection services. Third, there is limited research on network features and anomaly detection algorithms for activities that cross the control and data plane boundaries.

We present a scalable framework, called *Athena*, for constructing network monitoring services in large-SDN deployments, and providing flexible third-party development of new detection models. *Athena* exports an API that provides a well-structured development environment for implementing a wide range network anomaly detection applications across a large physically distributed SDN control and data plane. *Athena*'s API offers developers an abstraction from a complex data extraction service, reducing the programming effort required to implement and deploy new anomaly detection services. In contrast to existing work, *Athena* includes a wide range of network features and detection algorithms for use in simplifying the design and deployment of general-purpose network data plane anomaly detection applications in large-scale SDN networks. In fact, other than its requirement for OpenFlow support, the framework avoids the need for specialized hardware, thereby dramatically minimizing the need to modify an SDN stack when introducing new anomaly detection services.

To address the issue of scalability across large distributed SDN deployments, *Athena*'s network feature collection and data management framework employs a distributed database, a computing cluster, and a distributed controller. It collects and generates network features above the SDN controller instances in a distributed manner, and publishes the network features to a distributed database. To accelerate runtime detection model generation, *Athena* incorporates a machine learning (ML) library from which anomaly detection algorithms may be implemented and then deployed as jobs across *Athena*'s computing cluster. *Athena* exports high-level APIs that allow operators to design and deploy anomaly detection applications with a minimum of programming effort. This approach both reduces the total computation time necessary to perform anomaly detection, while increasing the scalability of data management services on which these algorithms are run.

The key contributions presented in our paper include the following:

• Introduction of *Athena* as a new anomaly detection application development framework that leverages SDN functionality to explicitly support ML-based network anomaly detection. *Athena* integrates without modification into existing SDN infrastructures.

• Presentation of a set of SDN-wide features that enable *Athena* to host a wide range of anomaly detection services, including the detection of anomalies directly within the SDN control and data planes. Specifically, we considered eight major operational functions performed by SDN networks, and from each function extract relevant feature sets to drive anomaly detection services.

• Presentation of eight core APIs, over 70 utility APIs, and 11 popular machine learning algorithms that facilitate the rapid development of new anomaly detection services. Collectively, these facilitate both batch and live mode anomaly detection, while shielding *Athena* developers from the complexities of distributed feature extraction, data management, and threat response generation.

• Presentation of a fully-distributed architecture enabling highly scalable network anomaly detection. We evaluate the system on a large-scale dataset from a datacenter-like physical network environment, and assess performance degradation by evaluating the overhead when it is integrated into one of the most visible open-source network operating systems (ONOS).

• Demonstration of the generality of our framework by replicating detection algorithms from prior publications of SDN security services (that had previously required the integration of specialized hardware) and development of a specialized SDN stack anomaly detector capable of detecting a novel anomaly that we refer to as the *Network Application Effectiveness (NAE)* problem.

We have released the *Athena* prototype implementation as an open-source project to support the SDN community and stimulate other academic research efforts[1].

II. Five Considerations in Designing Scalable SDN Anomaly Detection Services

The design and integration of network anomaly detection services in traditional networks have been well studied [12]. We understand how to integrate hardware elements to extract network traffic features at strategic network points, how to apply a wide range of analytics to these features, and how to correlate relevant reports to effect a network's security posture. However, SDNs offer a departure from these strategies by providing new methods for dynamic instantiation of monitoring services with a global view of the network topology that is continuously updated.

The design goals for the *Athena* anomaly detection framework for SDNs include the following: 1) provide for an extensive feature extraction infrastructure without hardware device integration, 2) abstract data acquisition and simply anomaly detection service implementation, and 3) simplify large-scale deployment of monitoring services without modification to the SDN infrastructure itself. This section discusses several key choices and considerations toward addressing these goals.

[1]The source code is publicly available at SDNSecurity.org [11].

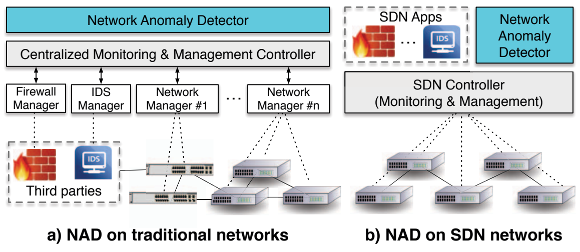


Fig. 1. An illustration of network anomaly detector (NAD) in a) traditional networks and b) SDNs.

1) Centralized Management Perspective: We illustrate the two management design paradigms, legacy versus SDN, in Figure 1. While SDN's support centralized network-wide monitoring, developers are burdened with the requirement to implement a centralized controller that monitors and manages the network with a global network view. Furthermore, each device may export a wide menagerie of network features (e.g., logs, statistics, and alerts) forcing the developer to perform post-processing to normalize network features using approaches like VAST [13].

2) Network Feature Quality and Accessibility: The selection and accessibility of legitimate network features for real-time monitoring is an essential task when designing any network monitoring framework, particularly frameworks that are intended to facilitate *off-the-shelf sharing* of anomaly detection algorithms. *Athena* directly addresses these needs for feature quality and accessibility, by providing (i) normalized network-feature access and libraries from which a wide range of ML-based detection algorithms may be designed and deployed across large-scale distributed SDNs, and (ii) dynamic threat mitigation APIs that can be launched in response to perceived threats to the SDN. We will leverage example use cases to address well-known anomaly detection problems, and demonstrate the utility of the *Athena* framework by applying it to detect a novel SDN-specific anomaly (Section V).

3) Monitoring Networks with Highly Dynamic Topologies: Enterprise networks continue to become more complex, more virtual, and dynamic to meet an increasingly diverse set of operational requirements. Traditional security and network devices are often integrated at physically strategic points in the network, and it is the burden of the operator to manage and adjust their integration and configuration as the network topology changes. Since *Athena* is built into the SDN control layer, applications built over this framework can automatically extend monitoring and anomaly detection capabilities to dynamic network topologies.

4) Flexible Scale-up and Scale-out of the Anomaly Detection Framework: Large network environments imply increased event volumes, larger sets of distributed switches within the data plane, and complex network operating requirements that increase the complexity of both the control plane and the hosted network applications. A key challenge for *Athena* is to introduce a network monitoring framework that will enable anomaly detection applications to scale in speed, to offer distributed computation, and to provide data management APIs that enable high volume event processing. To date, several network monitoring projects have pursued scalability among their design requirements, such as [14], [15], [16], [17], [18], [7], [3], [2], [19], [20], [5]. However, most of these efforts (with the exception of [17]) focus on uncontrolled environments and none of them provide explicit support for anomaly detection algorithms. In contrast, *Athena* is designed as a fully-distributed event collection and processing framework with scalable feature collection and management to support anomaly detection.

5) Coding Efficiency for Anomaly Detection Algorithm Development: While SDN controller APIs [21], [22], [23], [24], [14] enable a wide range of network applications that have been implemented and shared, these APIs are largely insufficient to support network anomaly detection applications. This absence of a generalized network monitor development framework for SDNs contributes to slower progress in the design and deployment of SDN security applications. To overcome this issue, *Athena* exports high-level APIs that support the extraction of critical features, management of data streams, and detection of anomalous network behaviors.

III. Athena Design

Athena is a fully-distributed network anomaly detection framework, in which an *Athena* instance is hosted above each distributed SDN controller, such as with ONOS instances deployed across a wide area network. For example, Figure 2 illustrates three *Athena* instances that are distributed across three SDN controllers. Each *Athena* instance monitors the network behavior that is associated with its hosted network controllers and the data plane hosted by the controllers.

As shown in Figure 2, conceptually, *Athena* incorporates the *Feature Generator*, which collects SDN control messages issued by the local control and data plane, generates network features, and publishes features to a distributed database (a DB cluster) for feature management. The *Attack Detector* detects potential network problems using the developer-defined detection algorithm. The *Attack Reactor* has responsibility for mitigating detected threats by issuing mitigation actions to the data plane. Operators need not modify their existing SDN stack to host *Athena*, as its inputs are SDN control messages, along with small code stubs within the controller. We discuss the details of *Athena's* architecture in Section III-A.

Above the framework, *Athena* provides the set of components that compose its user-friendly development environment. *Athena* exports a high-level API called as the *Athena NB API*, which allows developers to create anomaly detection applications in a manner that is agnostic to the underlying SDN implementation. *Athena* offers an abstraction to the controller and data plane implementations and versions, enabling rapid prototyping and minimizing deployment costs.

Athena provides 8 core and 70 utility APIs, described in Table II. Developers implement anomaly detection tasks as *Athena apps* (shown in Figure 2), using the *Athena Northbound API* (NB API). These applications generate anomaly detection models, perform real-time detection, and implement live threat responses.

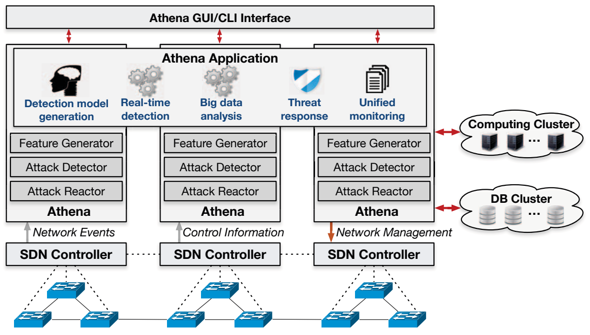


Fig. 2. *Athena's* conceptual architecture; An illustration of the *Athena* anomaly detection framework hosted over a wide-area SDN with distributed controllers. Each controller hosts an instance of *Athena* that instantiates the anomaly detection task per instance. These components can also integrate with the third-party DB cluster and computing cluster. *Athena* applications perform diverse anomaly detection tasks, and operators control the *Athena* applications via a graphical user interface.

A. Athena System Design

Figure 3 illustrates the three major elements of the *Athena* framework: the *southbound element*, the *distributed DB and computing clusters*, and the *northbound element*. The southbound element monitors the network behaviors including the control plane, extracts features from the SDN control messages, implements live detection algorithms, and invokes mitigation reaction. The northbound element exports high-level APIs to enable analysis applications to perform anomaly detection tasks, including network monitoring. The third major element is composed of a distributed database cluster that provides network-wide feature access, and a computing cluster for running distributed parallel instances of *Athena* applications.

1) The Athena Southbound (SB) Element: The Southbound element's main purpose is to isolate control messages, extract features to drive the analysis algorithms, and mitigate detected problems. However, these tasks must be performed across several parallel controller instances and many physically distributed switches. *Athena* achieves this scale by employing SB instances that operate separately on each controller. Further, it uses distributed 3rd party applications to provide parallel data processing environments. Each instance is responsible for monitoring its associated controller and those switches that the controller directly manages, then provides detection algorithms and reaction strategies. The *Athena* SB element consists of four major sub-components:

1A) SB Interface: The main role of the SB Interface is to monitor (selected) SDN control messages issued by both the data plane and the control plane, and to deliver network management commands issued by the Attack Reactor (e.g., issuing flow rules) through the *Athena* Proxy. The proxy is a small code snippet instantiated at each controller instance. *Athena* leverages proxy-stubs that work like general network applications to avoid consistency issues that might arise from issuing control messages to the data plane without involving the controller. When the Athena Proxy issues flow rules to the data plane, the controller automatically updates its internal status according to the issued flow rules.

TABLE I. AN ENUMERATION OF *Athena* FEATURE TYPES.

Category	*Description*	*Examples*
Protocol-centric	Features derived from SDN control messages directly.	Volume (Packet counts, byte counts)
Combination	Combined features derived from *Protocol-centric* by pre-defined formula.	Flow utilization (Packets / Duration)
Stateful	Stateful features according to the network states.	Pair flow ratio (Pair flows / Total flows)

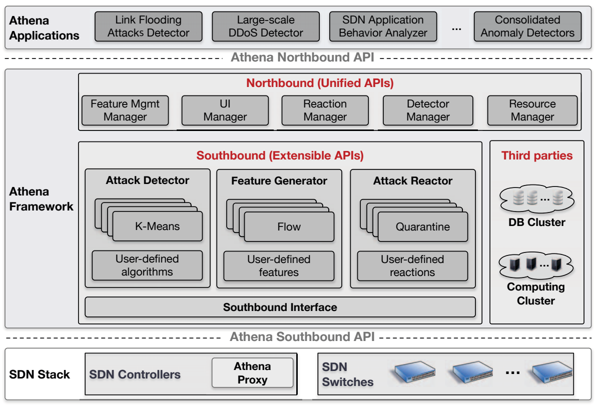


Fig. 3. An overview of the *Athena* system architectural components. The *Athena* framework is composed of the extensible southbound element, the unified northbound element, and the *Athena* application instantiation layer.

1B) Feature Generator: The Feature Generator examines incoming control messages to derive *Athena* features, which we enumerate in Table I, and the internal status of control plane to extract important behavioral features from the control plane (e.g., tracking flow origins). The Feature Generator maintains hash tables to track the status of the previous features for generating *Variation* and network status for maintaining message *State*. The Feature Generator includes a garbage collector to periodically remove outdated entries. It also attaches additional meta information (e.g., timestamp). We discuss the details of *Athena* features in Section III-A3.

1C) Attack Detector: The Attack Detector uses detection algorithms to find potential network threats. The detector generates detection models according to requests from the Detector Manager in the *Athena* NB, and analyzes generated features from the Feature Generator. It is designed to operate live or in batch mode. When it receives a request related to one or more tasks, it translates the request to functions and performs jobs with a single and a distributed manner according to the type of job. For example, while in learning mode, the Attack Detector distributes jobs to the computing cluster to provide a scalable analysis environment. For a small dataset, it handles the request on a single instance to reduce communication overhead.

1D) Attack Reactor: The Attack Reactor enforces mitigation strategies to the data plane. When it receives mitigation strategies from the Detector Manager, it translates requests to network management messages to be sent to the data plane through the Athena Proxy.

2) The Athena Northbound (NB) Element: The *Athena* Northbound element exports Northbound APIs, which allow application developers to utilize *Athena*'s functionalities for an anomaly detection, providing scalability as well as SDN implementation transparency. We describe below the five major sub-components of the NB element.

2A) Feature Management Manager: The Feature Manager provides a unified mechanism that applications use to retrieve and receive network features according to user-defined constraints. It receives feature requests from each application and translates them into queries that it issues to the *Athena* distributed database. It transfers requested data from the distributed database to the computing cluster managed by the Detector Manager, reducing data transmission overhead while transferring large-scale datasets. It maintains an *event delivery table*, which maintains a set of application constraints, that is triggered when incoming network features match the constraints. When processing real-time incoming features from the *Athena* SB, it forwards the features to both the *Athena* applications and the Detector Manager.

2B) Detector Manager: The Detector Manager provides a wide-range of well-known ML algorithms to generate detection models including a simple threshold-based detection algorithm. It also validates large-scale network features. It works with the Feature Manager to dynamically validate incoming network features and provides unified APIs that allow operators to perform detection tasks with transparency to details of the algorithms. For example, when running a K-Means algorithm in the clustering category with the Decision Tree algorithm in the classification category, the operator will use the same APIs in Table II. An operator does not have to consider the characteristics of each ML type, since the Detector Manager automatically configures ML parameters according to requests of ML types by operators (e.g., labeling features).

2C) Reaction Manager: The Reaction Manager provides mitigation strategies that allow the *Athena* applications and operators to enforce mitigation actions by issuing flow rules to the data plane. The applications enforce pre-defined course-of-actions to handle network problems, independently from the underlying network controller, by issuing requests to the SB Attack Reactor, which are automatically translated to flow rules.

2D) Resource Manager: The Resource Manager exports functions to manage resources related to feature collection. It dynamically adjusts the number of monitored network entities and generated network features, according to requests from *Athena* applications. This allows *Athena* applications to control monitoring fidelity based on dynamic network conditions.

2E) UI Manager: The UI Manager services an interface, that displays *Athena* application's results and provides an interaction mechanism.

3) Athena Off-The-Shelf Strategies: As an anomaly detection framework, *Athena* provides a set of off-the-shelf strategies including network features, detection algorithms, and reactions.

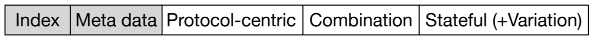


Fig. 4. The format of a single *Athena* feature: The gray box represents index fields, the white box is for feature fields.

3A) *Athena* Features: In total, *Athena* exposes over 100 network monitoring features to the NB API. The full list of network features is available at SDNSecurity.org [11]. The types of features are enumerated in Table I. Protocol-centric features are directly derived from OpenFlow control messages, such as packet count from flow statistics. Combination features refer to combined features derived from pre-defined formulas, such as the features in [10], and include more meaningful information regarding SDN-specific features. For example, Flow Utilization represents how much traffic a flow delivers to its associated output port. The Stateful field represents the features including states of indicator operations. For example, Pair Flow Ratio represents how many flows manifest active two-way connection between a sender and a receiver.

Athena's feature format is illustrated in Figure 4. The feature format consists of the index fields and the feature fields. The index fields include the `Index`, which contains information about feature's origins (e.g., Switch ID, port ID) including indicators (e.g., OpenFlow match fields), and `Meta Data` that represents additional information such as a timestamp and semantics of the control plane associated with the feature (e.g., Flow origins). The feature fields are appended after the index fields to represent an actual behavior of the network.

3B) *Athena* Detection Algorithms: *Athena* provides a set of detection algorithms that allow its applications to find potential network threats and problems. The algorithms include five categories, which are described in Table IV. Currently, *Athena* supports 11 machine learning algorithms hosted on a computing cluster to perform scalable analyses.

3C) *Athena* Reactions: *Athena* Reactions manage the data plane according to changes of the network status. After detecting a network threat, the applications hosted by *Athena* may choose to invoke a mitigation strategy. *Athena* currently supports two types of mitigation actions: `Block`, which blocks certain hosts; and `Quarantine`, which isolates suspicious hosts to user-defined destinations.

IV. The Athena Development Environment

The *Athena* development environment (DE) exports a set of high-level APIs that allow operators to design and implement a scalable network anomaly detector in a manner that abstracts both the SDN version dependencies and infrastructure-specific configuration details. Here we discuss *Athena*'s northbound APIs and outline the steps involved in implementing various anomaly detection services.

A. Athena Northbound API

The Northbound (NB) API is configuration-based. Developers use it by configuring parameters to execute anomaly detection tasks. For example, one can generate a detection model by defining a set of 1) detection parameters, such as "`TCP_PORT==80 && time==1 day`"; 2) detection features, such as "`sampling (20%), default normalization`"; and 3) a preferred detection algorithm, such as "`K-means, k==5`".

Athena currently supports eight core functions described in Table II, and several pre-defined parameters, described in Table III. `RequestFeatures` is a monitoring API that retrieves desired *Athena* features using the query interface. For example, a developer may create a query that requests "`flow utilization per network application`", "`unstable ports during a 1-day temporal window`" and "`top 10 congested links`". *Athena* currently supports the query operators described in Table IV. The `ManageMonitor` API uses queries to turn monitoring on/off for specific network features.

As a detection-related API, we provide the `GenerateDetectionModel` for creating detection models, and `ValidateFeatures` for large-scale feature validation. The APIs commonly receive a query to retrieve a desired feature set, and apply the `Preprocessor`, which transforms the features before use. `GenerateDetectionModel` defines a detection algorithm with its parameters (e.g., *K* of the *K-Means* algorithm), and generates a detection model. The model is used by `ValidateFeatures` to validate target features, and it produces results that summarize the validation task. *Athena* generates a detection model during the learning phase when a machine-learning (ML) algorithm is employed, and exports a pre-defined model without a learning phase when using other algorithms (e.g., threshold-based detection). Table IV describes the supported functions of the *Preprocessor* and ML algorithms.

Athena provides APIs that allow an application to handle network features in an online manner. First, `AddEventHandler` enables analysis applications to receive *Athena* features dynamically. Applications register an event handler with a user-defined query. The manager then dynamically evaluates whether an incoming feature satisfies the query, and if so it forwards the feature to the applications. For example, an application may pass the query "`IP_DST==server address && Port==80`". The event handler is also used for live validation by `AddOnlineValidator`. This allows an operator to define an *operational mode* for specific anomaly detection tasks (e.g., A stand-alone mode, and a distributed mode). `ShowResults` represents the results as a visualized graph, providing operators with direct insight to *Athena's* results. The `Reactor` enforces mitigation strategies to the data plane according to requests of the application with queries, such as "`IP_SRC in {suspicious hosts}`" and invokes response functions described in Table IV.

B. Athena Application

Figure 5 illustrates the steps involved in designing and implementing an anomaly detector. Developers select off-the-shelf strategies (e.g., network features, detection algorithms, and reactions) to perform anomaly detection tasks. Based on these selections, they use supported NB APIs to construct a consolidated anomaly detector, including the selected network feature generation, model creation with fine-grained filtered

TABLE II. THE *Athena* CORE NORTHBOUND API.

Function	***Description***
RequestFeatures(*q*)	Request a set of *Athena* features with user-defined constraints including feature re-organization.
ManageMonitor(*q, o*)	Turn on/off a network monitoring including a feature generation.
GenerateDetectionModel(*q, f, a*)	Generate an anomaly detection model according to an user-defined algorithm and features.
ValidateFeatures(*q, f, m*)	Validate a set of *Athena* features with a generated detection model.
AddEventHandler(*q*)	Register an event handler to retrieve features from *Athena* according to user-defined constraints.
AddOnlineValidator(*f, m, e*)	Register an online validator to examine an incoming feature in an online manner.
Reactor(*q, r*)	Enforce an action to the data plane.
ShowResults(*r'*)	Display the results from *Athena* with a graphical interface.

TABLE III. PARAMETERS OF THE *Athena* NORTHBOUND API.

Parameter	***Description***
Query (*q*)	Unified query to retrieve *Athena* features with constraints.
Preprocessor (*f*)	Preprocessing statement to re-design features.
Algorithm (*a*)	Description of an algorithm including parameters of the algorithm.
Model (*m*)	Generated detection model.
Results (*r'*)	Results of a validation or a feature request.
Event handler (*e*)	Event handler to receive online *Athena* events.
Reactions (*r*)	Reactions for handling suspicious hosts.
Operations (*o*)	Flags for a network monitoring per features.

TABLE IV. AN ENUMERATION OF SUPPORTED FUNCTIONS PER PARAMETER.

Query (*q*)	Operators
Arithmetic	>, >=, ==, !=, <=, <
Relationship	*and, or*
Options	*Sorting, Aggregation, Limiting*
Preprocessor (*f*)	Description
Weighting	Emphasize certain features
Sampling	Select a subset from entire features
Normalization	Standardize the range of independent variables
Marking	Mark a set of entry labeled as malicious entry
Algorithm (*a*)	Supported algorithms
Boosting	Gradient Boosted Tree
Classification	Decision Tree, Logistic Regression, Naive Bayes, Random Forest, SVM
Clustering	Gaussian Mixture, K-Means
Regression	Lasso, Linear, Ridge
Simple	Threshold
Reactions (*r*)	Description
Block	Block target hosts
Quarantine	Isolate hosts in honeynets
Operations (*o*)	Description
True	Turn on network monitoring
False	Turn off network monitoring

network features, feature validation with a large-scale dataset, run-time threat detection logic, a dynamic threat mitigation policy, and results from GUI/CLI generation.

Athena automatically performs the anomaly detection task, including task integration with the external DB cluster and computing cluster. It reports (intermediate) results to the application while performing anomaly detection. The application updates internal status and configures new *Athena* hosted anomaly tasks based on the results. Furthermore, *Athena* provides a GUI/CLI interface that allows the operator to receive alerts and manage the *Athena* application in a centralized manner.

V. ATHENA USE CASES

To illustrate the utility of the *Athena* framework, we present multiple sample anomaly detection applications. Due to space limitations, we only show the pseudocode for the DDoS Detector (scenario #1).

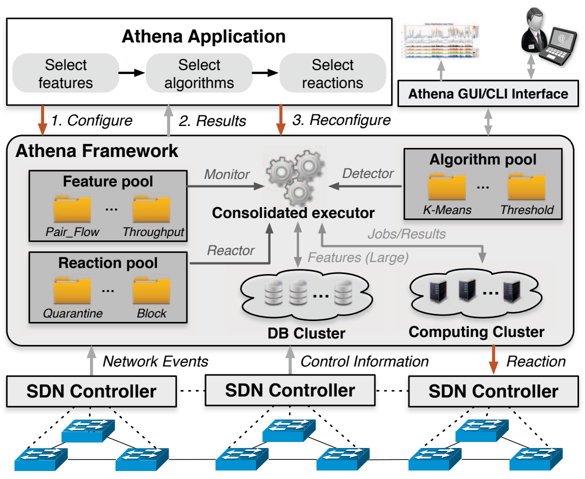


Fig. 5. Implementing a general purpose anomaly detector across a distributed SDN stack with *Athena*.

TABLE V. A LIST OF POSSIBLE FLOW-RELATED FEATURES TO DETECT DDOS ATTACK. THE (*) STAR NOTATION INDICATES THE PREFIX OR POSTFIX OF THE *Athena* FEATURES.

Characteristic	**Possible features**
Unidirectional traffic	PAIR_FLOW, PAIR_FLOW_RATIO
Traffic volume pattern	PACKET_COUNT, BYTE_COUNT, BYTE_PER_PACKET, PACKET_PER_DURATION, BYTE_PER_DURATION
Duration of flow	DURATION_SEC, DURATION_N_SEC

A. Scenario 1: A Large-scale DDoS Attack Detector

One strength of the *Athena* framework is its ability to create scalable network anomaly detection services across large and physically distributed network environments. To demonstrate this scalability, we implement a large-network DDoS attack detection application using *Athena*'s northbound APIs. Since *Athena* automatically collects all of the network features across the SDN data plane by default, an operator may deploy *Athena* on the network to automatically gather the features necessary to drive our detector. Here, we discuss DDoS model creation, feature validation, and summarize our test results.

Creating the DDoS Detection Model: Detection model creation begins with a definition of the desired network features for use during the training phase. The developer sets

Application 1 A pseudo-code illustration of an *Athena*-based DDoS attack detection application.

```
/* Define the features to be trained */
q_train = GenerateQuery (constraints of features);

/* Define data pre-processing */
f = GeneratePreprocessor (Normalization,
          Weight for certain features,
          Marking malicious entries,
          ...);

/* Register the features used in the algorithm */
f.addAll(candidate features);

/* Define an algorithm with parameters */
a = GenerateAlgorithm (a detection algorithm);

/* Generate a detection model */
m = GenerateDetectionModel (q_train, f, a);

/* Define the features to be tested */
q_test = GenerateQuery (constraints of features);

/* Test the features */
r' = ValidateFeatures(q_test, f, m);

/* Show results with CLI interface */
ShowResults(r');
```

TABLE VI. A COMPARISON OF THE TEST ENVIRONMENT.

Category	[10]	Athena
Switch	3 OF switches	18 OF switches (6 physical, 12 OVS)
Link	3 links	48 links
Controller	1 instance	3 instances
Feature	6-tuples	10-tuples
Algorithm	SOM	K-Means

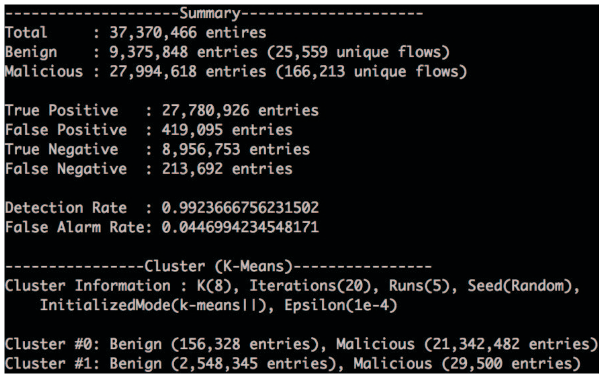
the data preprocessing parameters to normalize the features that capture the characteristics of a DDoS attack described in Table V. These features are set by the `f.addAll()` utility API. Here, we configure `Weight` for emphasizing certain network features, and `Marking` for annotating malicious entries [2].

`Algorithm` represents a detection model, and it is configured with a machine learning algorithm and its parameters (e.g., we may choose K-Means with k = 5, and 20 iterations). Developers then invoke `GenerateDetectionModel` to create the detection model, and *Athena* distributes the ML detection tasks to compute worker nodes. After job completion, the application receives the detection model. These steps are outlined in the pseudocode.

DDoS Feature Validation: The developer next defines the desired network features to receive results from an analysis of the testing phase. The developer defines `Query` and `Preprocessor` in the same way, and calls `ValidateFeatures` with the pre-defined `Query`, `Preprocessor`, and the `Model`. Upon completion of the testing phase, *Athena* generates a testing summary, as illustrated in Figure 6.

DDoS Testing Environments and Results We established a testing environment to reflect an enterprise-scale network topology as illustrated in Figure 7, and compare our environment with previous work for an SDN-based DDoS detection [10], as described in Table VI. Our topology consists of 48 links and 18 switches (6 physical switches, and 12 OVS switches) managed by three distributed network controllers, including *Athena* instances. We deployed a *K-Means*-based algorithm with 10 tuples, and simulated the testing environment using a Mininet environment. The attack scenario is similar to traffic patterns in [10]. The detection rate is 99.23%, and false alarm rate is 4.46%. We will discuss these results further in Section VII-B.

[2]These labels are used by the supervised and unsupervised learning algorithms.



```
--------------------Summary--------------------
Total     : 37,370,466 entires
Benign    : 9,375,848 entries (25,559 unique flows)
Malicious : 27,994,618 entries (166,213 unique flows)

True Positive   : 27,780,926 entries
False Positive  : 419,095 entries
True Negative   : 8,956,753 entries
False Negative  : 213,692 entries

Detection Rate  : 0.9923666756231502
False Alarm Rate: 0.0446994234548171

---------------Cluster (K-Means)---------------
Cluster Information : K(8), Iterations(20), Runs(5), Seed(Random),
    InitializedMode(k-means||), Epsilon(1e-4)

Cluster #0: Benign (156,328 entries), Malicious (21,342,482 entries)
Cluster #1: Benign (2,548,345 entries), Malicious (29,500 entries)
```

Fig. 6. Output of the DDoS detector application. The details of cluster information are excluded after cluster #2.

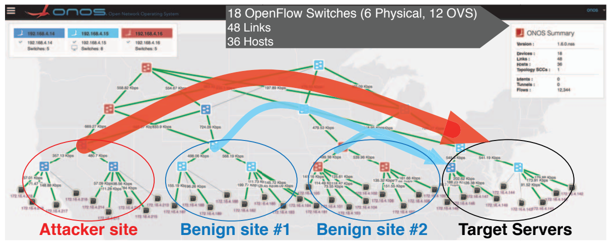


Fig. 7. Enterprise-scale topology for evaluating the DDoS Detection application.

B. Scenario 2: Link Flooding Attacks (LFA) Mitigation

LFA represents a serious and common network attack, in which the adversary saturates a target network area with few resources [25]. A noteworthy aspect of the *Athena* framework is that, unlike some other tools, it produces monitoring applications that are deployable without modification to the underlying network infrastructure. For example, we compare the implementation of LFA mitigation using *Athena* with *Spiffy* [26], which utilizes the transmission rate control mechanism within the TCP protocol to detect and mitigate LFA. Spiffy adjusts how much traffic should be delivered through a conditional assessment of the current network status. It assumes that malicious flows do not respond to a temporary change of a network status, which differs from the normal profile of legitimate flows. As a solution, *temporary bandwidth expansion* (i.e., expanding bandwidth temporarily to detect malicious flows by leveraging characteristics of TCP) and *runtime flow migration* (i.e., re-assigning flows to expand bandwidth for a suspicious link) are proposed.

TABLE VII. Comparison of Link Flooding attack detection and mitigation strategies using SDNs.

Category	Spiffy [26]	Athena
Link congestion	SNMP	Built-in
Rate change	OpenSketch [4]	OF switch
Traffic engineering	Edge router	All switches
Insider threat	Out of scope	Covered

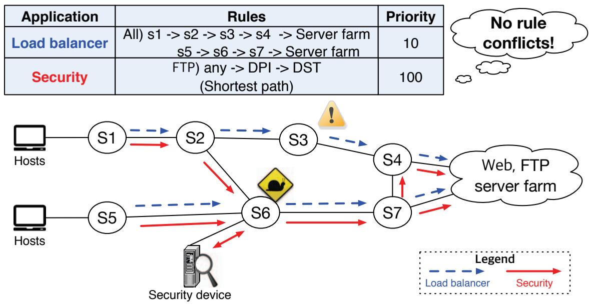
LFA Mitigation Service using *Athena*: We have implemented a comparable Link Flooding Attacks mitigation service as an *Athena* application. In Table VII, we present a comparison of the Spiffy implementation of the LFA mitigation service using the same solution implemented with the *Athena* framework. Here, the demanding functions involved in LFA detection and mitigation include solving link congestion detection, recognizing per-flow rate changes, and implementing flow alterations.

LFA Event Handler Registration: The LFA detector receives link usage to measure link utilization and per-flow changes to distinguish attackers. Since *Athena* provides various volume-based features including per-flow changes, we define these candidate features, including a threshold. For example, the candidate features are volume-based features such as `port_rx_bytes_var`, which represents changes at each port, and `flow_byte_count_var`, which represents the change in byte counts. Likewise, we choose volume-based features (e.g., `port_rx_bytes`). Finally, we simply call the `AddEventHandler` API with a pre-defined event handler to perform the detection and mitigation of incoming events.

LFA Detection Logic: Developers may implement the custom detection logic in the *Event_Handler*. The detection logic includes lightweight threshold-based flooding detection, which measures volume per port, and may use a TBE-based detector that tracks per flow changes. Lastly, the mitigation logic simply blocks suspicious hosts based on the detection results by invoking `Reactor`.

Using *Athena*, we can implement the proposed Spiffy LFA detection and mitigation mechanism with our SDN test environment in under 25 lines of Java code, excluding the custom detection logic.

Comparing Athena-based LFA mitigation with Spiffy: Table VII shows the comparison of LFA detection and mitigation using Spiffy and *Athena*. Spiffy uses an SNMP-based link utilization measurement to detect congested links. However, operators need to configure SNMP-based measurement functions, and the network's switches must also support this function. Spiffy leverages *OpenSketch*-enabled switches [4] to detect rate changes from an edge. Although OpenSketch could reduce overhead to measure rate changes, operators must deploy OpenSketch-enabled switches into their existing network infrastructure. This increases the deployment cost of the Spiffy solution, as it requires features that are non-standard in many SDN environments. Implementing the same anomaly detection algorithm using *Athena* removes the need for vendor-specific network devices and SNMP-enhanced monitoring capabilities. In contrast to the Spiffy environment, *Athena's* application can operate directly on the SDN without data plane alterations.



Application	Rules	Priority
Load balancer	All) s1 -> s2 -> s3 -> s4 -> Server farm s5 -> s6 -> s7 -> Server farm	10
Security	FTP) any -> DPI -> DST (Shortest path)	100

Fig. 8. An illustration of the Network Application Effectiveness problem.

C. Scenario 3: Network Application Effectiveness (NAE)

Network applications are critical elements of a network's SDN stack, as they embody the logic that defines the flow policies for the entire network. Thus, validating application behavior is important, particularly before their adoption into critical or sensitive network environments. Unfortunately, it is difficult to verify application behavior, as modern SDN environments may allow multiple applications to run on a controller in parallel. These applications can also cause conflicts in the form of flow rule contradictions. Several SDN researchers have tried to solve this problem through control mechanisms to resolve network application conflicts [27], [28], [29], [30]. However, they did not explore misuse anomalies that can violate network policies defined at deployment. We refer to this as the *Network Application Effectiveness* (NAE) problem, as illustrated in Figure 8.

Let us assume that an operator installs a load-balancing (LB) application, which defines flow rules intended to evenly distribute a target traffic load across a given set of network services. Consider a security application that attempts to direct FTP-related traffic through an inline security device that analyzes the FTP traffic for signs of malicious command patterns. In this scenario, the LB app and the security app may conflict in their decisions regarding packet forwarding. To handle conflicts, operators set a higher priority for the security app, thus allowing it to over-rule the LB app when their rules conflict. While the problem is well known and addressed by prior work [27], [28], [29], [30], these projects do not consider how to detect the wide range of unwanted network anomalies that may arise as flow rules are evaluated and discarded by the control layer.

The above scenario leads to interesting potential flow pattern anomalies that can arise as the control plane begins to resolve conflicts in the rules produced by the competing applications. For example, if the primary purpose of the network in Figure 8 is to serve FTP users, the network may suffer significant overhead by the unbalanced load produced by the security policy. When the network is dominated by FTP flows, this traffic will begin to saturate S6 due to the security app. Although S3 is available to deliver traffic to the destination, it cannot receive traffic due to the security app's shortest path policy which dictates that all traffic from S6 goes to S7. To evaluate this, we set up an experimental environment, with the edge switches S1 and S5, where each host downloads files or accesses pages from the FTP and web

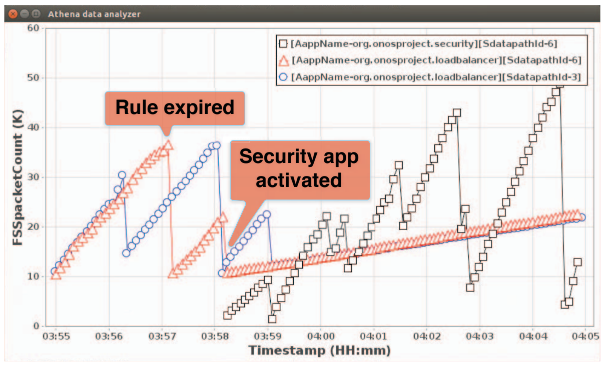


Fig. 9. The coarse-grained analysis result alerted by the *Athena* UI manager, when applications obey the user-defined SLA.

servers respectively.

NAE Monitor Implementation: Detecting the NAE problem is straightforward with *Athena* since it provides a strong query mechanism to retrieve network features with advanced data preprocessing capabilities (e.g., sorting, aggregation, and ranking). We register an event handler to retrieve flow-related network features per application by `AddEventHandler`, and analyze whether a given feature obeys a user-defined SLA (service level agreement)[3] by the `Check_SLA()`[4]. If an incoming event obeys the SLA, the `ResultsGenerator` utility API is invoked to generate the `Results` to notify operators. Finally, it reports anomalous behavior to the operator's GUI interface via the `ShowResults` API, as illustrated in Figure 9.

NAE Analysis Results: Figure 9 shows our application results. Since we set up a query with "`Match DPID==(6 or 3)`", the results only represent relevant features aggregated by app ID, switch ID, and timestamp. It shows a global view of packet count information per switch. The sawtooth pattern in this graph is caused by the expiration of flow rules, since LB app issues flow rules with *soft timeout* [5]. After the security app is activated from *03:58*, the security app takes over most traffic flows, re-routing their packets into the path of the security device. Therefore, the LB App loses forwarding control due to its low priority. Although the LB app is active, the network begins to suffer unexpected saturation in some links and low volume in others. We implemented the NAE problem detector on *Athena* within 30 lines of Java code.

VI. Implementation

We have developed a prototype implementation of the *Athena* framework that integrates within ONOS [14], which is an emerging SDN distributed controller for large-scale networks, focusing on service provider use-cases. *Athena* also employs MongoDB [31] for its distributed database, Spark [32], [33] for its scalable computing cluster, and JfreeChart [34] for the graphical interface. The prototype operates on ONOS version 1.6, using OpenFlow 1.0 and 1.3, MongoDB version 3.2, Spark version 1.6, and JfreeChart version 1.0.13. We have implemented the prototype of *Athena* with approximately 15,000 lines of Java code.

Athena is implemented as an ONOS subsystem, which provides services to an application layer. We modify the implementation of `OpenFlowController` to get OpenFlow control messages directly, and `OpenFlowDeviceProvider` to issue statistics request messages to the SDN data plane. We mark an XID value for statistics request messages to calculate variation features exactly, as ONOS issues request messages to the data plane as part of its management functions. To extract application information per flows, *Athena* leverages the FlowRule subsystem, which manages flow entries within the controller. The *Athena* application operates as a separate process and communicates with the *Athena* framework via interprocess communication to reduce dependencies.

VII. Evaluation

We now evaluate *Athena* with respect to its usability, network scalability, and overhead. To explore its usability, we consider the design of *Athena's* anomaly detection applications against comparable applications developed without *Athena*. For the scalability assessment, we evaluate performance of the large-scale DDoS anomaly detection algorithm introduced in Section V-A. Finally, we measure the overhead of *Athena* feature extraction using the Cbench benchmark. To evaluate our work, we created an experimental environment with five high performance servers (four Intel hexa-core Xeon E5-1650, one Intel octa-core Xeon E5-2650) with 64GB RAM, two Intel I5 quad-core I5-4690 and 16GB RAM memory, seven physical switches [6].

A. Evaluating Usability of Athena

TABLE VIII. The lines of Java codes for a DDoS detector per algorithm (excluding imports).

DDoS detector (Algorithm)	Athena	Spark	Hama [35]
K-Means	45	825	817
Logistic Regression	42	851	829

In evaluating the usability of *Athena*, we implemented a DDoS detection application within different environments, and then provide a rough approximation of the implementation complexity by quantifying the *source lines of code* (SLoC) required to implement the application. While imperfect, SLoC is often used as a metric of usability (e.g., [36], [37]). We believe that SLoC (application compactness), given the lack of a large developer-base for feedback, provides a useful early usability measure, similar in spirit to how it was applied to answer this same question in related prior work.

Each resulting application embodies the functionality of the DDoS attack detector in Section V. As summarized in Table VIII, the application based on *Athena* uses 5% of the lines of code that comparable functionalities require when implemented on Spark [32] and Hama [35].

[3]In this scenario, the SLA is that traffic should be distributed evenly per each switch.

[4]This function is a custom algorithm to detect asymmetric traffic patterns.

[5]The soft timeout is used for deleting flow rules, when there are no incoming packets within a certain time.

[6]Two Pica8 P3290, two PICA8 P3297, two PICA8 AS4610, and one ARISTA 7050T-36.

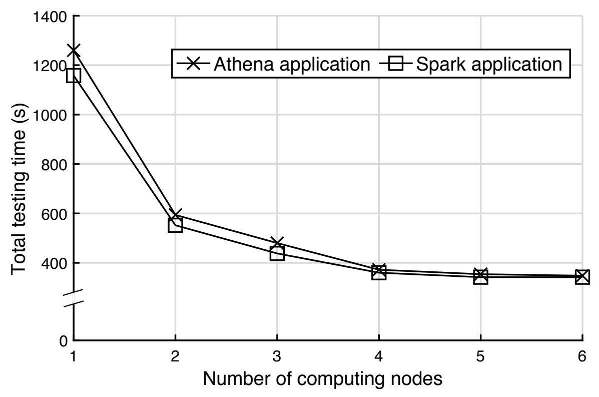


Fig. 10. A performance assessment of the DDoS application while performing anomaly detection tasks per the number of computing nodes.

B. Measuring Scalability of Athena

Figure 10 presents the performance results for the DDoS detection application in Section V-A. We use an experimental environment with ten instances on the three Xeon servers. The instances consist of six compute nodes and a master node on the two hexa-core Xeon servers, and three DB nodes on the octa-core Xeon server. Here, we measure the total testing time according to the number of compute instances. The dataset includes 37,370,466 entries for a 50GB dataset. As the number of computing instances increases, we observe a *linear* decrease in the total processing time and the total test time with six nodes is approximately 27.6% of the test time with a single-compute node instance. We compare the application hosted by *Athena* with an application on Spark, and results show *Athena* introduces a small overhead (under 10%) over the Spark application.

C. Overhead of Athena's Feature Extraction

We measure the overhead of *Athena*'s feature extraction while handling external events and compare this overhead to the ONOS baseline (e.g., Cbench benchmark, and CPU usage). We set up an experimental environment with the hexa-core Xeon server to test Cbench benchmark and two hexa-core Xeon servers with seven physical switches to measure CPU usage while gathering events from the switches.

TABLE IX. CBENCH BENCHMARK FLOW INSTALL THROUGHPUT WITH AND WITHOUT ATHENA (RESPONSE/S) OVER 50 ROUNDS OF TESTING.

	MIN	MAX	AVG
Without	773,618	883,376	831,366
With	107,245	610,724	389,584
With (no DB)	631,647	686,227	658,514
Overhead (no DB)	86.13% (18.35%)	30.86% (22.31%)	53.13% (20.79%)

1) Cbench benchmark with/without Athena: The ONOS testing group evaluates the scalability of ONOS to measure how many burst events could be handled. For example, they evaluate burst Packet_IN event handling throughput with a single instance, which is called the Cbench benchmark. To do this, we evaluate *Athena* using Cbench's throughput mode with the ONOS's recommended settings [38] and summarize results in Table IX. On average *Athena* has 53.13% lower throughput and in the worst-case has 86.13% performance degradation. However, without DB operations, *Athena* induces only 20% performance degradation.

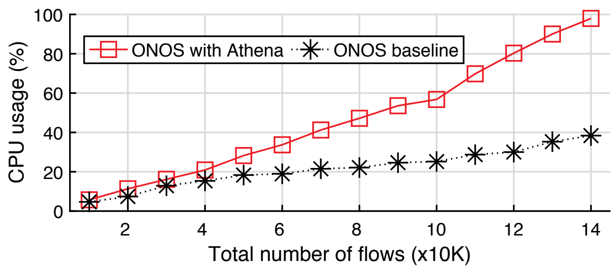


Fig. 11. Average CPU usage while handling flow events with/without *Athena*.

2) CPU usage with/without Athena: The overhead of *Athena* is dependent on how much information is in the SDN stack, including the control and data plane, as it passively monitors and analyzes them both. To compute the monitoring overhead of handling network events from SDN stacks, we conducted experiments to measure the CPU load when using ONOS with and without *Athena*. To do this, we established a testing environment with a controller on the hexa-core Xeon server, which is connected to the six physical switches, and 12 OVS instances on the two hexa-core Xeon and the two i5 servers respectively with dummy flows to generate monitoring events. Figure 11 illustrates the testing results. Since *Athena* stores events to the data plane while maintaining internal status to generate *stateful features*, the flow handling overhead increases according to the total number of flow entries in the switches. We find that ONOS with *Athena* saturates at about 140K flows per second, while the CPU utilization is about 31% for the basic ONOS instance.

3) Discussion: We found that the performance overhead of our system primarily originates from MongoDB related operations. To boost *Athena*'s performance, we will consider replacing MongoDB with a high-performance database like Cassandra [43].

VIII. RELATED WORK

We now discuss how prior projects have sought to address the challenges described in Section II, including various limitations in their coverage. Table X provides a comparison between *Athena* and existing work related to network anomaly detection and monitoring in SDN environments.

Anomaly detection strategies: The inherent centralized control-layer design of SDNs enables operators an efficient potential network-wide choke-point from which to gather a wide range of network features. In fact, several prior anomaly detection projects [26], [6], [9], [5], [39] have successfully leveraged volume-based network features available through monitoring the OpenFlow protocol to detect network anomalies, such as DDoS attacks and switch anomalies. Mehdi et.al. [8] explored the feasibility of adopting traditional anomaly detection techniques with OpenFlow's monitoring capabilities, and Braga

TABLE X. COMPARISON OF *Athena* WITH RELATED WORK FROM NETWORK ANOMALY DETECTION AND NETWORK MONITORING WITH DIVERSE PERSPECTIVES. (*V*: VOLUME-BASED, *S*: STATEFUL, *SP*: SAMPLING, *D*: DPI, *SS*: SDN-SPECIFIC)

Network Anomaly detection										
	Purpose	*Architecture*	*NB Interface*	*SB Interface*	*Feature Management*	*Network Features*				
						V	*S*	*SP*	*D*	*SS*
Athena	Framework for anomaly detection	Distributed	Ad-hoc API	OpenFlow	✓	✓	✓			✓
[37]	Framework for security apps	Single	Script	OpenFlow	✓	✓	✓		✓	
[8], [10]	General anomaly detection	Single	-	OpenFlow		✓	✓			
[6]	Malicious switch detection	Single	-	OpenFlow		✓				
[16]	NFV optimization	Distributed	-	OpenFlow						
[5], [9]	General anomaly detection	Single	-	OpenFlow, sFlow		✓	✓	✓		
[26]	LFA detection	Single	-	OpenFlow, SNMP		✓				
[39]	Framework for anomaly detection	Single	-	OpenFlow		✓				

Network Monitoring								
	Purpose	*Architecture*	*NB Interface*	*SB Interface*	*Custom Switch*	*Resource Optimization*	*Data Persistency*	*Query*
Athena	Framework for anomaly detection	Distributed	Ad-hoc API	OpenFlow			✓	✓
[40]	Framework for network monitoring	Single	RESTful API	OpenFlow			✓	✓
[17]	Distributed network analysis	Distributed	NetConf	Netflow, SNMP, IPSLA	✓			✓
[18]	Distributed network monitoring	Distributed	-	OpenFlow	✓	✓		
[4]	Scalable flow counter monitoring	Single	Ad-hoc API	OpenSketch	✓	✓		
[3]	Scalable flow counter monitoring	Single	Ad-hoc API	OpenFlow	✓	✓		
[2]	Resource allocation for measurement	Distributed	Ad-hoc API	-	✓	✓		
[7], [19]	Efficient flow counter monitoring	Single	-	OpenFlow		✓		
[20]	Low latency flow counter monitoring	Single	-	sFlow		✓		
[41]	Network monitoring with NFV	Single	Policy Language	OpenFlow				
[42]	Framework for network monitoring	Single	Policy Language	Various sources				

et.al. [10] leveraged OpenFlow statistics to demonstrate low-cost detection of DDoS flooding attacks. FRESCO [37] exports a script-based development environment that facilitates the creation of security applications that monitor well-known network features (e.g., TCP session). However, these efforts do not consider how to fully utilize network features derived from an SDN environment to monitor it for behavioral and operational stability.

From the perspective of detection strategies, most related efforts [26], [6], [9], [5], [39], [8], [10] focus on fixed detection algorithms against specific attacks, not general purpose algorithms. While FRESCO [37] and ATLANTIC [39] provide a set of libraries that aim to facilitate attack detection (e.g., port scanning), only the former provides well-structured mitigation actions and it does not support multi-instance controller environments.

Scalable SDN monitoring: Several prior projects have explored various ways to reduce the overhead of gathering volume-based network features [20], [19], [7], [3], [2], [4], [18]. In fact, there have been prior efforts to incorporate a distributed architecture that allows operators to scalably gather network features, such as [17], [2], [18]. However, those projects have not proposed detection strategies for tracking malicious network behaviors. Furthermore, most of these project have assumed the adoption of additional customized switches to perform their scalable network monitoring [17], [18], [4], [3], [2].

Bohatei [16] proposed a scalable DDoS detection and mitigation solution, leveraging an optimization technique that distributes jobs to NFV machines to increase data processing throughput. Bohatei leverages additional NFV devices to perform network functions, and does not directly implement the anomaly detection algorithms. There have also been sampling-based SDN monitoring approaches to reduce collection overhead when collecting statistics from OpenFlow environments [9], [5]. These techniques rely on the adoption of sFlow sampling. Spiffy [26] has demonstrated strategies for Link Flooding Attacks mitigation [25], relying on the SDN's centralized management to insert flow mitigation. A limitation of Spiffy is that it requires a customized switch to measure the network behavior. Finally, there are several additional projects that have explored the feasibility of anomaly detection in SDNs: [37], [10], [8], [6], [39]. Unlike *Athena*, none of these projects address our scalability requirements.

Improving SDN usability: Several prior projects have introduced northbound APIs that allow operators to conduct various forms of network monitoring [42], [41], [2], [3], [4]. Payless [40] provides a semantical monitoring capability that helps operators monitor networks by calling well-structured RESTful APIs that reflect a high-level set of monitoring requirements. DNA [17] introduces a scalable network monitoring function to examine telemetry data sources. Although these previous projects enhance usability by exporting well-structured APIs to operators, they do not provide a detection algorithm to find network anomalies.

IX. CONCLUSION

We explore several challenges in designing scalable anomaly detection services in large-scale SDN environments. We evaluate an initial prototype implementation of our solution, *Athena*, over the open-source ONOS distributed SDN controller. We discuss how *Athena* enables security researchers and developers to make anomaly detection applications with a minimum of programming effort through its API abstraction layer. We also discuss generalized use of off-the-shelf strategies for driving network anomaly detection algorithm development and introduce a new SDN-specific anomaly.

Athena employs a distributed database and a clustered computing platform, which can deploy these detection algorithms across a large-scale distributed control plane. The *Athena* framework is designed to operate on existing SDN infrastructures, enabling operators to deploy it in a cost efficient manner. Our evaluations demonstrate that *Athena* can support well-known network anomaly detection services in an efficient manner, by scaling to a large-scale dataset from a large-scale datacenter-like physical network environment. *Athena* has been publicly released as an open-source project to the academic and SDN research community.

Acknowledgment

This work was supported by the ICT R&D program of MSIP/IITP, Republic of Korea [No. B0190-16-2011, Korea-US Collaborative Research on SDN/NFV Security/Network Management and Testbed Build]. This material is based upon work supported by the National Science Foundation under Grant Numbers 1446426 and 1642150. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of network and computer applications*, vol. 34, no. 1, pp. 1–11, 2011.

[2] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Scream: Sketch resource allocation for software-defined measurement," in *CoNEXT, Heidelberg, Germany*, 2015.

[3] ——, "Dream: dynamic resource allocation for software-defined measurement," in *Proceedings of ACM SIGCOMM 2014*.

[4] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013)*.

[5] K. Giotis, C. Argyropoulos, G. Androulidakis, D. Kalogeras, and V. Maglaris, "Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on sdn environments," *Computer Networks*, vol. 62, pp. 122–136, 2014.

[6] A. Kamisiński and C. Fung, "Flowmon: Detecting malicious switches in software-defined networks," in *Proceedings of the ACM Workshop on Automated Decision Making for Active Cyber Defense, 2015*.

[7] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "Flowsense: Monitoring network utilization with zero measurement cost," in *Passive and Active Measurement*, 2013.

[8] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting traffic anomaly detection using software defined networking," in *Recent Advances in Intrusion Detection*, 2011.

[9] K. Giotis, G. Androulidakis, and V. Maglaris, "Leveraging sdn for efficient anomaly detection and mitigation on legacy networks," in *Third European Workshop on Software Defined Networks (EWSDN), 2014*.

[10] R. Braga, E. Mota, and A. Passito, "Lightweight ddos flooding attack detection using nox/openflow," in *IEEE Local Computer Networks (LCN), 2010*.

[11] SDNSecurity, http://www.sdnsecurity.org/.

[12] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network anomaly detection: methods, systems and tools," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, 2014.

[13] M. Vallentin, V. Paxson, and R. Sommer, "Vast: a unified platform for interactive network forensics," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI 2016)*.

[14] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "ONOS: towards an open, distributed SDN OS," in *Proceedings of ACM HotSDN 2014*.

[15] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *Proceedings of OSDI*, 2010.

[16] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: flexible and elastic ddos defense," in *Proceedings of the 24th USENIX Conference on Security Symposium*. USENIX Association, 2015, pp. 817–832.

[17] A. Clemm, M. Chandramouli, and S. Krishnamurthy, "Dna: An sdn framework for distributed network analytics," in *IFIP/IEEE International Symposium on Integrated Network Management (INM)*, 2015.

[18] Y. Yu, C. Qian, and X. Li, "Distributed and collaborative traffic monitoring in software defined networks," in *Proceedings of ACM HotSDN*, 2014.

[19] L. Jose, M. Yu, and J. Rexford, "Online measurement of large traffic aggregates on commodity switches." in *Hot-ICE*, 2011.

[20] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *International Conference on Distributed Computing Systems (ICDCS), 2014*.

[21] POX, "Python network controller," http://www.noxrepo.org/pox/about-pox/.

[22] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," in *Proceedings of ACM SIGCOMM Computer Communication Review*, July 2008.

[23] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *2014 IEEE 15th International Symposium on WoWMoM*, 2014.

[24] FloodLight, "Open sdn controller," http://floodlight.openflowhub.org/.

[25] M. S. Kang, S. B. Lee, and V. D. Gligor, "The crossfire attack," in *IEEE Symposium on Security and Privacy*, 2013.

[26] M. S. Kang, V. D. Gligor, and V. Sekar, "Spiffy: Inducing cost-detectability tradeoffs for persistent link-flooding attacks," 2016.

[27] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013)*.

[28] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI 2012)*.

[29] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI 2012)*.

[30] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proceedings of ACM HotSDN 2012*.

[31] MongoDB, https://www.mongodb.com.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, 2010.

[33] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *arXiv preprint arXiv:1505.06807*, 2015.

[34] Jfreechart, http://www.jfree.org/jfreechart/.

[35] Apache, "Hama," https://hama.apache.org/.

[36] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*.

[37] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks." in *NDSS*, 2013.

[38] OnLab, "Master-performance and scale-out," https://wiki.onosproject.org/display/ONOS/Master-Performance+and+Scale-out.

[39] A. S. da Silva, J. A. Wickboldt, L. Z. Granville, and A. Schaeffer-Filho, "Atlantic: A framework for anomaly traffic detection, classification, and mitigation in sdn," in *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*.

[40] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "Payless: A low cost network monitoring framework for software defined networks," in *IEEE Network Operations and Management Symposium (NOMS) 2014*.

[41] J. R. Ballard, I. Rae, and A. Akella, "Extensible and scalable network monitoring using opensafe." in *INM/WREN*, 2008.

[42] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[43] Apache, "Cassandra," http://cassandra.apache.org/.