Enabling L2 Network Programmability in Multi-Tenant Clouds

Thomas Lin, Byungchul Park, Hadi Bannazadeh, and Alberto Leon-Garcia
Dept. of Electrical and Computer Engineering, University of Toronto
Toronto, ON, M5S 3G4, Canada
{t.lin, byungchul.park, hadi.bannazadeh, alberto.leongarcia}@utoronto.ca

***Abstract*—With the advent of SDN, efforts on network virtualization have accelerated. Many concepts have been developed to enable the coexistence of multiple logical network domains on a shared networking infrastructure while also ensuring isolation between them. In this context, we envision a multi-tenant cloud environment that grants users the ability to dynamically alter the behaviour of the underlying network, thus achieving a truer sense of *Infrastructure* as a service. In this paper, we propose a method for providing users in multi-tenant cloud environments open APIs for installing custom flows into the network at layer 2 (L2). Our proposed system ensures tenant isolation while detecting possible flow-level conflicts that can lead to tenancy violations and to problems in end-to-end reachability. We prototyped our system on a nationwide testbed environment, which showed that our proposed method reduces flow-level conflict detection time over existing proposals from ms to μs scale.**

I. Introduction

Network virtualization allows for the coexistence of multiple virtual networks on the same physical infrastructure while providing isolation between virtual networks. This helps network service providers to create heterogeneous network architectures for new types of services without having the limitations inherited from existing network infrastructures. The concept of network virtualization has existed since the late 1990s. For example, Tempest [1] introduced virtualization in ATM networks by splitting an ATM switch into multiple switchlets. While it is possible to realize network virtualization with existing technologies such as VLAN, MPLS, and overlay networks, these primitives do not provide a unified abstraction of the underlying network which is essential for complete network virtualization. To provide complete network virtualization, network infrastructures should be able to support arbitrary network topologies and network addressing schemes while providing a single unifying abstraction to configure the network [2].

With the advent of Software-Defined Networking (SDN), efforts on network virtualization have intensified. SDN refers to a network architecture where the data plane and control plane are decoupled and the forwarding devices are remotely managed by a logically centralized controller. The decoupled control plane may further enable network programmability for upper layers by providing an abstraction of the underlying network infrastructure to the services and applications that utilize the network. By taking advantage of SDN's programmability, multiple network hypervisors have been proposed [3], [4], [5], [6] that enable virtualization of SDN networks. Similar to a hypervisor for hosting virtual machines, a network hypervisor over SDN infrastructure allows for multiple concurrent control domains to co-exist with the illusion of ownership over the entire network.

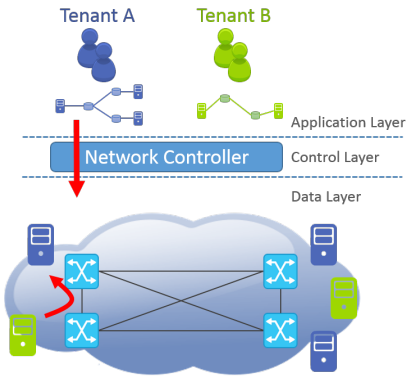


Fig. 1: Multi-tenancy violation occurs when a tenant's new flow rule request may impact another tenant's network traffic

In this paper, we will discuss how to utilize network virtualization technology in cloud environments to provide users of multi-tenant clouds SDN capabilities directly at layer 2. The enablement of programmability at such a low level will allow Future Internet researchers to directly deploy and test new protocols on the physical infrastructure itself. Enabling network programmability in a multi-tenant cloud environment is more complicated than the scenarios previously discussed in existing network virtualization literature, as they often assumed a single network administrative entity for the entire infrastructure. In an SDN-enabled multi-tenant cloud, multiple users can request new network flow rules to be created for their applications or services. Opening direct network programmability to users can make the infrastructure error prone. Figure 1 illustrates the problem of multi-tenancy violation, where a new flow rule inserted by a user of Tenant A redirects the network traffic from the resources of Tenant B. While the basic approach for solving this issue is analogous to network virtualization, other aspects such as authentication and authorization of user requests need to be considered in order to make it applicable to cloud environments.

We will also shed a new light on the flow conflict problem, which is directly linked to security, in SDN-based virtualized networks. We will present an approach for detecting possible flow-level conflicts at the controller level in advance before installing flows into the network. We have validated our proposal by implementing a prototype in the Canadian nationwide SAVI testbed (a multi-tier SDN-enabled multi-tenant cloud).

978-3-901882-89-0 @2017 IFIP

TABLE I: Example flow table

ID	Ingress Port	Eth Src	Eth Dst	Eth Type	VLAN ID	IP Src	IP Dst	IP Proto.	IP ToS	Src Port	Dst Port	Actions
F_1	*	*	FF:FF:FF:FF:FF:FF	0x0800	*	*	*	*	*	*	*	action 1
F_2	*	6E:CF:A6:93:40:E1	*	0x0800	*	*	*	*	*	*	*	action 2
F_3	*	6E:CF:A6:93:40:E1	*	0x0800	*	*	*	TCP	*	*	*	action 3
F_4	*	*	*	0x0800	*	*	*	TCP	*	*	22	action 4

The rest of this paper is organized as follows. Section II provides background information for describing our goal and reviewing related work. In section III, we present our method for detecting tenant and resource violations in SDN-enabled multi-tenant clouds, as well as describe the flow-level conflict detection algorithm. Section IV describes our prototype implementation on a nationwide testbed and presents the performance evaluation results of the implemented system. Finally, conclusions and future work are presented in section V.

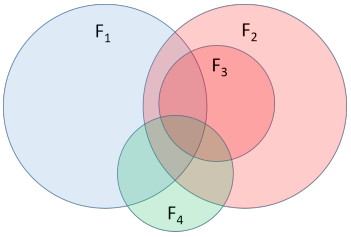


Fig. 2: A logical view of flow entries in the flow table (Table I)

II. Background & Related Work

In this section, we provide background information required to describe our problem and we review previous works related to our proposal.

A. OpenFlow and Flow Conflict

OpenFlow [7] is the most widely accepted and deployed SDN protocol. In the OpenFlow architecture, OpenFlow controllers offer various types of northbound APIs to network applications or services. In this setting, multiple services can install flows through the OpenFlow controller and this can cause flow conflicts to occur. In [8] and [9], a flow conflict is defined as the situation when an incoming packet matches with more than one flow entry in the flow table.

In this work, we define a flow conflict as a situation where a newly installed flow intersects with any existing flow entry in the flow table. Since wildcard expression for a header field is possible, a flow entry can be considered as a geometric object in n-dimensional space, where n is the total number of header (or match) fields (e.g., n = 12 in OpenFlow v1.0 and n = 39 in v1.3), and the surface is determined by the header values. If the geometric object for a new flow entry overlaps with any existing objects, it causes one or more flow conflicts.

Table I shows an example OpenFlow table with four flow entries, where each flow conflicts with one another. For example, F_1 and F_2 have a common space that may result in a conflict (e.g. an incoming packet with source MAC address 6E:CF:A6:93:40:E1 and destination MAC address FF:FF:FF:FF:FF:FF). Figure 2 illustrates the logical relations between the four flow entries in Table I with a Venn diagram.

Once a flow conflict occurs, incoming packets may match multiple flow entries, and consequently, unwanted actions may be applied to the packet. It is highly possible that this conflict will create end-to-end reachability problems, security violations, and open network vulnerabilities. We have tested three different OpenFlow-enabled switches (Open vSwitch v2.3.1, Pronto 3920, and HP 5900v), and all of these OpenFlow switches simply overwrite the existing flow entry if the new flow has identical header match fields, as well as resets the counters used for analyzing flow statistics. Inadvertently resetting flow counters may negatively affect any applications or services dependent on accurate flow statistics. Therefore, an efficient flow conflict detection mechanism is required to maintain the configuration integrity of the network.

B. Network Hypervisors

FlowVisor [3] was the first OpenFlow-based network virtualization hypervisor. FlowVisor allows multiple logical networks to share the same OpenFlow networking infrastructure by essentially acting as a transparent proxy that resides between controllers and switches, and intercepts the OpenFlow messages between them. The benefit of this architecture comes from its transparency, meaning that no modification to controllers or switches are required. FlowVisor can ensure tenant isolation by examining, re-writing, and policing OpenFlow messages passing through it.

In order to provide isolation, FlowVisor defines the concept of a FlowSpace which is a logical geometric space of possible packet headers. Each slice is a set of flows that spans certain areas of entire FlowSpaces. FlowVisor achieves isolation by preventing overlap between FlowSpaces from different slices. After FlowVisor, other network hypervisors were introduced [3], [4], [5], [6]. OpenVirtex [4] extended network virtualization to consider not just FlowSpace, but also address virtualization and control function virtualization. AutoSlice [5] focused on the automation of deployment and operation of SDN slices on top of shared network infrastructures. While providing additional features, such as different levels of abstraction and the isolation address spaces, their basic operations for tenant isolation are quite similar to FlowVisor's.

This paper focuses on bringing these techniques into multi-tenant cloud environments in order to provide low-level network programmability directly to cloud users. Current public cloud services such as Amazon Web Services, Google Cloud Platform, and Microsoft Azure do not expose SDN network programming APIs to users. We believe enabling network programmability in multi-tenant cloud environments is a very essential step in allowing users to create innovative services and applications.

III. Design and Methodology

The main challenge of our work lies in enforcing the necessary isolation between tenants when users are granted programmatic access via APIs to control the network, while providing users the illusion of a single network infrastructure.

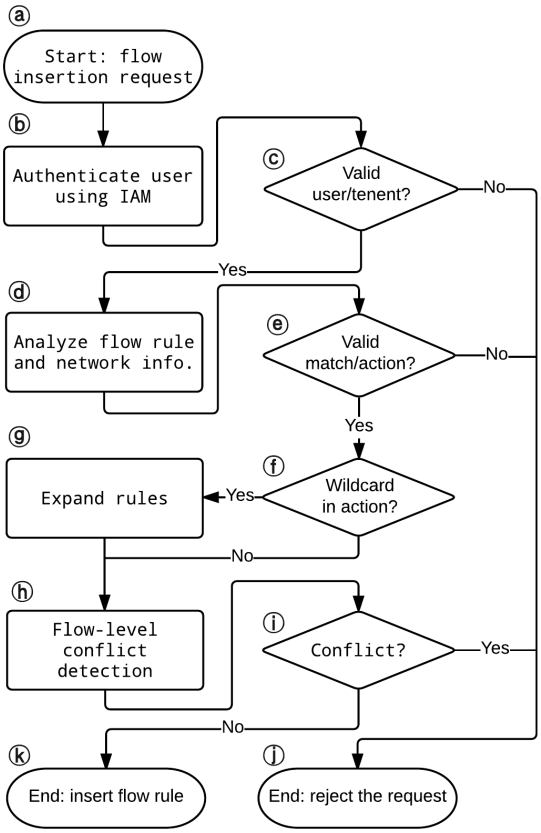


Fig. 3: Flow-chart of overall process for detecting multi-tenancy violations and flow-level conflicts

In this section, we present our method for detecting tenant and resource violations, as well as an overview of the flow-level conflict detection algorithm. The overall process is illustrated in Fig. 3, and each step will be explained in the rest of this section.

A. Network Flow Model

We begin by discussing the network flow model exposed to the users when they request new flow installations. As OpenFlow has received much attention in the network research community for the better part of the past decade, we decided to use a similar model. Our model is a restricted subset of OpenFlow's flow model, which is defined by a set of matching header fields and a set of actions [7], [10].

B. Tenant and Resource Isolation

As one of the key goals of this work is to enforce tenant isolation, we must first understand how isolation is achieved within the context of a given cloud infrastructure. This will vary depending on what level of abstraction is available to users via the APIs. For our work, we define our management policies for enforcing tenant isolation based on the assumption that the APIs will offer users the ability to install flow rules into the network starting at layer 2 using Ethernet. Given this assumption, we are able to define a set of basic checks to be performed when a user requests a new flow rule to be installed into the network.

1) Authentication and Authorization: All requests to install or modify network flows must pass through authentication and authorization prior to being implemented in the physical network. At minimum, requests should contain the user's identification as well as specify the tenant to which the request is intended. When the cloud's API controller receives a request (Fig. 3.a), it first liaises with the cloud's Identity and Access Management (IAM) system to verify the user's identification credentials, and ensures that the user has the appropriate role in the tenant to make the requested changes, thus completing authentication and authorization of the user (Fig. 3.b-c).

2) Authorizing the Flow Request: After the user has been authenticated and authorized, the API controller must further parse and process the flow request in order to ensure it does not violate tenant isolation (Fig. 3.d). Since our basic flow model is comprised of a match and associated actions, we must authorize both. This protects against cases where a malicious user may create flows that match on endpoint addresses not owned by their tenant, or cases where they create actions with the potential to divert traffic to destinations in other tenants. The controller checks the matches and actions (Fig. 3.e) to ensure adherence to the following policies:

- In the match fields, flow requests must always specify at least a source or destination MAC address which belongs to the tenant. If one of the fields is left unspecified as a wildcard, it will be automatically expanded into multiple rules that enumerates the L2 addresses owned by the tenant (Fig. 3.f-g);
- Actions that set either the source or destination MAC address to one not owned by the tenant shall be denied;
- Actions are free to modify source or destination IP addresses so long as the resulting flow still includes at least a source or destination address that is owned by the tenant.
- Flows can only be installed within a limited range in priority levels. This protects the basic operations of the cloud infrastructure by ensuring that users cannot override vital flow rules, while granting users some flexibility in creating unique networking applications and services which utilize the priority field.

C. Flow-level Conflict Detection Algorithm

The final step in authorizing a new flow installation request is to check if the addition of that new flow would result in any conflicts with existing flows (Fig. 3.h-i). This final check prevents flow conflicts within a single tenant, which may occur when a single user mistakenly installs conflicting flows, or when multiple users within the same tenant are installing flows into the network.

1) Description: The conflict detection algorithm we utilize considers each header field in the match separately. For each header field in the user's requested flow, it identifies the set of existing flows where the same field has a matching value (i.e. a set of potentially conflicting flows). For example, if there are 12 header fields to consider, we will obtain 12 sets of potentially conflicting flows. We consider a conflict to exist if and only if there is an existing flow that is common throughout all sets of potentially existing flows. We note that this same approach was utilized in FlowVisor for storing and searching

the existing FlowSpace entries for matches when a controller attempts to install a new rule.

This algorithm is divided into two parts: storing flows and detecting conflicts with existing flows. When a new request to install a flow is received, it is first checked for conflict. Only when no conflicts are found is the flow authorized, and thus a copy of it is inserted into the cache of existing flow entries. We use the term *cache* here to refer to the collection of existing flows, but realistically it can be any type of data store. The pseudo-code for both inserting a new flow into the cache, and detecting conflicts upon a new request, are shown in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1: Pseudo-code for new flow insertion

procedure InsertFlow
input : flow, flow = $\{f_1, f_2, \cdots, f_n\}$
f_i is ith field of OpenFlow
output : flow id (negative value means a flow conflict)
variables: bitarray indicates all flows whose ith field matches with f_i
functions: id_generator() generates a unique ID where ID $\in \mathbb{Z}$,
$Hash_i$ is a hashing function for ith field

```
1  begin
2      if DetectConflict(flow) is true then
3          return −1
4      end
5      else
6          uid ← id_generator()
7          for 1 ≤ i ≤ n do
8              bitarray ← Hash_i (f_i)
9              bitarray[uid] ← 1
10         end
11         return uid
12     end
13 end
```

Algorithm 2: Pseudo-code for flow conflict detection

procedure DetectConflict
input : flow, flow = $\{f_1, f_2, \cdots, f_n\}$
f_i is ith field of OpenFlow
output : Boolean
variables: bitarray indicates all flows whose ith field matches with f_i
functions: $Hash_i$ is a hashing function for ith field

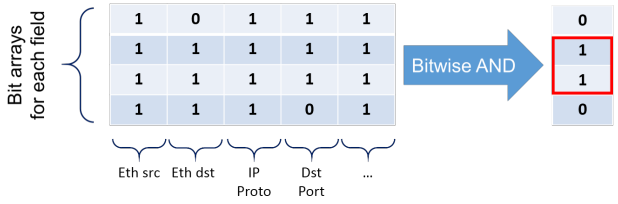
```
1  begin
2      initialize bitarray                    // set all bits to 1
3      for 1 ≤ i ≤ n do
4          bitarray &= Hash_i (f_i)           // bitwise AND operation
5      end
6      if bitarray is all 0's then
7          return false
8      end
9      else
10         return true
11     end
12 end
```

A functional example of this conflict detection methodology is illustrated in Fig. 4. Figure 4(a) shows a sample cache of four existing entries, F_1 to F_4, with a new flow F_{new} attempting to be added. The binary table in Fig. 4(b) shows the

ID	Eth src	Eth dst	IP Proto	Dst Port	...	Actions
F_1	*	FF:FF:FF:FF:FF:FF	*	*	*	A_1
F_2	6E:CF:A6:93:40:E1	*	*	*	*	A_2
F_3	6E:CF:A6:93:40:E1	*	TCP	*	*	A_3
F_4	*	*	TCP	22	*	A_4
F_{new}	6E:CF:A6:93:40:E1	00:25:90:88:55:0C	TCP	80	*	A_{new}

※ We assume that all other fields are wildcards (*)

(a) Insertion of a new flow F_{new}



(b) Detected F_{new} conflicts with F_2 and F_3

Fig. 4: Illustration of conflict detection algorithm

conflict detection algorithm at work: each column represents one of the header fields, and each row represents one of the four existing flow entries. For a cell in a given row and column (i.e. flow and header field), a value of *1* means that it can match with the corresponding header field of F_{new}, and *0* means it cannot match. Thus, the binary table represents all the potential matches for each field of F_{new}. A horizontal bit-wise AND operation is performed to obtain the final bit array, and it can be seen that the second and third entries remain *1*, thus indicating F_2 and F_3 conflicts with F_{new}.

In regards to memory consumption, we note that this is directly affected by the size of the bit arrays. The size of the array essentially represents the maximum number of flows within a given network. Thus, there is a design trade off to be considered during implementation if a dynamic structure or a static structure should be used for the bit arrays, which will affect the run-time speed and memory consumption.

2) Time Complexity: As seen in Algorithm 1, the two primary data structures in use are hash tables and bit arrays. Insertion and lookup in hash tables have a theoretical average time complexity of *O(1)*. Furthermore, insertion and lookup in arrays also have a theoretical average time complexity of *O(1)*. Similarly, for the conflict detection algorithm in Algorithm 2, we see that the main data structures in use are also hash tables and bit arrays. Thus, the theoretical average time complexity for flow insertion and conflict detection are both *O(1)*. Note that the bit-wise AND operation, as well as the bit array initialization operation, are both constant time operations so long as the length of the bit arrays are static. If the size of the arrays vary over time (e.g. using some type of dynamic data structure), then these operations cannot be constant time.

IV. Prototype Implementation

In this section, we will describe our prototype implementation of this conflict detection method on the SAVI testbed, a multi-tenant cloud in operation since late 2012. In addition,

we show performance evaluation results to provide evidence that our implementation achieves constant time lookup and conflict detection.

A. SAVI Testbed

The *Smart Applications on Virtualized Infrastructure* (SAVI) project is a Canadian research network whose aim is to explore future applications and application platforms. Key to the project was the development of an experimental multi-tier, multi-tenant testbed for SAVI researchers to deploy and test novel future architectures, protocols, applications, and services. The current SAVI testbed architecture is composed of OpenStack [11] components as well as components built in-house, and supports a wide range of heterogeneous resources including traditional virtualized computing infrastructure (i.e. VMs and storage), as well as unconventional resources (e.g. GPUs and FPGAs) and network elements (e.g. switches, Wi-Fi access points, software-defined radios, etc.). In support of the effort to pursue research on Future Internet protocols, the SAVI testbed's network is fully OpenFlow-enabled.

In an early iteration of the SAVI testbed, FlowVisor had been integrated into the system, which enabled the delegation of certain *slices* of the testbed network to external OpenFlow controllers. However, our operational experience found that FlowVisor itself was a bottleneck and became a single point of failure. A later iteration of the testbed attempted to compensate for these issues by deploying FlowVisor in a distributed fashion [12]. Ultimately, FlowVisor was retired for two main reasons: the abandonment of the FlowVisor project by its maintainers, and the realization that many network researchers were inexperienced with programming (and thus found it difficult to work with OpenFlow controllers).

B. System Architecture

In order to continue providing researchers with the ability to control their tenant's network on the testbed, a set of network control RESTful APIs were exposed from the SAVI testbed's Software-Defined Infrastructure (SDI) manager [13]. This enabled the creation of web-based network services, which users have an easier time understanding. As can be seen in Fig. 5, the SDI manager is the central control and management component of the SAVI testbed, and serves as our API controller for receiving network flow installation requests from the testbed's users. With its ability to liaise with the other key components, the SDI manager is well positioned to handle the incoming flow installation requests from the testbed users.

The network state information is provided by two primary external sources: OpenStack plugins for network end-point information, and a topology manager [14] for the connectivity graph. As previously stated, the SAVI testbed is fully OpenFlow-enabled, thus the network controllers are a set of Ryu OpenFlow controllers [15]. Finally, OpenStack's Keystone serves as the IAM component and completes the basic system architecture for conflict detection. Further details about the overall SAVI network control architecture can be found in [16].

C. Implementation Details

Within the SDI manager, we are able to check each incoming flow installation request for violations of multi-tenant isolation. All the OpenStack components (Nova, Neutron, and Keystone), our graph topology manager, as well as the Ryu OpenFlow controllers, have existing RESTful APIs which we were able to leverage in order to perform the necessary authentication and authorization of each flow request as illustrated by the flow-chart in Fig. 3.a-g.

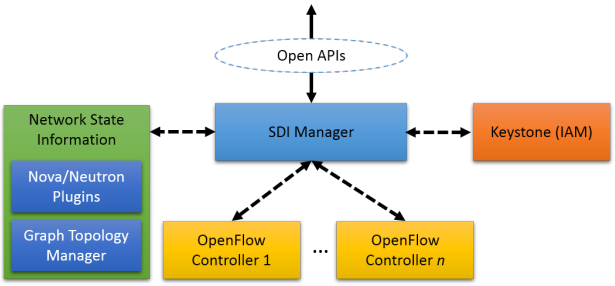


Fig. 5: Basic system architecture of prototype implementation. The SDI manager exposes APIs for network flow installation requests. Upon receiving requests, it 1) Authenticates the requester via the Keystone Identity and Access Management (IAM) component; 2) Authorizes the request using information provided by Nova, Neutron, and the Topology manager; and 3) If the previous two tasks pass, issues flow installation directives to the appropriate OpenFlow controller(s).

For the flow-level conflict detection, our focus during implementation was primarily on optimizing detection speed. Thus, we implemented the detection algorithm in C++ to avoid any unnecessary overheads introduced by interpreters, virtual machines, or just-in-time compilation. This has the benefit of allowing any future controllers written in C++ to leverage the library. In choosing the data structure to represent the bit arrays, we leveraged an optimized bit array library called *BITSCAN* [17].

A new class called *FlowRecords* was designed whose primary role is to store a record of all the OpenFlow rules in a given network. In its current implementation, FlowRecords works with OpenFlow 1.0 fields. When a new rule is to be inserted, it performs conflict detection against the existing flows in the network and returns an error if a conflict is found. Optionally, it can print any conflicting flows detected, thus making it a viable tool for auditing networks for existing conflicts.

To improve speed, our bit array sizes are determined at compile time. To reduce memory consumption, our implementation dynamically allocates bit arrays only when needed. Since the hash tables in Algorithms 1 and 2 maps header field values to bit arrays, this means that more diverse flows will require more bit array allocations. More objectively, memory consumption is proportional to *flow diversity*, which we define as the total number of unique header field values across all flows.

Since the SAVI SDI manager (i.e. the API controller responsible for receiving new flow requests) is written in Python, we also implemented a Python extension module to interface with our FlowRecords library. This essentially creates a wrapper around the FlowRecords library which any existing Python program can simply import. An advantage of this approach is that it keeps the core computational tasks in C++ (i.e. in machine code), thus minimizing the overhead due to the Python interpreter. We were able to integrate this into the SDI manager to fulfill the steps in Fig. 3.h-i.

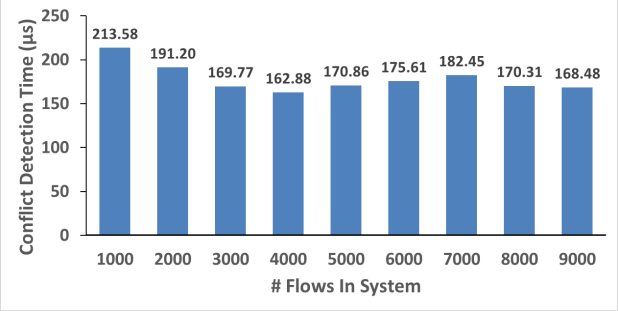


Fig. 6: Conflict detection time vs. number of flows in system (Bit-array size = 512,000)

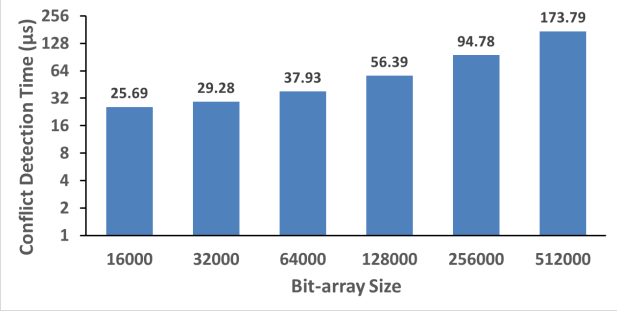


Fig. 7: Conflict detection time vs. Bit-array size

D. Performance Evaluation

To benchmark the flow request authentication and authorization stages (Fig. 3.a-g) in the SDI manager, we serially issued 1000 flow installation requests and observed a mean time of 59.5 *ms* per request. If we remove the authentication stage and limit the benchmarking to the authorization stage (for detecting multi-tenancy violation), we observe a mean time of 24.5 *ms* per request.

We next performed a series of experiments to evaluate the extent to which our optimized library is able to reduce flow-level conflict detection time (Fig. 3.h-i). We opted to benchmark the FlowRecords implementation through the Python extension module in order to determine the latency experienced by Python-based programs. To force conflicts, we first stored a number of non-conflicting flows into an instance of FlowRecords, and then replayed those flows by attempting to insert them in again. The average conflict detection time can thus be calculated by dividing the total replay time by the number of flows replayed. To isolate the conflict detection code and avoid any network overhead, we performed our evaluations directly using the software APIs rather than the SDI manager's RESTful APIs.

The first evaluation determined whether the conflict detection time increases as the number of existing flow entries in the network (i.e. the number of flows stored in FlowRecords) increases. Selecting a bit array size of 512,000, which represents a maximum of 512,000 flow rules in the network, we varied the number of unique flows from 1000 to 9000 and observed the average conflict detection times. The results can be seen in Fig. 6, which shows that the average conflict detection times are all in the order of microseconds, and shows no significant increase as the number of flows in the system increases. These results are in line with our expectations as the algorithms have a theoretical time complexity of *O(1)*. These results also show that our flow-level conflict detection outperforms existing flow-level conflict detection proposals. Time complexity of [9] was exponential and [8] was varied from *O(1)* to *O(n)* depending on the algorithm used. Compared with best case of [8], our conflict detection performs better by an order of magnitude (ms to μs scale) in terms of absolute detection time.

Our second evaluation was to determine whether the average conflict detection time increases with the size of the bit array. We held the number of unique flows in the system constant while varying the size of the bit array. The results of this set of experiments, as seen in Fig. 7, shows that the average time increases linearly with the bit array size. We note that this result is not surprising as CPUs cannot perform atomic bit-wise AND operations beyond the size of the hardware registers (64-bit in modern x86 general-purpose registers), thus the array must be segmented and processed in batches.

Due to space limitations, we present just a brief look on the memory consumption of our library. In our review of comparative works, we found that most experiments were conducted with a limited number of flows, the most being 3600 in [8]. Thus, we conducted an experiment with a bit array size of 3600 and inserted 3600 unique flows with a flow diversity of 3612 (i.e. the flows only differed within a single header field). The system consumed roughly 11.5 MB of RAM in this scenario, with an average conflict detection time of 28.8 μs. Considering our target deployment environment is composed of datacentre servers, we expect memory to be in abundance.

V. Conclusion and Future Work

Enabling SDN capabilities for users in a multi-tenant cloud presents many challenges that must be addressed in order to ensure the integrity of the overall environment. In this work, we have proposed an architecture that allows users in a multi-tenant cloud to install custom L2 flows into their network. Our system design provides assurance for multi-tenant isolation while providing flow conflict detection in order to catch inter- and intra-tenant flow-level conflicts, which may cause interruptions to running applications and services. We implemented a flow-level conflict detection library optimized for detection speed, and validated it within the SAVI testbed, a nationwide multi-tenant cloud in Canada. Preliminary performance evaluations reveal that our implementation can detect conflicts up to an order of magnitude faster than alternative proposals in literature.

In future work, we will work to extend our library in support of OpenFlow 1.3, as well as provide multi-table support. We will also try to optimize the memory consumption further to reduce the amount required by the system. In addition to these, we also plan to implement our algorithm on FPGAs for further performance improvement of our flow-level conflict detection algorithm. Finally, we plan to deploy our proposed system across all regions of the SAVI testbed and open the flow installation APIs to general testbed users.

REFERENCES

[1] J. E. van der Merwe, S. Rooney, L. Leslie, and S. Crosby, "The Tempest-a practical framework for network programmability," *IEEE Network*, vol. 12, no. 3, pp. 20–28, May 1998.

[2] N. M. K. Chowdhury and R. Boutaba, "A Survey of Network Virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862 – 876, 2010.

[3] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "FlowVisor: A Network Virtualization Layer," *OpenFlow Switch Consortium, Tech. Rep*, pp. 1–13, 2009.

[4] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "OpenVirteX: Make Your Virtual SDNs Programmable," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620741

[5] Z. Bozakov and P. Papadimitriou, "AutoSlice: Automated and Scalable Slicing for Software-Defined Networks," in *Proceedings of the 2012 ACM Conference on CoNEXT Student Workshop*, ser. CoNEXT Student '12. New York, NY, USA: ACM, 2012, pp. 3–4. [Online]. Available: http://doi.acm.org/10.1145/2413247.2413251

[6] D. Drutskoy, E. Keller, and J. Rexford, "Scalable Network Virtualization in Software-Defined Networks," *IEEE Internet Computing*, vol. 17, no. 2, pp. 20–27, March 2013.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[8] S. Natarajan, X. Huang, and T. Wolf, "Efficient Conflict Detection in Flow-Based Virtualized Networks," in *Computing, Networking and Communications (ICNC), 2012 International Conference on*, Jan 2012, pp. 690–696.

[9] B. Lopes Alcantara Batista, G. Lima de Campos, and M. Fernandez, "Flow-Based Conflict Detection in OpenFlow Networks Using First-Order Logic," in *Computers and Communication (ISCC), 2014 IEEE Symposium on*, June 2014, pp. 1–6.

[10] Open Networking Foundation, "OpenFlow Switch Specification," 2014, [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf [Accessed: September 2016].

[11] OpenStack Foundation, "OpenStack Open Source Cloud Computing Software," [Online]. Available: http://www.openstack.org/ [Accessed: 20-September-2016].

[12] T. Lin, "Implementation and Evaluation of an SDN Management System on the SAVI Testbed," *MS dissertation, University of Toronto*, 2014.

[13] J.-M. Kang, H. Bannazadeh, H. Rahimi, T. Lin, M. Faraji, and A. Leon-Garcia, "Software-Defined Infrastructure and the Future Central Office," in *Communications Workshops (ICC), 2013 IEEE International Conference on*, June 2013, pp. 225–229.

[14] J. M. Kang, H. Bannazadeh, and A. Leon-Garcia, "SDIGraph: Graph-based management for converged heterogeneous resources in SDI," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, June 2016, pp. 88–92.

[15] NTT DoCoMo, "Ryu SDN Framework," [Online] Available: http://osrg.github.io/ryu/ [Accessed: 20-September-2016].

[16] B. Park, T. Lin, H. Bannazadeh, and A. Leon-Garcia, "JANUS: Design of a Software-Defined Infrastructure Manager and Its Network Control Architecture," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, June 2016, pp. 93–97.

[17] P. San Segundo, "BITSCAN," [Online]. Available: https://github.com/psanse/bitscan [Accessed: September 2016].