SMART: Shared Memory based SDN Architecture to Resist DDoS ATtacks

Sana Belguith[1][a], Muhammad Rizwan Asghar[2][b], Song Wang[3], Karina Gomez[3] and Giovanni Russello[2]

[1]*School of Computing, Science and Engineering, University of Salford, Manchester, U.K.*

[2]*Cyber Security Foundry, The University of Auckland, Auckland, New Zealand*

[3]*School of Engineering, RMIT University, Melbourne, Australia*

Keywords: SDN, Security, Shared Memory, Tuple Spaces, DDoS, Availability, OpenFlow.

Abstract: Software-Defined Networking (SDN) is a virtualised yet promising technology that is gaining attention from both academia and industry. On the one hand, the use of a centralised SDN controller provides dynamic configuration and management in an efficient manner; but on the other hand, it raises several concerns mainly related to scalability and availability. Unfortunately, a centralised SDN controller may be a Single Point Of Failure (SPOF), thus making SDN architectures vulnerable to Distributed Denial of Service (DDoS) attacks. In this paper, we design SMART, a scalable SDN architecture that aims at reducing the risk imposed by the centralised aspects in typical SDN deployments. SMART supports a decentralised control plane where the coordination between switches and controllers is provided using *Tuple Spaces*. SMART ensures a dynamic mapping between SDN switches and controllers without any need to execute complex migration techniques required in typical load balancing approaches.

1 INTRODUCTION

Software-Defined Networking (SDN) is a virtualised yet promising technology that is gaining attention from both academia and industry. The SDN architecture provides a clear separation between a data plane and a control plane. The former one is for forwarding traffic and the latter one deals with routing decisions made by SDN switches. In SDN, a centralised control plane is responsible for giving directions to the data plane (Scott-Hayward et al., 2016). A control plane is formed by one or more SDN controllers; whereas, SDN switches constitute a data plane. An SDN controller is a powerful tool and is considered the brain of an SDN network.

In SDN, a controller installs flow rules on SDN switches. These rules allow SDN switches to handle network flows. When a new flow, which does not match any existing flow rules, is encountered, the switch sends a packet-in message to the SDN controller for receiving instructions on how to deal with that traffic (*cf.* Fig. 1). The use of a centralised SDN controller enables configuration, management, and optimisation of network resources dynamically through SDN applications in an efficient manner, independent from the underlying network infrastructure.

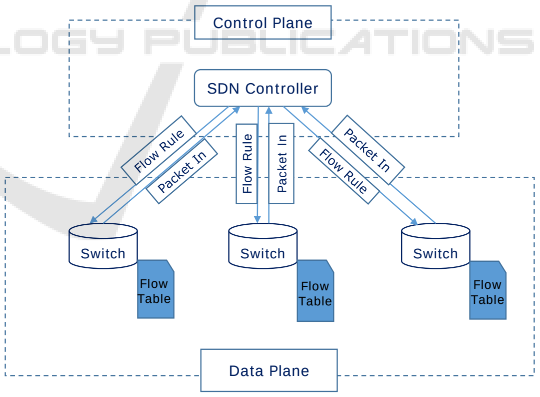

Figure 1: The SDN architecture illustrating a logical and physical separation of both control and data planes, placement of flow tables, and high-level interactions between an SDN controller and SDN switches.

Despite its benefits, the centralised nature of an SDN controller raises several concerns mainly related to scalability and availability. Indeed, a cen-

[a] https://orcid.org/0000-0003-0069-8552
[b] https://orcid.org/0000-0002-9607-376X

Belguith, S., Asghar, M., Wang, S., Gomez, K. and Russello, G.
SMART: Shared Memory based SDN Architecture to Resist DDoS ATtacks.
DOI: 10.5220/0009864906080617
In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications (ICETE 2020) - SECRYPT*, pages 608-617
ISBN: 978-989-758-446-6

tralised SDN controller may not be suitable for applications involving multi-domain and multi-technology features, such as the architecture of Internet Service Providers (ISPs) interconnecting each other, where each ISP owns different types of equipment and technologies. Moreover, a centralised SDN controller may be a Single Point Of Failure (SPOF), which causes the vulnerability of SDN architectures to attacks (Dixit et al., 2013; Scott-Hayward et al., 2016). To mitigate these issues, there are proposals to use logically centralised, but physically distributed SDN controllers (Phemius et al., 2014) to avoid the SPOF and to offer better network scalability. Although scalability and availability of the SDN architecture can be improved using distributed controllers, such an approach has one main drawback when it comes to linking switches to controllers; actually, a switch is statically connected to a controller, which prevent the control plane from adapting traffic load variations. To address this drawback, a naive solution is to over-provision controllers in order to deal with the expected peak load (Dixit et al., 2013). However, this solution is inefficient due to its high cost and energy consumption.

A key limitation of SDN is its vulnerability to Distributed Denial of Service (DDoS) attacks (Bhushan and Gupta, 2019). Indeed, there is a paradoxical relationship between SDN and DDoS attacks. On the one hand, the separation of the control and data planes results in an easier detection and optimised defence against DDoS attacks. On the other hand, the centralisation of the control plane introduces new DDoS threats as the control plane may be a target of these attacks (Yan and Yu, 2015). In DDoS attacks, an attacker could generate data packets for which SDN switches might have no flow rule, thus forcing them to forward those packets to the SDN controller. By sending a large number of new data packets, SDN could be flooded, thus resulting in a DDoS attack (*cf.* Fig. 2). Basically, these are table-miss packets that will trigger massive packet-in messages from SDN switches to the controller, that is incurring an excessive consumption of the bandwidth, CPU resources, and memory in the control plane as well as the data plane.

A promising approach could be to balance load dynamically among distributed controllers to handle table miss-packets in normal use as well as during flooding attacks. This requires the introduction of a load balancer to efficiently distribute the control plane requests among different controllers. Moreover, dedicated switch migration algorithms have to be executed to map switches to controllers while performing the load balance strategy.

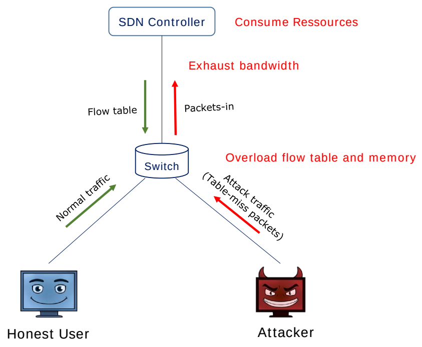

Figure 2: An illustration of DoS attacks on SDN: An attacker generates table-miss packets and forwards them to the SDN controller. By sending a large number of new data packets, an SDN controller could be flooded.

A more logical solution would be to use a coordination layer between switches and controllers to handle the control plane requests and ensure dynamic mapping between SDN switches and controllers. To this end, we introduce the use of *Tuple Spaces* as a decoupling mechanism for coordinating the communication between SDN switches and controllers. Tuple spaces were introduced as part of the LINDA language in (Gelernter, 1985) for building parallel/distributed applications. Tuple spaces are basically shared memory objects that provide a simple API to store and retrieve ordered sets of data, called *Tuples*. Using tuple spaces as a coordination layer enables the decoupling of the communication between SDN switches and controllers in space and time: in space as SDN controllers and switches are not statically bound to each other; in time as SDN switches and controllers are not required to be online at the same time to transfer data. Through this decoupling, it is possible to balance requests across all the active available controllers. Moreover, one can dynamically activate new controllers when there is a traffic peak.

In this paper, we present SMART, a Shared Memory based SDN Architecture to Resist DDoS a**T**tacks. SMART leverages a decentralised control plane where tuple spaces are used to coordinate the communication between SDN switches and controllers. In SMART, SDN switches can execute operations to store packet-in requests as tuples as well as to retrieve flow rules. SDN controllers consume packet-in tuples inserted by switches, generate related flow rules and put them back in the tuple space.

Our proposed architecture is multi-fold. First, SMART is based on using a distributed control plane involving several controllers, which makes the network more scalable. Second, thanks to the use of tuple space, SMART ensures a dynamic mapping between switches and controllers. Unlike other decentralised SDN networks, SMART allows a switch to communicate in a decoupled manner with controllers through tuple space instead of being statically mapped to a master controller. Third, through this decoupling, we provide a smooth load balance among controllers, which can be useful to defeat DDoS attacks as the traffic may be efficiently distributed among controllers without offloading any node. Last but not least, tuple space can detect excessive accesses from switches during DDoS attacks that could trigger a mitigation strategy. For instance, when an attack is detected, packet-in requests are detoured to a data plane cache. This module is used to store the table-miss packets received during DDoS attacks, which are filtered to distinguish benign ones and then move them to tuple space to be addressed by the controllers. Therefore, this technique maintains system availability during a flooding attack.

The remainder of this paper is organised as follows. Section 2 reviews related work. In Section 3, we describe research challenges to be addressed by the solution. Section 4 provides background information on tuple spaces and their features relevant to SDN. Section 5 gives an overview of SMART as well as its building blocks. In Section 6, we provide solution details before concluding in Section 7.

2 RELATED WORK

SDN architectures have been widely studied in the literature to enhance their scalability and security against attacks. In the following, we first review the literature covering load balancing solutions for SDN. Then, we discuss existing DDoS mitigation techniques.

2.1 Load Balancing for Distributed SDN

Several research works have proposed load balance algorithms for SDN architectures. Obviously, load balance solutions in SDN can be categorised into two types: centralised and decentralised algorithms.

To ensure dynamic load balance on distributed SDN networks, ElastiCon (Dixit et al., 2013) has been designed so that the workload is dynamically balanced to enable the controllers to perform at a pre-specified load window. When the aggregate overhead changes over time, the system dynamically adds or removes controllers from the controllers cluster as needed.

COLBAS (Selvi et al., 2016) is a controller load balancing scheme for hierarchical networks that relies on controller cooperation via cross-controller communication. COLBAS assigns one of the controllers as a super controller that can flexibly manage flow requests handled by each controller. This super controller is located at the network node closest to the geographical centre of the topology. All controllers publish their own load information periodically through a cross-controller communication system. When traffic conditions change, the super controller reassigns different flow setups to proper controllers and installs allocation rules on switches for load balancing. BalCon (Cello et al., 2017) has been designed to improve load balance between controllers by implementing a dedicated switch migration algorithm. This solution relies on measuring the load of each controller. Once a controller's load reaches a pre-fixed threshold, it is considered as overloaded and a set of switches are chosen to be migrated into another controller.

Centralised load balancing solutions rely on the use of a centralised node whose performance is limited by memory, CPU computation, and network bandwidth. Besides, a centralised node should collect all the controllers' overheads in the network, which requires exchanging data with them frequently. Moreover, if the central node collapses, the whole load balancing strategy is down. To this end, decentralised load balance solutions have introduced (Zhou et al., 2014; Ammar et al., 2017). DALB is a distributed load balance architecture (Zhou et al., 2014) running as a module of an SDN controller to enable load estimation. This algorithm avoids the single point of failure problem, thus providing scalability and availability. The controller periodically collects its own load information. Then, it checks whether the load is beyond the threshold. If the load exceeds the threshold, the controller gathers other controllers' load information. After aggregating all load information from the controller cluster, the high load controller makes a migration decision and selects which switch will be migrated to a low load controller.

Although distributed load balance approaches ensure an efficient flow distribution among the controllers' pool, SDN architectures remain an attractive target for DDoS attacks. Indeed, an SDN controller usually chooses an alternative route between entry point and end point only if the first current route is not available, which ensure network reliability. Nevertheless, during DDoS attacks, this strategy is inefficient as attack traffic will be redirected through a different route. Thus, there is a high probability that the new route will be overloaded as well.

2.2 Solutions against (D)DoS Attacks on SDN

Several solutions have been proposed to mitigate DDoS attacks in SDNs (Shin et al., 2013; Chen et al., 2016; Wang et al., 2015). AVANT-GUARD (Shin et al., 2013) has been introduced to countermeasure DDoS attacks impacts on SDN networks. First, AVANT-GUARD extends SDN data plane using a *connection mitigation* feature that reduces interaction between data plane and control plane. This added feature drops failed TCP sessions at the data plane prior to any notification to the control plane. Second, AVANT-GUARD relies on *actuating triggers* which enable the data plane to asynchronously report network status and payload information to the control plane. Nevertheless, this solution only defeats TCP-based flooding attacks but does not resist other kinds of attacks such as UDP and ICMP. Moreover, the authors do not propose any mechanism to handle packets during the attack. SDNShield (Chen et al., 2016) incorporates two defence lines to mitigate DDoS attacks on SDN. This solution is based on filtering incoming flows to identify the legitimate ones to be routed directly to their destinations. SDNShield relies on the use of a network of a specialised cluster of virtual machines to double-check the rejected packets. To mitigate flooding attacks, Shang *et al.* (Shang et al., 2017) have proposed four functional modules (including attack detection, table-miss engineering, packet filter, and flow rule management) to extend the SDN architecture in order to mitigate DoS attacks. This solution mainly applies a packet migration mechanism to handle table-miss packets on saturated switches. Obviously, missed packets are detoured to the attacked switches neighbours to be sent later to the controller. In (Wang et al., 2018), the authors introduced a set of triggers to resist DoS attacks in SDN. The proposed solution, called SECOD, executes several functions to monitor and detect excessive packet-in requests and launches a DoS mitigation mechanism. Recently, Alshra *et al.* (Alshra'a and Seitz, 2019) have designed a hardware solution to stop injection attacks. Indeed, this solution introduces a new hardware component called INSPECTOR, which is mainly responsible for verifying the authentication of packet-in messages that access the network before forwarding them to the data and control planes. This mechanism stops malicious packet-in messages from being processed and reduces the risks of DoS attacks.

Although these solutions ensure efficient detection and mitigation of DDoS attacks, they do not consider the scalability issue. Thus, the system may be overloaded without a capability to be adapted to traffic variation in normal use or even during DDoS attacks. Moreover, they incur complex migration techniques to change switch mapping to controllers.

In this paper, SMART relies on a distributed control plane to mitigate SPOF risks while deploying novel load balancing techniques based on Tuple spaces. This enables a dynamic load balance among distributed controllers to handle table miss-packets in normal use as well as during flooding attacks, without executing complex migration algorithms.

3 RESEARCH CHALLENGES

In this paper, we address the following research challenges:

- **R1 Scalability.** Expand/shrink the resources pool by dynamically adding/removing controllers to be adapted to traffic load variations with simple plug and play approach.
- **R2 Automatic/Efficient Switch Migration.** Ensure dynamic mapping between SDN switches and controllers to provide elasticity and efficiency of the system during traffic load variations.
- **R3 Availability.** Detect and prevent DDoS attacks to guarantee the availability of both control and data planes during DDoS attacks.
- **R4 Flexibility.** Be able to handle attacks for different protocols (*e.g.,* TCP-based attacks and UDP-based attacks), *i.e.,* design a solution that works with all kinds of attack traffic protocols.
- **R5 Reliability.** Efficiently deal with table-miss packets during the DDoS attack without losing benign traffic.

In the following sections, we will describe how SMART tackles these research challenges.

4 AN OVERVIEW OF TUPLE SPACES

In this section, we start by presenting some background information about tuple spaces. Then, we detail GSpace, the tuple space middleware we aim to use in our solution.

4.1 Tuple Space

A tuple space can be considered as a shared memory object allowing different processes to execute operations to store and retrieve ordered data sets (Carriero Jr, 1987). A tuple is an ordered sequence of fields where a field that contains a value is said to be defined. A tuple is called entry or tuple t if all its fields are defined while a tuple is called a template $\tilde{t}$ if at least one of its fields does not have a defined value. Templates are tuples used to retrieve tuples from the tuple space, *i.e.,* a template and a tuple match if and only if they have the same number of fields and all the values and types of the defined fields in $\tilde{t}$ are identical to the values and types of the corresponding fields in t (Floriano et al., 2017). Three basic operations are executed to store and retrieve data in tuple spaces: $out(t)$ used to store tuple t in the space; $in(\tilde{t})$ that reads and removes a tuple t that matches $\tilde{t}$; $rd(\tilde{t})$ that accesses a tuple t without removing it from the space. $in(\tilde{t})$ and $rd(\tilde{t})$ are known as blocking operations: if no tuple t matches the template $\tilde{t}$, the process is blocked until it gets a matching one.

4.2 Bridging Gap between SDN and Tuple Space

A tuple space is similar to a persistent storage memory: tuples can be stored in until they are retrieved by a process. A process that creates a tuple is called a producer while a consumer is a process that retrieves the stored tuples. A process could be both a consumer and producer. The communication between the producer and consumer is decoupled in space because any process can store a tuple which can be read by any other process without the need to know the exact location of the process producing the tuple. In addition, it is decoupled in time as the tuple can be retrieved by any process at any time. These features might be useful in SDN architecture to allow switches to store packet-in and then retrieve related flow rules generated by controllers. Similarly, controllers have to read and remove the packet-in from the tuple space, then write flow rules to be retrieved by the switch. This decoupling in the communication between switches and controllers provides more dynamic and efficient communication architecture.

Logically, a tuple space is a shared memory because the objects are shared between different processes. However, the way in which tuple spaces are implemented can be abstracted from its logical view. In principle, a tuple space could be implemented as a centralised repository. However, this would introduce another SPOF. Recent implementations of tuple spaces are built as distributed middleware where part of the space is stored within each physical components in the network. The main advantage here is that it does not need any extra hardware as it can be distributed over the switches and controllers in an efficient way. Each middleware component may be applied to audit the accesses performed by the local process. In this way, it becomes easier to measure switches' accesses to the tuple space and detect any out of the norm access frequency which could indicate a flooding attack.

Finally, most of tuple space implementations (Russello et al., 2004a; Hari, 2012) provide dynamic load balancing features. When a node is overloaded with tuples, the system automatically balances the load by shifting the tuples to other nodes. This feature is extremely useful to ensure efficient load balance among distributed controllers without requiring any dedicated algorithms.

4.3 GSpace Middleware

In our proposed solution, we use GSpace which is an adaptive middleware based on the distributed shared data space model. In GSpace middleware, a number of GSpace kernels are installed on several networked nodes. Each kernel enables the storage of tuples locally besides establishing communications with other kernels located in other nodes. GSpace kernels collaborate together to guarantee a unified view of the shared data space to the components of the application. Therefore, from the point of view of the application, the shared data space is viewed as an only one combined logic layer despite of being shared across several nodes. Internally, each GSpace kernel is organised as follows.

- **The System Boot Module:** is responsible for initiating all the other modules of GSpace kernel. Subsequently, it advertises its presence to other GSpace nodes in order to establish communication channels and join a GSpace group.
- **The Controller:** provides the *in*, *out* and *rd* operations to be used by the GSpace nodes.
- **The Dynamic Invocation Handler (DIH):** determines which distribution policy to apply based on the type and content of the tuple. When a node executes a GSpace operation, DIH checks a distribution policy descriptor to determine which distribution policy to apply. Once the policy has been identified, DIH invokes a distribution manager that implements the applicable policy.
- **The Distribution Managers:** are responsible for enforcing distribution policies. For each distribution policy supported by the system there is a specific distribution manager. The manager may impose that tuples are sent to or requested from

GSpace kernels on other nodes with respect to the executed distribution policy.

- **The Data Space Slice:** provides local storage for tuples together with the associative method for retrieving them. In addition, a Memory Sensor module associated with this module calculates the memory used to store tuples on each kernel.
- **The Communication Module:** provides facilities for sending or retrieving tuples to and from other GSpace kernels.
- **The Connection Manager:** is responsible for keeping track of information about network locations of other GSpace nodes that can be used by distribution managers.
- **The Logger:** audits the executed operations on the local kernel. Upon receiving a space operation request, the Logger is informed and then it keeps records of all the operations executed for each tuple type. If the number of operations for a particular tuple type reaches a prefixed threshold, the Logger informs its local Adaptation Module.
- **The Adaptation Module (AM):** makes the decision on starting the different phases of the adaptation mechanism. The adaptation mechanism enables the selection of the most optimised distribution policy for a specific tuple type during runtime.

More details about GSpace functionalities can be found in (Russello et al., 2004a; Russello et al., 2004b).

5 SYSTEM DESIGN

To address the security challenges discussed in section 3, we introduce SMART as an extension of the SDN architecture to defeat DDoS attacks. In this section, we begin by introducing a general overview of the proposed SDN architecture 5.1. Then, we present SMART building blocks in Section 5.2.

5.1 SMART General Overview

In this paper, we present SMART, a scalable and secure SDN architecture (see Fig. 4). SMART extends the SDN basic architecture to defeat DDoS attacks based on a novel use of the tuple space concept. In SMART, switches are not directly connected to a master controller, however they store their packet-in requests in the tuple space as tuples. A switch creates a tuple containing its identifier and the packet-in request and stores it in the tuple space using $out(S_{id}, Packet-in)$. Controllers can read stored tuples in the space tuple by issuing $in(*, Packet-in)$ ('*' denotes an undefined field, called wildcard). Therefore, a controller can get any packet-in request stored and process it regardless of the switch that stores it. Afterwards, the controller generates a tuple of the related flow table entry $out(S_{id}, Flowtable)$ using the same switch identifier S_{id} to ensure that it is retrieved by the right switch. This latter issues an $in(S_{id}, *)$ operation to retrieve the flow table generated as an answer to its packet-in request. The *in* operation ensures that the switch waits until the flow table rule is stored by the controller to get it.

Beyond ensuring efficient packet-in management, SMART allows early detection of denial of service attacks. Indeed, switches' accesses to the tuple space are audited to detect any flooding attacks produced by a compromised switch. When an attack is detected, packet-in requests are detoured to a data plane cache. This module is used to store the table-miss packets received during DDoS attacks which are filtered to distinguish benign ones and then move them to the tuple space to be addressed by the controllers. This technique maintains the system availability during a flooding attack.

5.2 SMART Building Blocks

We first assume that an initialisation phase has been executed to grant access to the tuple space by different controllers and switches in the network. SMART components are detailed as follows:

- **Space Tuple Access.** A switch receiving a table-miss packet generates a packet-in request and stores it in the space tuple by executing an *out* operation; $out(S_{id}, Packet-in)$. The switch includes its identifier S_{id} in the *out* operation in order to retrieve later the flow table generated by a controller for its packet-in request. Then, the switch may execute an $in(S_{id}, *)$ operation to seek the flow table related to its stored packet-in request. The *in* operation blocks the switch until the expected flow table is stored by a controller to access and remove it from the tuple space. On the other side, controllers store $(*, Packet-in)$ tuple using *in* operation. By this, a controller is blocked and waits for any packet-in request to be stored without any restriction on the source of the request, *i.e.,* the switch that has generated the request. A controller accesses and removes the packet-in request, processes it and then stores the generated flow table using the switch identifier $out(S_{id}, Flowtable)$. Therefore, controllers dy-

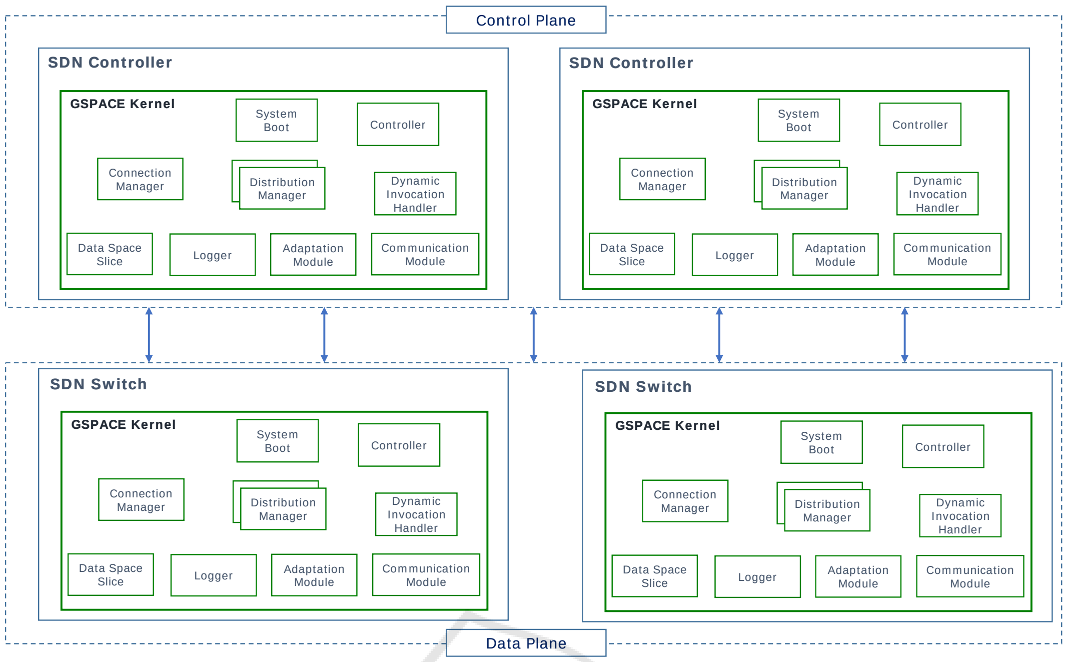

Figure 3: Deployment of GSpace components in the control and data planes of SDN. A typical setup consists of several GSpace kernels instantiated inside all SDN controllers and switches. Each kernel provides facilities for storing tuples locally, and for discovering and communicating with other kernels. GSpace kernels collaborate with each other to provide to the application components a unified view of the tuple space.

namically access and process packet-in requests. This ensures an efficient and smooth communication between controllers and switches without any need to execute sophisticated switch migration mechanisms. In addition, the space tuple ensures an efficient load balancing among controllers. For instance, if a controller is filled, tuples are dynamically shifted to others.

- **Shared Memory Monitoring.** As the tuple space limits switches to store only one tuple by using *out* primitive at a time, it is easy to count the number of tuples stored by each switch and detect if a switch exceeds a threshold of packet-in. Therefore, a flooding attack can be detected on time and a defence mechanism is executed.
- **Scalability Module.** As the tuple space manages the load balance among controllers, it can detect if the controllers cluster is not adapted to the network traffic. Consequently, a scalability module can be executed to add new controllers to the cluster. Obviously, thanks to the load balancing feature offered by the space tuple, tuples will be shifted to this new controller. For ease of presentation, we assume that there is a number of controllers dedicated to be added to each cluster.
- **DDoS Detection.** When the tuple space detects an excessive number of stored packet-in requests by the same switch, it is considered as a malicious switch launching a flooding attack. Hence, a flow entry is generated to make the switch drops all mis-matched packets. If the switch continues to send requests, then it is considered as an adversary and the function *Switch Defence* is executed. Otherwise, after executing the received flow entry by the switches, the source of the compromised packets is identified. For instance, if it is a host machine then the *Host Defence* module is activated. Meanwhile, if the malicious packets source is another switch, then this detected switch is considered as malicious and the current switch is allowed to access the tuple space again.
- **Switch Defence.** To mitigate DDoS attacks, the compromised switch is prohibited from accessing the shared memory. Obviously, the switch stores packet-in requests on the tuple space using its identifier S_{id}. Therefore, to block a malicious switch, the tuple space routes all *out* operations containing the switch's identifier to the data plane cache. During the DDoS attacks, most of traffic will be transmitted to the data plane cache instead of flooding the controllers to be filtered by the *Packet Filter* module. However, a switch may forward packet-in requests received from another switch to the tuple space. A compromised switch may use an honest switch to launch a DDoS at-

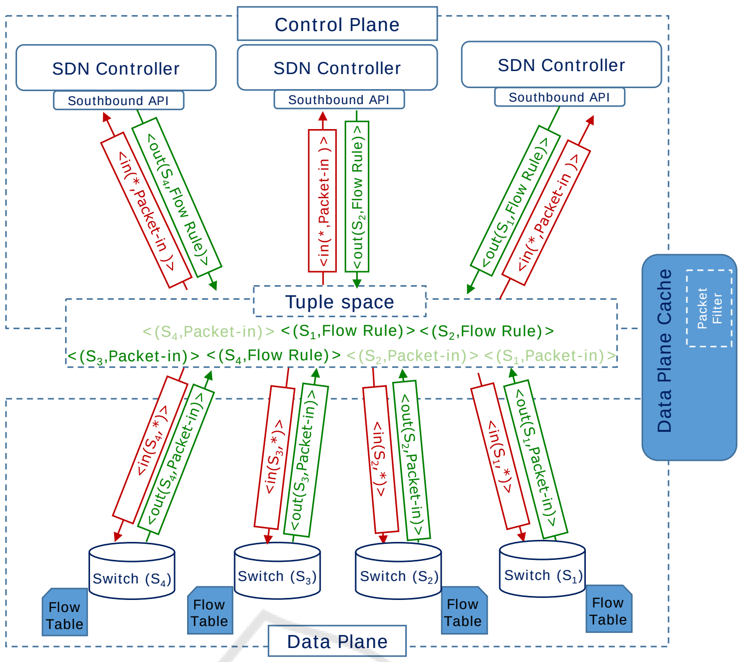

Figure 4: The SMART architecture: Switches store their packet-in requests in the tuple space as tuples without being statically connected to a master controller. A switch stores an $out(S_{id}, Packet-in)$ in the tuple space. Then, it issues an $in(S_{id},*)$ operation to keep waiting for the flow table generated by a controller. Controllers can read stored tuples in the space tuple by issuing $in(*,Packet-in)$. The controller generates instructions for the packet-in request using $out(S_{id}, Flowtable)$ and including the switch identifier S_{id} to be retrieved by the right switch.

tack. In this case, when a switch is blocked the attack will continue through another honest switch. To defeat this kind of attacks, before blocking a switch the *DDoS Detection* module first retrieves the source of the attack, whether it is a switch or a host machine.

- **Packet Filter.** This module uses traffic information and monitoring rules to identify normal traffic. If one packet-in request is regarded legal, it will be moved to be processed by controllers.
- **Host Defence.** This module aims to identify the sources of the DDoS attack when the switches are not compromised. Then, all the packets coming from this IP address are dropped.

6 SMART: TOWARD A SECURE AND SCALABLE SDN ARCHITECTURE

Basically, an SDN network consists of separating the control plane from the data plane. Therefore, when a switch receives a packet which does not match any entry in its flow table, it stores a packet-in request on the control plane asking for instructions on how to handle the received packet. The controller answers by forwarding a flow rule to the switch. To make this communication efficient and smooth, we introduce a shared tuple space, *GSpace*, between the two network layers. GSpace is a logically centralised shared memory but physically deployed inside all SDN nodes (see Fig. 3). In our model, a GSpace kernel is implemented inside all SDN controllers and SDN switches. In the initialisation phase, *the system boot* module advertises its presence to all other GSpace nodes. Afterwards, it establishes a communication channel and joins the GSpace group. The SDN controllers, first, store an *out* operation over a template $(*, Packet-in)$ to seek for packet-in requests registered by switches.

When a switch stores a tuple on the local GSpace kernel using $out(S_{id}, Packet-in)$ operation, *the Dynamic Invocation Handler (DIH)* checks the content and the type of the tuple and executes the required distribution policy. DIH matches the tuple against the stored templates and invokes *the Distribution Manager (DM)* to execute the distribution policy. Afterwards, DM follows the distribution policy and dictates to which SDN controller node the packet-in tuple needs to be forwarded. The communication between the source distribution manager and the destination distributed manager is done through the *communication module*. For each stored tuple in the GSpace kernel, the *data space module* measures the amount of memory that is used for storing tuples on each kernel. This measurement is done on the switches' kernels as well as on the controllers' kernels. The switch stores an $out(S_{id}, *)$ on its local kernel to keep waiting for the instruction generated by a controller. Following the packet-in processing by the SDN controller, it stores an $out(S_{id}, Flowtable)$ on its local GSpace kernel. Afterwards, *the dynamic invocation handler* matches the stored tuple with the templates already registered by the switches ($out(S_{id}, *)$). Once a matching tuple is found, DIH orders the distribution manager to execute the related distributed policy in order to forward the flow table rule to the switch.

In each GSpace kernel deployed on SDN controllers and switches, the *memory sensor module* located in the *data space module* measures the tuple stored locally by each node. For switch kernels, we set a threshold for $out(S_{id}, Packet-in)$ operations. Indeed, measuring *out* operations executed is used to detect excessive accesses to the control plane by each switch. If this threshold is reached, the *logger* module considers the switch as malicious and notifies the *DDoS Detection* module to be activated. For instance, this module invokes the dynamic invocation handler on the SDN controllers' kernels side to insert a drop flow instruction by storing an $out(S_{id}, DropFlow)$. The switch which is already waiting for instructions is notified of the generation of the flow rule. Then, it processes the received flow rule.

Using information received from the *memory sensor module*, the *DDoS Detection* module checks whether the switch has stopped writing in its local GSpace kernel or not. In this scenario, there are two cases to be considered. On the one hand, if the switch has installed the flow rule and executed it carefully, then the *DDoS Detection* module activates the *host defence* module to find out the source of the attack. Once the IP address of the host machine is identified, the *host defence* module orders a dynamic invocation handler on the SDN controllers' kernels to generate a new flow rule to invoke the switch to continue storing tuples on its kernel while dropping all packets received from the identified IP address. In the case that the identified IP address belongs to a switch in the GSpace network, the *DDoS Detection* module activates the *switch defence* module explained in the following. On the other hand, if the switch ignores the flow rule generated by an SDN controller to drop all mis-table packets and continues storing tuple on its GSpace kernel, the *DDoS Detection* module considers the switch as compromised. Therefore, it orders the DIH to stop processing the generated tuples and invokes the distribution manager to execute a specific distribution policy aiming that these tuples are routed to the *data plane cache* module. Afterwards, the *packet filter* module processes the packet-in requests and moves the benign ones to the controllers to process them. In this paper, we apply the *packet filter* module concept in the same way as introduced in (Shang et al., 2017).

In the GSpace kernels deployed on SDN controllers, the *memory sensor module* measures tuples stored locally by each node. A threshold is set to monitor the SDN controllers overhead. If this threshold is reached, the *logger* module notifies the *Adaptation Module* (AM) to select the best data distribution policy to be adopted. In other words, a new policy consisting of shifting tuples to other SDN controllers' kernels is executed. In the case where the AM receives requests from all the *loggers* over the GSpace network, it activates the *scalability module*. This latter proceeds by adding a new controller to the cluster. While joining the cluster, the GSpace kernel deployed in the new controller advertises its presence to all other GSpace nodes using the *system boot* module. Afterwards, the *adaptation module* selects a particular distribution policy aiming at shifting tuples to the added controller. Therefore, an efficient load balance among controllers is achieved.

7 CONCLUSION

Although the popularity of SDN is increasing, it is still facing several challenges related to scalability and availability. In this paper, we have presented SMART, a shared memory based architecture to resist DDoS attacks. Our proposed solution combines tuple spaces and decentralised SDN control plane. Indeed, using a dedicated tuple space, SDN switches may store their packet-in requests, which can be accessed by SDN controllers in a dynamic way. After processing the packet-in requests, the controller stores the related flow rule as a tuple in the tuple space which is retrieved by the switch. In addition to the dynamic mapping between switches and controllers, the tuple space can measure the number of packet-in requests stored by a switch in order to detect any malicious activity that triggers a mitigation strategy. Furthermore, the controller overheads are monitored at runtime to ensure an efficient load balance among the network nodes as well as executing a scalability process to add new controllers when needed. In the future, we aim at implementing a prototype of SMART and measuring its performance in deployment scenarios.

REFERENCES

Alshra'a, A. S. and Seitz, J. (2019). Using INSPECTOR device to stop packet injection attack in SDN. *IEEE Communications Letters*.

Ammar, H. A., Nasser, Y., and Kayssi, A. (2017). Dynamic SDN controllers-switches mapping for load balancing and controller failure handling. In *Wireless Communication Systems (ISWCS), 2017 International Symposium on*, pages 216–221. IEEE.

Bhushan, K. and Gupta, B. B. (2019). Distributed denial of service (ddos) attack mitigation in software defined network (sdn)-based cloud computing environment. *Journal of Ambient Intelligence and Humanized Computing*, 10(5):1985–1997.

Carriero Jr, N. J. (1987). Implementation of tuple space machines.

Cello, M., Xu, Y., Walid, A., Wilfong, G., Chao, H. J., and Marchese, M. (2017). BalCon: A distributed elastic SDN control via efficient switch migration. In *Cloud Engineering (IC2E), 2017 IEEE International Conference on*, pages 40–50. IEEE.

Chen, K.-y., Junuthula, A. R., Siddhrau, I. K., Xu, Y., and Chao, H. J. (2016). SDNShield: Towards more comprehensive defense against DDoS attacks on sdn control plane. In *Communications and Network Security (CNS), 2016 IEEE Conference on*, pages 28–36. IEEE.

Dixit, A., Hao, F., Mukherjee, S., Lakshman, T., and Kompella, R. (2013). Towards an elastic distributed SDN controller. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 7–12. ACM.

Floriano, E., Alchieri, E., Aranha, D. F., and Solis, P. (2017). Providing privacy on the tuple space model. *Journal of Internet Services and Applications*, 8(1):19.

Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112.

Hari, H. (2012). Tuple space in the cloud.

Phemius, K., Bouet, M., and Leguay, J. (2014). Disco: Distributed multi-domain SDN controllers. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–4. IEEE.

Russello, G., Chaudron, M., and Van Steen, M. (2004a). Dynamic adaptation of data distribution policies in a shared data space system. In *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*, pages 1225–1242. Springer.

Russello, G., Chaudron, M., and Van Steen, M. (2004b). GSpace: Tailorable data distribution in shared data space system. Technical report, Technical report Technical Report 04/06, Technische Universiteit Eindhoven, Department of Mathematics and Computer Science.

Scott-Hayward, S., Natarajan, S., and Sezer, S. (2016). A survey of security in software defined networks. *IEEE Communications Surveys & Tutorials*, 18(1):623–654.

Selvi, H., Gür, G., and Alagöz, F. (2016). Cooperative load balancing for hierarchical SDN controllers. In *High Performance Switching and Routing (HPSR), 2016 IEEE 17th International Conference on*, pages 100–105. IEEE.

Shang, G., Zhe, P., Bin, X., Aiqun, H., and Kui, R. (2017). FloodDefender: Protecting data and control plane resources under sdn-aimed dos attacks. In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*, pages 1–9. IEEE.

Shin, S., Yegneswaran, V., Porras, P., and Gu, G. (2013). Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 413–424. ACM.

Wang, H., Xu, L., and Gu, G. (2015). Floodguard: A DoS attack prevention extension in software-defined networks. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 239–250. IEEE.

Wang, S., Chandrasekharan, S., Gomez, K., Kandeepan, S., Al-Hourani, A., Asghar, M. R., Russello, G., and Zanna, P. (2018). SECOD: SDN secure control and data plane algorithm for detecting and defending against DoS attacks. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5. IEEE.

Yan, Q. and Yu, F. R. (2015). Distributed denial of service attacks in software-defined networking with cloud computing. *IEEE Communications Magazine*, 53(4):52–59.

Zhou, Y., Zhu, M., Xiao, L., Ruan, L., Duan, W., Li, D., Liu, R., and Zhu, M. (2014). A load balancing strategy of SDN controller based on distributed decision. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 851–856. IEEE.