Enhancing OpenFlow Actions to Offload Packet-In Processing

Hamid Farhadi, Ping Du, Akihiro Nakao
The University of Tokyo
{farhadi, ping, nakao}@nakao-lab.org

***Abstract*—Software-Defined Networking (SDN) increasingly attracts more researchers as well as industry attentions. OpenFlow as a major API for SDN applies <match, action> rules to every packet. However, it only supports a few actions that are all predefined. We extend this limitation of OpenFlow and propose User-Defined Actions (UDAs) for SDN. We discuss usecases of UDAs and propose an architecture to realize UDAs. Using our architecture we conduct a series of tests. We indicate that our UDAs can elevate millisecond-scale running time of current proposals to nanosecond-scale (including proposals from northbound applications of SDN community and virtual appliances of Network Function Virtualization or NFV community). Also, regarding ease of programmability, we show that our proposal decrease the lines of code of by 72.9% and 79.3% compared to implementing the same functionality as a northbound application and as a standalone middlebox, respectively. In addition, we extended OpenFlow to support UDAs and implemented a few sample UDAs.**

I. Introduction

OpenFlow is a wide-spread SDN API. This API installs a set of *<match, action>* rules on network switches. Any flow matching to the rule, receives the corresponding action. OpenFlow proposes a programmable control plane but a configurable-only data plane. That is, the user can write a program on top of the controller to perform a task such as load balancing. By the configurable data plane we refers to the TCAMs that are only able to match flows against a predefined set of fields and the OpenFlow enabled switch (i.e., OpenFlow data plane) can be configured to execute some of the predefined actions on packets. So, OpenFlow has two major shortcomings:

- *Predefined flow field-matching*: OpenFlow defines a set of fixed predefined protocols and fields and instructs the data plane to match all the packets against them.
- *Predefined actions*: OpenFlow defines a small set of actions including forward, drop and meter to be executed on packets. The set is not extensible or programmable.

In this paper we relax the latter limitation of OpenFlow and propose User-Defined Actions (UDA) to increase the flexibility of current SDN definition. Our contribution in this paper are two folds. First, we show usecases of UDAs and propose a programmable architecture that extends the traditional SDN definition to support UDA. Second, via our evaluations we show the feasibility of UDAs in terms of throughput and their effectiveness in terms of ease of programmability. The rest of the paper is organized as follows. After discussing use usecases of UDAs we explain our proposed architecture following by evaluation results and concluding remarks.

II. User-Defined Action Usecases

The main reason behind the need for user-defined actions is to offer a more flexible SDN data plane compared to current limited and hardware-centric SDN data plane. The main challenge in front of this is the tradeoff between flexibility and performance. In this paper, we discuss this tradeoff and evaluate the feasibility of providing such a flexibility using real world experiments. In addition, there are other reasons that make user-defined actions interesting for the community.

In the OpenFlow context (and to some extent in the current SDN context) a *northbound* application is implemented on top of controller and calls the controller to perform a task. In contrast, *southbound* means the layer consist of switches and forwarding plane. We believe data plane programmability is as important as that of control plane programmability. However, there are less work from the community in this area compared to OpenFlow model that advertises control plane programmability. Todays, northbound applications are considered as hot topic(e.g.,[1]). However, giving examples of useful applications in this area is not a trivial task as there are several limitations in the control plane.

The major limitation in the control plane is the lack of access to the packet payload. That is why it is infeasible to have applications that filter more than a bitstream match (i.e. the OpenFlow classification model) on the packet. For example, an Intrusion Detection System needs to look at the whole packet and even buffer a couple of packets to detect an intrusion while sending packets to the controller for intrusion detection is infeasible because of performance considerations. With current proposals from Network Function Virtualization (NFV) community, all data plane related tasks should be executed in dedicated boxes that are possibly getting advantage of virtualization technology. While this promising approach is beneficial in many ways, it can be a limiting factor as well. Although for heavy tasks (such as running a heavy IDS) we may need an external box, in case of light small tasks that come handy in networking, making a separate box is not a reasonable approach. Current SDN proposal only offers northbound interface to realize such applications.

To support our argument, we implemented some examples of northbound security applications from a related work [1] in three deferent architectures to see the difference. In particular,

Copyright IEICE - Asia-Pacific Network Operation and
Management Symposium (APNOMS) 2014

TABLE I. Comparison of overhead in northbound (i.e., FRESCO[1]) applications versus southbound user-defined actions versus virtual appliance middleboxes

	SDN overhead (msec)		NFV overhead (msec)
	Control Plane [1]	**Data Plane (UDA)**	**Virtual Appliance**
PortScanner Detector	7.196	0.000001	0.001280609
BotMiner Detector	15.421	0.000004	0.001630215
P2P Plotter	11.775	0.000004	0.001312178

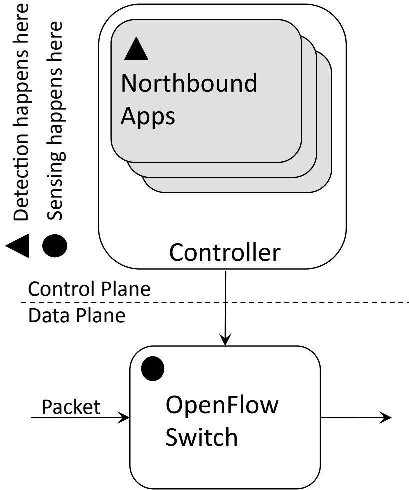

Fig. 1. Northbound application overhead evaluation architecture

once as UDAs in the data plane (i.e. southbound) and once in the form of virtual appliances as Network Function Virtualization (NFV) components and compare it with the northbound application architecture proposed in [1]. We discuss more implementation details in Section IV.

Specifically, we implemented three actions; Port scanner detector and BotMiner Detector and P2P Plotter based on the descriptions in [1]. Port scanner detector looks for repeated attempts to connect to a closed port on a system (i.e., the victim) from another system (i.e., the attacker). This kind of detectors help to find worm-infected machines that are scanning the network for new victims. BotMiner detector is another simple action that clusters network nodes based on the output of port scanner detector to detect bots via co-clustering of nodes producing network anomalies. Finally, P2P Plotter is a malware detection service that looks for two characteristics to detect peer-to-peer malware. First, P2P malware usually produce less amount of traffic compared to benign P2P client software. Second, P2P malware nodes have less churning rate. That is, P2P malware nodes commonly establish longer connections compared to benign clients. P2P Plotter co-clusters the nodes that exhibit both features.

Table I illustrates the difference between the overhead of running three applications as UDAs on the data plane (i.e., southbound) versus the control plane (i.e., northbound) and as a virtual appliance (i.e., NFV). To understand the difference between these three architectures we explain each of them briefly.

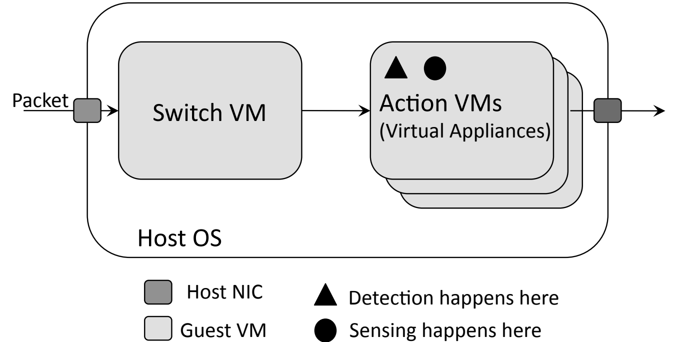

Fig. 2. NFV application overhead evaluation architecture

A. Northbound (or Control Plane) Applications

The control plane implementations show the overhead delay of running elements on FRESCO [1] framework. FRESCO is a Click [2] like framework to develop elements for security applications at northbound. Figure 1 indicates the architecture of northbound application. In a traditional OpenFlow inspired SDN architecture, the data plane consists of an open flow enabled switch that depending on the specific OpenFlow version support the related features. On the control plane side, there is a programmable OpenFlow controller with a set of applications running on top of the controller. Different controllers may support different languages and features for the guest (i.e., northbound) applications. A key point to understand the underlying reason behind the considerable difference in overheads in Table I is to note where the sensing and detection happens physically. The small black triangle shows that the detection mechanism happens on the controller based on the sensing information captured by data plane (denoted by a small black circle). Therefore, the sensing data goes via the wire from switch to the controller that produced some overhead.

B. NFV Appliances

Figure 2 indicates the architecture of NFV application we use for evaluation. The application is located within a guest virtual machine. The switch running the application is also running in another virtual machine and the connection among them is via the virtual network in the host operating system. Table I shows NFV applications are almost 100 times faster than control plane applications. In fact, it depends on the architecture. Since we put both the switch and the virtual appliance within a single virtual network, NFV shows a considerable performance increase. However, in a typical NFV scenario, switch is located on a separate machine from virtual appliances. In the latter case, the difference between NFV and northbound application is less as we move from virtual network

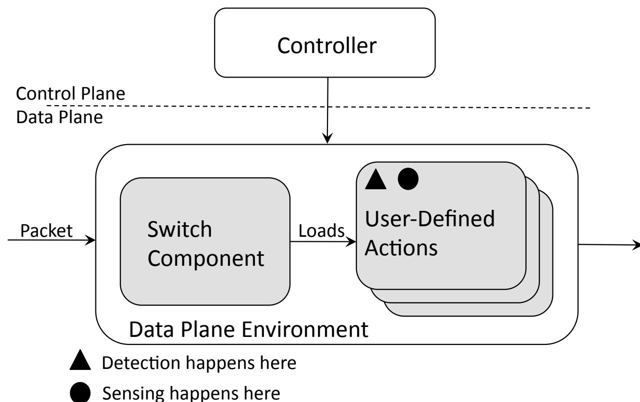

Fig. 3. UDA overhead evaluation architecture

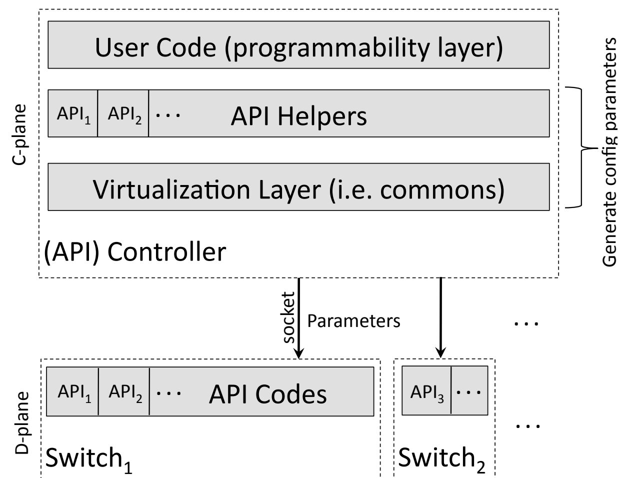

Fig. 4. UDA/API control and data plane architecture

to a physical network that connects machines hosting switch and appliance VMs.

C. User-Defined Switch Actions

Figure 3 illustrates the architecture we use for UDA evaluation. The figure shows a data plane environment that settles switch and actions as different components. We use Click Modular Router to implement this architecture. Hence, the switch as well as UDAs are all Click elements. As it is shown in Figure 3, we do not connect actions and switches in the serial manner (as the regular use case in Click configuration files). Rather, we implemented actions as a passive-like (or plugin-like) element and include it in the switch code in a way that the switch can load the external user-defined action element to execute it. We explain more implementation details in Section IV. The overhead of UDA deployed on the data plane is in nanosecond scale while the others are in microsecond scale. That is because, first, sensing and detection happens physically at the same machine and no information is transferred over physical or virtual network and second, the switch and UDAs are all on a single shared platform in contrast to NFV case in which the packet should go through the whole protocol stack of virtual appliances. We explain more details about UDA architecture in Section III. In conclusion, data transmission and VM overhead are two major limitations of alternative methods that make UDA an interesting candidate for applications.

III. SYSTEM ARCHITECTURE

In this section we introduce the architecture that enables UDAs to be defined and programmed. Figure 4 depicts the overall architecture of our proposal. Following current popular SDN architecture, our system consists of two physically separated planes: control plane (at the top of the figure) and data plane (at the bottom of the figure). Before digging into our design objectives we define three main roles involved in the system. We consider an environment to publish a typical (commercial) service for end-users such as online video streaming service.

- *Network/Infrastructure provider*: Is the entity that provides the hardware and cabling infrastructure plus the programmable SDN environment. Infrastructure provider gives the user, a programmable environment including control and data plane programmability features to develop, test and run arbitrary networking software. We do not define the exact form of such an environment since depending on the application different technologies can be used for this objective. An example technology to provide a programmable environment is network virtualization where each user has one slice and a set of resources fully isolated from other slices. There are a couple of ways to implement this kind of environment in the literature (e.g., OpenTag [3]) that are out of the scope of this paper. Our focus in this paper is how the infrastructure provider designs the a programmability feature of data and control planes particularly for UDAs. So, we leave other concerns such as isolation and management issues for future work.
- *User (or service provider)*: Is the entity that is going to program the data and control planes to publish a service for end-user. Needless to say, infrastructure provider and user can be the same entity.
- *End-user (or service user)*: Is an individual who consumes the service provided by the service provider (or user).

After defining the terminology we use to explain our architecture, we move on to defining our design objectives as follows:

- To keep the current architectural benefits of SDN caused by separation of data and control plane and related abstractions.
- To extend SDN data plane to support programmability. That is, using our architecture, the user is able to define arbitrary API or UDA and use it.
- To extend SDN control plane to support defining new APIs such as UDAs.

The data plane of our architecture consists of switches that can locate data plane API codes. Switches may have different types of APIs and each API may have its own set of parameters. Control plane can modify API parameters on the data plane (i.e., switches). The control plane is made up of three layers; The *Virtualization Layer* is responsible for common tasks including basic functions (e.g., connectivity) and gathering the information from the switches about available APIs and making an abstraction of network topology. The *API Helpers* layer are procedures that implement methods each API needs on the control plane. For example, if an API should read some information from the data plane and then execute some calculations on the data gathered from the network on the controller, then corresponding API helper component is responsible for such a task. Put it other way, each API function spans over both data and control plane. Hence, a part of the API code is physically on the data plane (indicated as API codes in the figure) and another part is located on the control plane (indicated as API helpers in the figure). Based on the user code, API helpers along with the virtualization layer compute and install appropriate parameters on the data plane. Finally, the *User Code* layer includes the user code on the controller.

The most important difference between our architecture and traditional SDN controllers is that we locate the API codes on the data plane. To clarify the issue, we use OpenFlow as an example of traditional SDN API which spans over both data plane (i.e., OpenFlow enabled switches) and control plane (i.e., OpenFlow controller). At the data plane side, OpenFlow uses hardware-centric data plane components such as TCAM that implements the data plane side of the API logic on the switch including the OpenFlow actions. In contrast, our architecture fosters a software-defined data plane that can include multiple user-defined APIs. We realize and evaluate UDAs as an example API in this architecture. So we look at UDAs as special APIs that can be loaded in a plugin-like fashion to the switch to be executed accordingly. The switch triggers execution of UDAs in a similar way to the traditional SDN. That is, once a flow is matched against a row in the switch forwarding table, a set of actions defined by the user for that specific flow are executed on every packets of that flow.

A. Ease of Programmability

In this paper, we propose a solution for programmability of the data plane. Accordingly, an important factor to show the effectiveness of the programmability is *ease of programmability* which refers to how easy it is to develop an arbitrary program using the proposed solution or method. To support our proposal, we implemented two anomaly detection algorithms from a related work (i.e., [4]) to compare ease of programmability in different architectures. Specifically, we implemented Rate Limit [6] and Threshold Random Walk with Credit Based Rate Limiting (TRW-CB) [5] algorithms. The TRW-CB is a method to detect infected hosts by worms that are already started scanning other nodes. It assumes the number of successful connection attempts from non-infected nodes is higher than infected nodes. The TRW-CB applies a likelihood ratio test to classify nodes using a queue of TCP SYNs for every node that is not received the SYNACK response yet. In case of time out or TCP RST message for any queued SYN, the likelihood ratio of that specific host will be incremented by the algorithm. Similarly, Rate Limit algorithm assumes infected nodes try to connect to a large number of nodes in a short span of time. It keeps track of recent attempts for connection from all the hosts in the network and matches new attempts against the recent attempts list. Connection from nodes that perform many attempts are delayed in a queue of a limited size. Once the queue reaches a threshold size, the source host will be considered as an infected node.

Table II compares the size of source code (i.e., LoC) to implement aforementioned detection methods in different architectures. The control plane implementations refer to Python language code written on top of the NOX controller. The middlebox implementations are in C language. By the term middlebox we refer to implementing the detection method on a standalone machine that sends/receives packets to/from its physical NICs. In contrast to typical Click element usecases (as we mentioned in Section II-C) we implemented UDAs as a passive-like Click elements and include them in the switch code in a way that the switch can load the external UDA element to execute it. The reason behind such an implementation is ease of programmability. Since the UDA is implemented in a separate element, the code is cleaner and more extendable. Furthermore, including the action element in the switch code takes a few lines of code so that the extra work caused by providing programmability is reasonable. This not only keeps the action code clean, but keeps the switch code simple and clean even after adding UDAs. Since we use Click to implement our UDAs, we calculate the LoC using the summation of Click configuration file and the LoC of the element source code. Note that we do not consider the LoC of the Click Router itself similar to NOX case that we only include the LoC of the program written on top of NOX. Moreover, Our controller is a standalone program that connects to the switch using sockets. It can retrieve the list of online switches and the catalog of available APIs from every switch. Using these information user can write programs. Currently, our controller supports only C++ programmability. We build a Python wrapper over some functions of the controller so that the user can write Python script to program the controller and manipulate parameters on data plane. Table III indicates the comparison of ease of programming using a northbound programming framework and UDA implemented at southbound. FRESCO as a northbound application development framework is made for easier application development at controller. However, comparing UDA and FRESCO using the three sample applications, we can see almost similar lines of code in both solutions. The minor difference we see can be the result of difference programming languages. In fact, Python scripting language (used in FRESCO) can reduce the code size compared to C++ programming language. We believe the majority of the difference we see in Table III is caused by the difference between Python and C++. Hence, we consider similar lines of code using two different approaches eventhough FRESCO is an additional framework on top of the controller. If we count the LoC of the FRESCO framework itself (excluding the controller LoC), the result will be different and UDA lines of code will be less than FRESCO applications. In conclusion, using UDAs implemented on data plane we can reduce the LoC by 72.9% and 79.3% compared to implementing the same functionality on control plane and as a standalone middlebox, respectively.

TABLE II. COMPARISON OF EASE OF PROGRAMMABILITY IN NORTHBOUND [4] APPLICATIONS VERSUS SOUTHBOUND UDAS VERSUS MIDDLEBOXES

	Lines of Code (LoC)		
	SDN		Middlebox [4] (C)
	Control Plane (i.e. NOX in Python) [4]	Data Plane UDA (C + config)	
TRW-CB [5]	741	196 (181+15)	1060
Rate Limit [6]	814	225 (205 + 20)	991

TABLE III. COMPARISON OF EASE OF PROGRAMMABILITY IN NORTHBOUND (I.E., FRESCO [1]) VERSUS SOUTHBOUND (I.E., UDAS)

	Lines of Code (LoC)	
	Northbound implementation via FRESCO (Python + config) [1]	Southbound implementation via UDA (C + config)
Port scanner Detector	229 (205+24)	264 (249 + 15)
BotMiner Detector	352 (312 + 40)	575 (548 + 27)
P2P Plotter	259 (227 + 32)	302 (281 + 21)

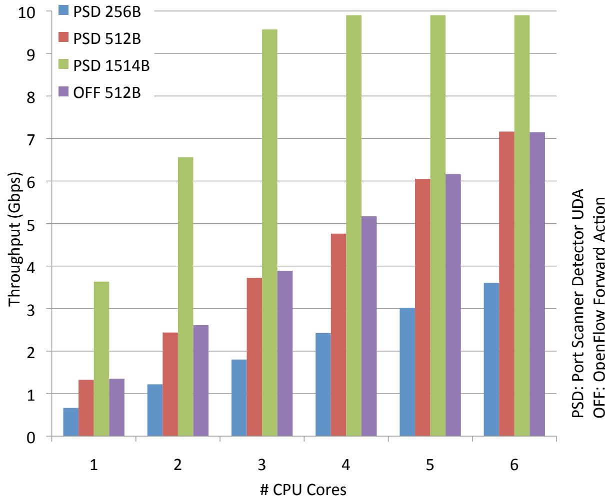

Fig. 5. OpenFlow extended to support UDAs

IV. THROUGHPUT EVALUATIONS

In Section II we reviewed some PC-based experiments to show per packet overhead using different architectures. In this section, we consider testing UDAs under higher loads to see how it performs. Particularly, we extend the OpenFlow to support such a feature and measure the overhead of UDAs. We use the same UDAs as in experiments in Section II (i.e., Botminer Detector, Portscan Detector and P2P Plotter). The experiment setup we use in all evaluations is as follows. We use Xena packet generator to generate 10 Gbps traffic and send it to FLARE [7] switch that hosts our UDAs. FLARE switch is a programmable switch using Click environment and multicore CPUs. It has a couple of SFP+ ports and provides a Linux and Click environment for network research. For more details on FLARE please refer to [7].

A. OpenFlow Plus UDAs

For OpenFlow experiments we use our software OpenFlow implementation on FLARE. Since FLARE uses the Click Modular router environment we implement all actions as Click elements. In case of OpenFlow, we use the original open source and publicly available OpenFlow implementation as a static library linked against the OpenFlow Click element and use library functions in the element and extend it to support UDAs. Figure 5 illustrates the throughput of running UDAs on OpenFlow using different packet sizes. We compared PSD with OFF. We use one to five processor cores to show the throughput is linearly increasing while we increase the number of cores.

B. More Complex UDAs

Compared with portscan detector UDA, the botminer detector and P2P Plotter UDAs are more complex since they look for specific aspects of the traffic and co-cluster the results to make the decision. For both of them we use multiple Click elements for implementation. Therefore, they have lower throughput because of the overhead of using multiple elements. In case of portscan detector we applied some optimizations to get the results presented in Figure 5. Particularly, before optimization, we used the CheckIPHeader Click element to set the IP address annotation on the pact header to prevent segmentation fault on our portscan element that was checking IP header on every packet. So, we embedded the annotation functionality within the portscan element. As the result, removing the overhead of using additional element, we reduced the overhead of portscan detector element from 0.000004 milliseconds to 0.000001 millisecond. In the same way, we can reduce the overhead of botminer detector and P2P Plotter UDAs since they are using the Counter and DelayUnqueue elements. Rather, we use them for another experiment. That is, how the throughput increases with the increase of the number of processor cores on heavier UDAs. Table IV indicates that the throughput follows a liner increase even up to 11 cores. As FLARE provides more than 30 cores, we can keep experiencing the same linear increase of performance which we exclude from the figure for brevity. Also, we excluded the results from one to five cores for briefness. We conclude that even for heavier tasks we can consider UDAs using more number of processor cores.

C. Portscan Detector UDA Experiment on Attack Trace

Most of the experiments we discuss, focus on performance of UDAs. For completeness we conducted a test on an attack trace from [4] to show if the UDA we made really detects attacks or not. Obviously, as we are not proposing a detection algorithm, our measurement is not based on popular intrusion detection metrics such as false negative and false positive. Instead, we select a portion of the detection attack along with the detection time. Figure 6 shows the detection time of a

TABLE IV. Overhead of heavy User-Defined Actions using extended OpenFlow

	# Processor cores	6	7	8	9	10	11
OpenFlow + UDAs (Gbps)	**BotMiner Detector**	2.3	2.9	3.1	3.5	3.9	4.3
	P2P Plotter	2.4	2.8	3.2	3.6	4.0	4.4

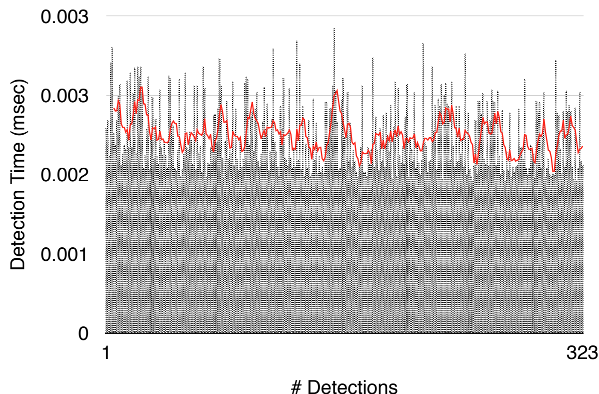

Fig. 6. Per attack detection time of Portscan Detector UDA on the attack trace from [4]

random set of more than 300 detections the UDA fired on the attack trace. The trace consists of about 2 million packets attacking from three computers to three targets. The red line shows the trend of detection time which illustrate reasonable fluctuation we usually see in a software and we do not see an unusual increase or decrease in detection times.

V. Related Work

To our best knowledge, there is a limited attention to user-defined actions in the literature. However, there are some works that use labeling and tagging approaches as well as proposing software solutions for SDN data plane. The only work (we are aware of) which focuses on user-defined actions is [8] in which authors propose a design of a chip as a replacement for TCAM. They propose the Reconfigurable Match Tables (RMT) model, a new RISC-inspired pipelined architecture for switching chips, and identify the essential minimal set of action primitives to specify how headers are processed in hardware. RMT allows the forwarding plane to be changed in the field without modifying hardware. As in OpenFlow, the programmer can specify multiple match tables of arbitrary width and depth, subject only to an overall resource limit, with each table configurable for matching on arbitrary fields. RMT allows the programmer to modify all header fields much more comprehensively than in OpenFlow. The paper describes the design of a 64x10 Gbps ports switch chip implementing the RMT model and claims that flexible OpenFlow hardware switch implementations are feasible at almost no additional cost or power [8]. However, even though the design looks promising, it is never implemented in the real world to prove the claims. Such works shows the community is skeptical about the capability of software controls. In this paper, we argue that software controls are capable and sound for such purposes.

Another work that overlaps in problem domain with ours is [9]. Authors introduce a tagging architecture in which middleboxes export tags to provide the necessary casual context (e.g., source hosts or internal cache/miss state). SDN controllers can configure the tag generation and consumption operations using their API. These operations help bindings between packets and their origins as well as ensuring that packets follow policy mandated paths. Middleboxes may use tags to execute an action dynamically on the packet. This paper is different from ours in two folds. First, it more focuses on how actions are executed rather than ours in which we consider how they are defined. Second, it discusses actions as middleboxes in contrast to our target in which we study the feasibility of deploying actions within the same box as the switch similar to basic OpenFlow actions implemented using TCAM. For completeness of this section we can mention other related SDN technologies that can be combined with UDA. For example TagFlow [10] which is a flow based switching system. Similar to OpenFlow case we discussed in our experiments, UDAs can be combined with TagFlow as well.

VI. Conclusion

In this paper, we propose UDAs as an extension to current proposal from SDN community. We propose an architecture and an extended version of OpenFlow to support such a functionality in Click environment. We illustrated evaluations in terms of throughput and ease of programability. We believe the community should pay more attention to the programmability of the data plane in order to have a better SDN.

References

[1] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "FRESCO: Modular composable security services for software-defined networks," in *Proceedings of Network and Distributed Security Symposium*, 2013.

[2] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[3] R. Furuhashi and A. Nakao, "Opentag: Tag-based network slicing for wide-area coordinated in-network packet processing," in *IEEE ICC*, 2011.

[4] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting traffic anomaly detection using software defined networking," in *Recent Advances in Intrusion Detection*, 2011, pp. 161–180.

[5] S. E. Schechter, J. Jung, and A. W. Berger, "Fast detection of scanning worm infections," in *Recent Advances in Intrusion Detection*, 2004, pp. 59–81.

[6] J. Twycross and M. M. Williamson, "Implementing and testing a virus throttle," in *USENIX Security Symposium*, vol. 285, 2003, p. 294.

[7] Akihiro Nakao, "FLARE: Open Deeply Programmable Switch," in *The 16th GENI Engineering Conference*, 2012.

[8] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proceedings of the ACM SIGCOMM*, 2013, pp. 99–110.

[9] 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 *USNIX NSDI*, 2014.

[10] H. Farhady and A. Nakao, "TagFlow: Efficient Flow Classification in SDN," *IEICE Transactions on Communication*, 2014.