Requirement Analysis for Abstracting Security in Software Defined Network

Ajay Nehra*, Meenakshi Tripathi†, M.S.Gaur‡
Department of Computer Science & Engineering
Malaviya National Institute of Technology, Jaipur
* Email: 2014rcp9553@mnit.ac.in
† Email: mtripathi.cse@mnit.ac.in
‡ Email: gaurms@mnit.ac.in

***Abstract*—Software Defined Network(SDN) decouples the control plane from the data plane. The aim of this decoupling in SDN helps to create an open programmable network. Programmability offers the capability to write custom network modules i.e. topology discovery, switching, routing, traffic monitoring, access control, etc. Building SDN applications in the primary controller(i.e. Pox, Opendaylight, etc.) configuration is tedious due to low-level programming. The programmability in SDN not only allows flexibility in network management but also introduce new security holes. Indeed the researchers have proposed several abstractions for network management, but we believe the similar abstractions for security is needed to realize the holistic view of SDN fully. In this paper, we review the existing programming model and available abstractions for SDN and show the need for a new security abstraction through an example. We determine that existing abstractions lack the expressiveness for security measures precisely. So there is a need for abstractions which can express the threat detection, mitigation or even prevention by analyzing huge number of logs and can classify them into groups based on their intent.**

***Index Terms*—SDN, Software Defined Network, Security, Language, Abstraction, Network Programming Languages, Domain Specific Languages.**

I. Introduction

Software Define Network(SDN) provides the opportunity to ”Program” the network and allows programmability with the precise motive of ”Openness” irrespective of design and vendor of devices. The openness in the network allows network administrator to customize system as per the developing needs which unquestionably increases the participation. SDN organizes campus or data center network as a set of well trainable devices conventionally known as forwarding elements(Data Plane) which are trained by logically centralized controller(Control Plane). To resolve single point failure and to improve performance, two methods exists. One is hierarchy of controllers which follows master-slave architecture while the other is collaboration of controllers where controllers works parallelly. To fulfill the request of the network, the controller is used to execute the control modules which are network independent and share the entire view of topology. Controller module(s) can either be developed in-house or outsourced to the third party, but it is required to follow OpenFlow specification. Control module(s) can perform various network related activities as routing, switching, load balancing, access control, security configuration, etc. Due to programmability, we may not able to think about some aspects at present but can be explored in future. In the nutshell, typical network activity specifies the network behavior and allows query the network status[1][2].

A network is a distributed system consists of networking elements which may have come from different manufacturers. The forwarding elements must process incoming packet by ”Match” and ”Action”. Currently, OpenFlow specification allows up to 45 different match attributes like source IP, source MAC, destination IP, destination MAC, etc[3]. Incoming packets should be matched with any of defined match patterns i.e. flow entry in the flow table and required action is performed. “Table miss” flow entry specifies the action to be taken for unmatched packets. Every flow entry is associated with a priority which helps to resolve the overlapping match entries. When the controller receives an unmatched packet, it determines the future event for all such packets. Traditional OpenFlow controller provides low-level API for programming. Consider a case where negation has to be applied to a packet from particular IP address to prevent it from entering into the network. In another case, a set of the packet must pass through two modules in a sequence like load balancing module will process only after firewall module. In real-time scenario, these tasks are not easy to program with available low-level APIs. Now high-level abstraction is responsible for network policy and runtime query. Controller handles the complexity of integrating different modules written in high-level API. Few attempts were made to solve these problems like frenetic, pyretic, kinetic, etc[4].

A network module typically lies around specifying packet forwarding policies, monitoring network traffic or updating policies dynamically. Forwarding policies consists of all the requirements for simple forwarding action, traffic shaping, traffic classification, security measures. Monitoring process requires statistics available at DataPlane i.e. port statistics, flow statistics, etc. The purpose of monitoring can be extended either forwarding decision or verification. Dynamically changing policies leads to dynamic reconfiguration of the network and to support this high-level requirement, programming in the available low-level interface is not an optimized idea. To program a fully functional network, several modules have

to be written for routing, switching, access control, security module, traffic monitoring, and so on. To write such module independently in low-level API leads to interference with each other. A race condition is common while programming in low-level API where packet reaches an inconsistent state of rules due to delay in transmission between controller and switches.

An efficient high-level abstraction will consist of high-level operators and operands to think a module at the conceptual level and leave the actual implementation at run-time. Abstraction should allow re-usability and modularity. A race free implementation is expected which should not affect the overall system beyond permissible limits.

This paper discusses the need for security abstraction in SDN. Every abstraction for a given domain has its own coverage area. In this research article, Section II addressing existing abstractions in SDN domain. There are few previous attempts to providing abstractions mainly focusing on forwarding, monitoring, and updating behavior. Security as a behavior is not considered yet in SDN domain. We can write security solutions in existing abstractions, but it may be a voluminous or inefficient implementation. Security in the context of SDN is bifurcated, one covering threats due to SDN architecture while other focusing on a new solution to traditional threats. A brief security survey and a specific threat which is an original problem related to unrestricted programming expressiveness are addressed in Section III. Later in Section IV various network parameters i.e. throughput, delay are discussed to develop the understanding to provide the solution in SDN for discussed Switch DoS threat. OpenFlow-based statistical counters are used to provide solution namely Packet Flow Ratio(PFRatio). Along with PFRatio, FTRatio and ZeroIdealTimeOut security abstractions are introduced to claim that security abstractions plays a great role in writing network policies in Section V.

II. Existing Abstractions

Abstractions are used to reduce programmer's overhead which allows the programmer to focus on the problem at much higher level instead of struggling at low-level details. Few good attempts have been made to provide useful abstractions in the form of Frenetic, Pyretic[5], Kinetic[6], NetCore[7], FlowLog[8], etc. These abstractions mainly focus on writing network policy in the form of predicate rather than a series of shallow "Match" &"Action". Knowing the current status of the network is useful for verification and reconfiguration of the network which also an important aspect for these abstractions. Reconfiguration of a network at run-time is itself a big motivation to develop such abstractions. Formal verification which assures correctness is a major concern too. Verification needs specification first and the same is provided by these abstractions. In this section, some of these abstractions along with syntaxes are presented. Table I shows syntax for these abstractions.

The Frenetic project creates and uses **'see-every-packet'** abstraction to accomplish the task of either network viewing or modeling the behavior. Each query returns time indexed

TABLE I
A few available abstractions

Abstraction Model	Syntax
Frenetic[9]	q = Select(a) * Where(fp) * GroupBy(qh) * SplitWhen(qh) * Every(n) * Limit(n) * a = packets | sizes | counts qh = inport | srcmac | dstmac | ethtype | vlan | srcip | dstip | protocol | srcport | dstport | switch fp = true_fp() | qh_fp(n) | and fp([fp_1 , . . . , fp_n]) | or fp([fp_1 , . . . , fp_n]) | diff fp(fp_1 , fp_2) | not fp(fp) Seconds $\in$ int E SwitchJoin $\in$ switch E SwitchExit $\in$ switch E PortChange $\in$ (switch $\times$ int $\times$ bool) E Once $\in \alpha \rightarrow \alpha$ E $\gg \in \alpha$ E $\rightarrow \alpha\beta$ EF $\rightarrow \beta$ E Lift $\in$ (a $\rightarrow \beta$) $\rightarrow \alpha\beta$ EF $\gg\in \alpha\beta$EF $\rightarrow \beta\gamma$EF $\rightarrow \alpha\gamma$EF ApplyFst $\in \alpha\beta$ EF $\rightarrow (\alpha \times \gamma)(\beta \times \gamma)$ EF ApplySnd $\in \alpha\beta$ EF $\rightarrow (\gamma \times \alpha)(\gamma \times \beta)$ EF Merge $\in (\alpha$E $\times\beta$E) $\rightarrow (\alpha$ option $\times\beta$ option) E BlendLeft $\in \alpha \times \alpha$E $\times\beta$E $\rightarrow (\alpha \times \beta)$ E BlendRight $\in \beta \times \alpha$E $\times\beta$E $\rightarrow (\alpha \times \beta)$ E Accum $\in (\gamma \times (\alpha \times \gamma \rightarrow \gamma) \rightarrow \alpha\gamma$ EF Filter $\in (\alpha \rightarrow$bool) $\rightarrow \alpha\alpha$EF $\gg \in \alpha$E$\rightarrow \alpha$L$\rightarrow unit$ Print $\in \alpha$L Register $\in$ policy L Send $\in$ (switch$\times$packet$\times$action)L Rule $\in$ pattern$\times$actionlist$\rightarrow$rule MakeForwardRules $\in$ (switch$\times$port$\times$packet) policy EF AddRules $\in$ policy policy EF
Pyretic[5]	P = Dynamic() | N | P + P | P $\gg$ P N = B | F | modify(h=v) | N + N | N $\gg$ N F = A | F & F | (F | F) | $\sim$F A = identity | drop | match(h=v) B = FwdBucket() | CountBucket() packets(limit=n, group_by=[f_1 ,f_2 ,...]) count_packets(interval=t, group_by=[f_1 ,f_2 ,...]) count_bytes(interval=t, group_by=[f_1 ,f_2 ,...])
Kinetic[6]	K = P | FSMPolicy(L,M) | K + K | K $\gg$ K L = f : packet $\rightarrow$ F M = FSMDef([var_name=W]) W = VarDef(type,init_val,T) T = [case(S,D)] S = D==D | S & S | (S | S) | ! S D = C(value) | V(var_name) | event
NetCore[7]	Switch s Header h Switch Set $S = \{s_1 , . . . , s_n \}$ Header Set $H = \{h_1 , . . . , h_n \}$ Bit b = 1 | 0 Packet p = $\{h_1 \mapsto \vec{b}_1 , . . . , h_n \mapsto \vec{b}_n\}$ State $\sum = \{\|(s_1 , p_1), . . . , (s_n , p_n)\|\}$ Snapshot x = ($\sum$, s, p) Wildcard w = 1 | 0 | ? Inspector $f \in$ State $[H_1] \times$ Switch $\times$ Packet $[H_2] \rightarrow$ Bool Predicate $e = h : \vec{w}$ | switch s | inspect $e\ f$ | $e_1 \cap e_2$ | $\neg\ e$ Policy $\tau = e \rightarrow S$ | $\tau_1 \cap \tau_2$ | $\neg\tau$
Flowlog[8]	block = ON <id>(<id>) [WHERE rformula] : rules rules = rule | rule rules rule = do_act | ins_act | del_act do_act = DO <id>(termlist) [WHERE rformula] ; ins_act = INSERT (termlist) INTO <id> [WHERE rformula] ; del_act = DELETE (termlist) FROM <id> [WHERE rformula] ; term = <num>| <$string$>| <id>| <id>.<id>| ANY termlist = term | term , termlist rformula = <id>(termlist) | term = term | NOT rformula | rformula AND rformula | rformula OR rformula | (rformula)

stream of packets, sizes or counts. The target traffic Select's on the restriction of predicate provided by Where clause. A GroupBy clause groups queried packets with fields like srcmac, dstmac, srcip, dstip, etc. These groups may be aggregated by the value of one or more given fields. Every clause provides period to get the result and Limit restrict result in term of the count. Few events i.e. SwitchExit, PortChange. etc are created, and Frenetic function will take these events into consideration. Here few event transformation functions are also specified like Lift. ApplyFst, ApplySnd functions which are used for event ordering as per requirement. Merge, BlendLeft, BlendRight function converts a pair of events into a single event with different options. 'Send' sends target packet to a switch and there it performs the required actions. The Register applies a network policy to a network. Few rule related functions are available to create apply rules to target switch[9].

A Pyretic policy can be static(N) or Dynamic(P) or a combination of both. Combinations are also introduced with parallel and sequential compositions of a target streams. Located Packet abstraction is used to apply any policy on the entire network. Few primitive filters (A) and derived filters (F) are made available to target the packet flow and can be applied with available boolean operations.The identity, none and match primitive filters select an original packet, none and as per specification respectively. Buckets are used to collect and count packets. Statistics are generated in the form of packets or count of packet or byte[5].

The Kinetic run-time is designed to run on top of Pyretic run time and it is an extension to Pyretic. Kinetic program uses located packet equivalence class (LPEC) abstraction to avoid state space explosion. The programmer can generate an own set of events top over primitive event supported by OpenFlow. The Kinetic policy may be independent, or a combination of pyretic and kinetic policies with parallel or sequential in order. FSMPolicy is a combination of LPEC (L) and an FSM description (M). A FSM description uses state variable(W) having some type, initial value, and transition function (T). Transition function uses a test (S) and a basic value (D) to store information related to transition from one state to another[6].

NetCore language is based on abstractions like predicates to target packets and policies to forward these targeted packets. This Language has few basic notations in the form of Switch, Header, Wildcard. Set used for these primitive notation and few set operations are defined over them like intersection, complement, union, etc. Packets are defined over bit-stream of the header field. State denotes pairs over particular switch and packet. An Inspector predicate uses a filter predicate to target a set of the packet and a boolean function over this set to concise it. Predicates can be specified either by matching bit-stream to the header field of packets or by using complex set of operations. Policies (τ) used to create the blueprint of how packets should travel. A complex policy also uses set operations or recursive definitions[7].

Alike SQL, FlowLog language declares table for events, state, and interfaces which are used in data manipulation. A rule-set is used to create a set of rules to be applied on trigger specified by ON clause. The rformula provides filter using primitive term or combination of other rformulas. DO, INSERT, DELETE are actions generally used to update any table[8].

Any abstraction model is providing a system by which programmer can specify target efficiently and accurately. Without abstraction, a network administrator has to consider each and every minute detail. It is same as a programmer of time critical system also considering that how a basic print function works. Table II presents the summary of available abstraction along with their covered behavior. Behaviour is a set of instructions to fulfill desired requirements. Various behaviors covered by available languages in SDN are forwarding, monitoring, updating, and security. Forwarding behaviour specifies instruction supporting packets to forward from one port to another. Monitoring behaviour defines instruction to perform a query for network status, device status, etc. Update behaviour gives the instruction set to dynamically update the network which may use a decision based on a network status. Security abstraction will provide the instruction set uniquely considering the security threats more precisely. Security threats may be classified by either threat to the traditional network for which SDN proposing new solutions or threat that is exploiting SDN architecture itself.

TABLE II
SUMMARY TABLE

Behaviour	Forward	Monitor	Update	Security
Frenetic	✔	✔	✔	✗
Pyretic	✔	✔	✗	✗
Kinetic	✔	✔	✔	✗
NetCore	✔	✔	✗	✗
FlowLog	✔	✔	✗	✗

Other abstractions are also available like Procera[10], Nlog[11] etc but we omit the details for granularity. We observed few good abstractions are available but a very little specifically for security. Here security includes security to SDN architecture and network security using SDN.

III. A TYPICAL SECURITY THREAT

Security is an important and non-trivial area of any research domain. All aspects must be considered like is it safe ? can anyhow it will exploit? if yes then what is the optimal and feasible solution. Security with the context of SDN covers two facets. In the first aspect, we are trying to figure out each security threat generated due to SDN architecture itself. The controller is main in this architecture so, what if the controller is compromised? It may happen due to a software bug or may be targeted by an attack. It may cause unintended flow entries, falsified traffic flows, DoS attack. The Possible solution may consist of replication, diversity, verification[12]. Other threat may be due to the compromised network application, where controller translates those network application in corresponding flow entries. This attack may possibly due to

a software bug, malicious app which may cause modify flow entries, flooding in flow entries, falsified traffic flows. This problem can be solved using app attestation, app-controller trust management or verification[12]. OpenFlow vulnerability also cause a threat which persists due to failure to adopt TLS or man-in-the-middle attacks. By this kind of threat, eavesdropping attacks, hijack downstream, capture traffic, reconfigure a switch as a proxy for further attacks may possible. To solve this issue some role-based constraints to the permitted rules or controller switch trust management will help[13]. Switch vulnerability due to software/hardware bug or DoS attack may cause drop/slow down packets, deviate traffic, or even inject traffic. To solve this switch software attestation or trust management or even behavior analysis will help. Other SDN Security Issues like vulnerable northbound API, malicious host/link also persist and can solve by efficient implementation of auto learning, unlearning and relearning[14][12].

Other class of security threat that can be considered in the hope of new and better solutions are those threats from the traditional network which still exist to prevent the best use of the network. SYN flood, ARP poison, ARP flood, Rouge DHCP Server and etc. are few of them. SDN offers programmability that removes the bottleneck of slow innovation due to limited vendor-specific development. However, programmability comes with its cost. Any language is designed to achieve domain-specific goals. It allows expressing requirements and specification to perform specific task. Some time expressiveness of a language create the vulnerabilities that can be exploited by any intended or accidental uses. This section further discusses a problem emerged with expressiveness of language.

Switch DoS(Controller Code BUG): In SDN, there is a setup of a logical controller and forwarding elements. Every time a switch receives a packet, it checks flow table to act accordingly. Flow table consists of flow entries, and flow entries are stored in TCAM based memory which is expensive and fast. HP5406R is having 65535 flow entry space in its TCAM based flow table, while Open vSwitch(OVS)[15] support 256*1000000 flow entries. Each flow entry in flow table consists of Match attributes, Action parameters, and some statistics counters. If there is no matching flow entry in flow table for any packet, A switch initiates a consultation with the controller. The controller pushes forwarding rule as per given logic and available view of the network. To construct a flow entry, the controller needs Match and Action set. Match set defines the matching criteria for incoming packet like source/destination mac, source/destination IP, etc. The action set specifying the action to be performed on matched packet like output on a port, send to the controller, etc.

Consider the Figure 1, host h2 needs to communicate with h4. First packet from h2 reaches the switch s1. If there is no prior communication which also means no flow entry exist in the flow table, the switch stores this packet in the buffer and generates PACKET_IN packet which includes the header of received packet. The controller checks its logic to be performed on the packet_in event. As per policy logic,

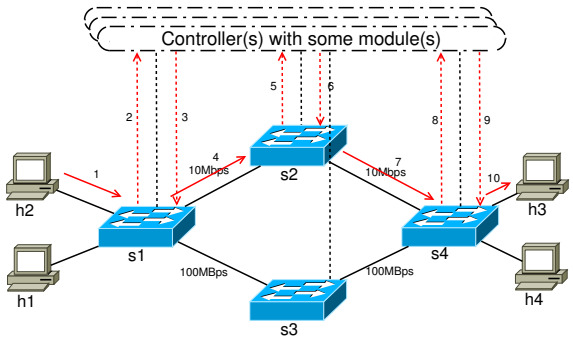

Fig. 1. Topology setup to create attack scenario

controller generate output packet. Output packet may be an MOD_FlOW entry which specifies to add the new table in flow table. Same process repeats on each node like s2 or s3, s4.

At the controller, Match can be specified by either static values or run-time generated values. Sometimes programmer uses the property from received packet rather than hard coding to generate flow entries. There is no problem until an attack occurs, so whenever a valid packet received by the controller, it extracts information required. A new flow entry is created and pushes to a forwarding element. But if an attacker successfully passes packets to the controller with random attributes, and the controller generates a new flow entry for every such random attributed packet. It may be used for flooding the flow table of a particular switch which causes a delay in any genuine request. A sample code from a controller module instance mentioned below. Using hard coding, we can specify the same statement as *msg.match.dltype = 0x0800.*

```
.....
.....
msg = of.ofp_flow_mod()
msg.match.in_port = event.port
msg.match.dl_type = packet.type
msg.match.nw_dst = ip4p.dstip
.....
msg.match.nw_proto = 6
.....
msg.actions.append(action)
event.connection.send(msg)
.....
.....
```

Figure 1 manifests experimental setup for illustrating discussed threat. The controller installs a flow ”h2-s1-s3-s4-h4” to all switches in path. We put ”buggy” code for Switch 3 where control module uses an attribute which is taken from the packet instead of hard coding. The controller uses such attribute to specify Match which completes flow entry in a particular switch. In a non-attack scenario, it works fine but in an attacked scenario, a single line of code in control module can cause burst in the flow table.

We emulate this attack on the mininet[16] environment as well as on an HP5406R[17] switch. Figure 2 gives statistics collected during a particular attack. In the figure, two curves show that flow entries are increasing with time which endorses the attack. Zoomed sub-parts of the graph highlights the exact point at which attack happens in both scenarios. A point at which curves takes a sharp turn indicates the rate of flow entries stored in the switch has sudden growth, is attacking point. In mininet environment and HP5406R, the attack is performed at 31^{st} second and 2^{nd} second respectively. At the time specified we can easily identify sudden growth in the rate. A flow entry which is installed with 0 idle timeout means flow entry will never uninstall due to the timeout event.

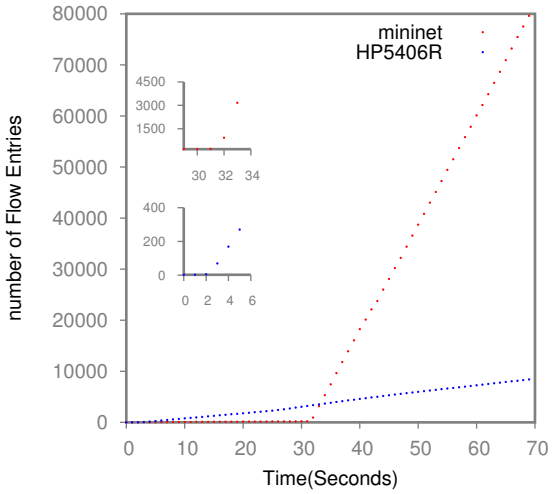

Fig. 2. Attack emulated on the mininet and HP5406R switch

There is a possible argument over why hardware switch HP5406R is slow over mininet(OVS) to store the number of flow entries, while attack rate is same. In mininet every packet is processed without passing through a physical interface, this is not the case with hardware switch environment. So in mininet based emulation, every packet is skipping physical layer(TCP/OSI) but in switch HP5406R each packet is processed by physical layer multiple times.

IV. Research Contribution 1: *A Possible Solution*

The computer network may consist of switches, routers, firewalls and much more. In SDN, each such device is logically made available in the network with the grace of controller although devices are not present physically. Here we have only two things architecture; One is forwarding element which behaves as instructed by a controller which is the other part of the architecture. The controller(s) is logically centralized within an autonomous system. To perform analysis on any such DoS attack various network parameters must be considered as throughput, delay, etc. DoS attack is all about resources, if attacker utilizes its available resources or those resources are heavier than the victim, the victim has to pay.

Throughput is successfully transmitted data over a fixed time which is different from bandwidth. Bandwidth is theoretical maximum data transmission rate. Both throughput and bandwidth sharing the same unit for measurement. So if Throughput(T_n^i) is throughput for network segment 'n' in time stamp i and Bandwidth(B_n) is bandwidth for network segment 'n'.

$$T_n^i < B_n \tag{1}$$

In able to understand the reason for throughput is lesser than bandwidth, there are few delays inserted in the network. Processing delay(P_d) is introduced due to header analysis of any incoming packet. In SDN, packet are matched with pre-stored flow table entries. If a packet is matched with any one of stored entries then corresponding action is performed. If more than one flow entries matched with a packet, higher priority flow entry will be considered. Queuing delay(Q_d) is the spent time by a packet in the queue. In a network, a packet must wait in various queues to allow process the packet which is already arrived at a port(input or output) on forwarding element. Consultation delay(C_d) is also important to the packet for which no matching flow entry exist. This C_d includes time taken for PACKET_IN by a packet to move from switch to controller and time taken in FLOW_MOD packet received by switch(C_{dr}). C_d also includes waiting time in queue for incoming and outgoing(C_{dq}) on port and time taken by the controller to process a request(C_{dc}).

$$C_d = C_{dr} + C_{dq} + C_{dc} \tag{2}$$

The propagation delay(Pr_d) is calculated with the time taken by a piece of information from source to destination. The transmission delay(Pt_d) is time needed to push all the bits stream into the wire which is related to a single packet. Both Pr_d and Pt_d are calculated in SDN same as traditional network.

$$D_t = P_d + Q_d + C_d + Pr_d + Pt_d \tag{3}$$

Suppose the attacker host is sending a packet with t bps. The switch will receive packets with t' bps. A relationship exists between t, t' and bandwidth(B_n) is given by the equation below.

$$B_n \ > \ t \ > \ t' \tag{4}$$

if $t >= B_n$ then packet is dropped and $t' >= t$ is not possible (network with zero delay).

OpenFlow also provides mechanism for storage and retrieval of statistical counters associated with flows, ports, tables. The problem discussed in the last section can be detected if we extract some of those counters. Flows are stored in one among many tables. If we can extract the number of flows and number of packets that matched in the table then we can detect that attack occurs. We use Packet Flow Ratio(PFRatio) to detect aforementioned attack.

$$Packet\ Flow\ Ratio = \frac{Number\ of\ Packets}{Number\ of\ Flows} \tag{5}$$

PFRatio is defined as Number of Packet over the Number of Flows. Equation 6 is determining PFRatio with more efficiency which uses in getting threshold value. Here sum of different packet flow ratio on different times is averaged by a number of samples. If n is 1, then PFRatio is a ratio of the number of packets to the number of flows. If n is more than one, then n samples are taken where a sample is a ratio of packets to flows. These simple ratios are averaged to calculate PFRatio with better precision.

$$PFRatio = \frac{\sum_{i=1}^{n} \frac{\wp_i}{\exists_i}}{n} \quad (6)$$

where $\wp$ is number of packets, $\exists$ is number of flow entries, and n is number of samples

There is a reasonable argument that once a flow is inserted into flow table at least few packets must match the given flow. But as discussed in attack scenario where the attack is made to flood the flow table, so no packets from valid communication will match with those false flows. This makes PFRatio is decreasing continuously i.e. packets are not increasing as flows. Algorithm 1 gives an idea how things can be performed. getObservedThreshold function observes threshold from previous history averaged by either BYWEEK, BYDAYS or BYHRS. The getStatPktCnt and getStatFlwCnt extract statistics from OpenFlow enabled switch.

Algorithm 1 Detecting Flow Table Flood Attack

```
1: function ISFLOWTABLEFLOOD( )
2:     tsld ← GETOBSERVEDTHRESHOLD(BYWEEK | BYDAYS
   | BYHRS)
3:     pktcnt ← GETSTATPKTCNT( )
4:     flwcnt ← GETSTATFLWCNT( )
5:     prf ← pktcnt ÷ flwcnt
6:     if prf < tsld then
7:         return True
8:     end if
9: end function
```

Graphical representation for statistics related to attack detection is shown in Figure 3 & 4. In both mininet and HP5406R setup for attack scenario, all host will ping each other when an attack occurs.

As shown in figure 3, PFRatio is increasing for all the switches except switch 3. As soon as the attack is made at switch 3, PFRatio on switch s3 starts decreasing. As observed in the figure, these cycles attain a zero for all switches because mininet environment resets the counters to zero. Apart from this, we can see that PFRatio of switch s3 will lie on x-axis after the attack happens while PFRatio for other switches are not, which claims that the number of flow entries in switch are increasing while the number of packets matched against those is not. Sub-part in the graph is a zoomed out version to see precisely what is happening with switch s1.

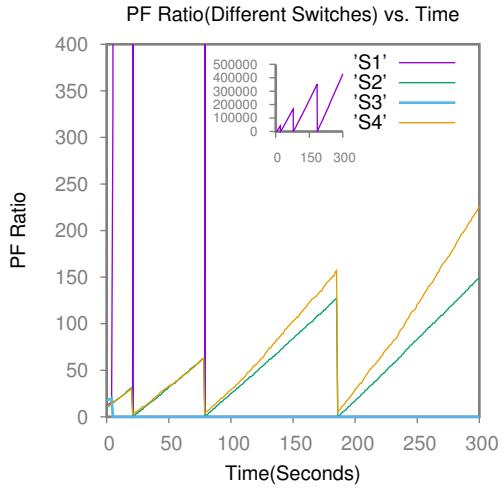

Fig. 3. Attack detection emulated on the mininet

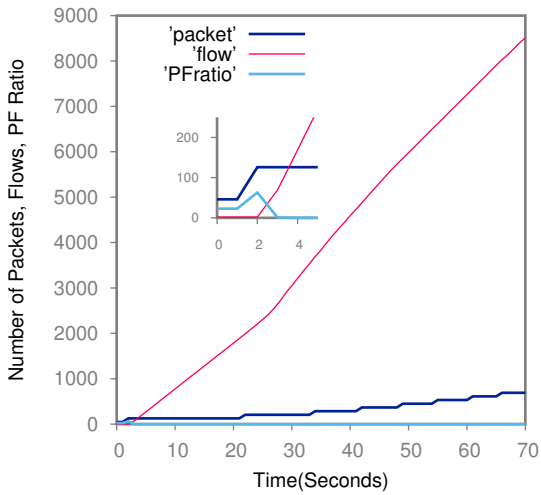

Fig. 4. Attack detection performed on HP5406R switch

In Figure 4, attack is detected on the HP5406R switch where along with the controller one host is added to the switch for performing and detecting the mentioned attack. As flows are increasing, but packets are not with the same pace hence PFRatio reaches to zero and lies on the x-axis. Again sub-part is zoomed in version to see the exact situation at 2 seconds. In the beginning, the number of packets is more than the number of flows. After the attack, while the number of flows is increased rapidly, PFRatio goes to zero which confirms the attack. Figure 5 gives the details of time taken by proposed detection strategy. In mininet and HP5406R switch time taken by the method are 1.129734 seconds and 1.561955 seconds respectively.

V. RESEARCH CONTRIBUTION 2: *A Need of Security Abstraction*

We discussed few SDN language abstractions and also presented a security threat. For detection of such attack, one

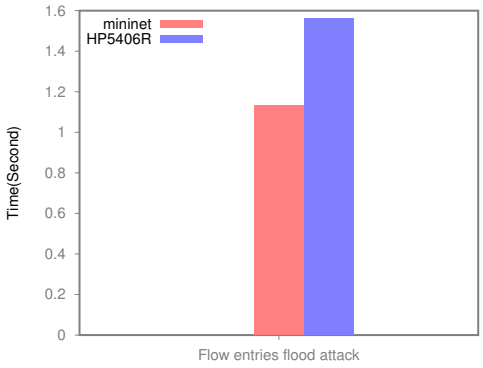

Fig. 5. Attack detection time

has to write in low-level API or use the provided abstraction. In both cases, as there is no explicit security abstraction which undoubtedly leads to writing a voluminous set of instruction. Although generating equivalent flow entries in different switches our production network should not suffer, but a voluminous set of instruction is always prone to manual errors, create a scope for race condition or create instruction which creates interference with other instructions. As we realized that security abstraction is missing and it will be more beneficial if taken into consideration.

---

if PFRatio(prd) < Thrsld :
& Thrsld = BYWEEK | BYDAYS | BYHRS

Take a look at above proposed abstraction which extracts number of flows with the number of packets. Abstraction determines the PFRatio that can be compared with a threshold value. The threshold value is evaluated from past observations averaged over days, weeks or hours.

We can think of many security abstractions which may be based on different security threats. Like if we define FTRatio as number of flow entries in a switch over a fixed time, it will again useful for probability based security threat detection model. This FTRatio will be calculated on particular time samples and if this is greater than previous observed threshold then it indicates possible abnormal behaviour.

---

if FTRatio(prd) > Thrsld :
& Thrsld = BYWEEK | BYDAYS | BYHRS

A flow entry can be removed from flow table by either specific request to remove or on the expiration of one of the timers(idle, hard) associated with it. A flow entry with zero idle timeout will never remove from flow table automatically which also means that it takes space in TCAM even if it is not in use. A security abstraction which gives details about the number and exact flow entries will definitely help a network administrator to write security solutions.

Detail(ZeroIdealTimeOut) :
if Detail(ZeroIdealTimeOut) − c > Thrsld :
& Thrsld = BYWEEK | BYDAYS | BYHRS

Here a question arises that when we can detect such attacks using low-level API, then why there is a need for security abstractions. Then answer lies in the base of SDN openness. The openness allows the programmer to write customized security algorithms. Sometimes language expressiveness creates a problem as in previously discussed threat. Therefore to limit the expressive power of a language in an efficient manner needs abstraction. Verification is a different aspect that can be considered. To verify SDN program against known security threats, then one possible way specifies a program in security abstraction. Therefore one can easily prove that a particular program addresses the known security checks or not. For verification at the initial stage, all attributes that can be used to detect the attack will be identified. Using these attributes an abstraction can be created. The model checker will use to find such attribute in any given abstractions.

VI. Conclusion and Future Work

Abstraction is used to help the programmer, but it should be expressive enough to capture every "emotions" or requirement. This paper discusses few topological and forwarding abstractions. We realized that SDN lacks in security abstraction. We also discussed a security threat example, provided a solution and demonstrated a need for security abstraction. In future, we will also try to generate other security abstractions so security algorithms can be specified and translated to desired OpenFlow entries. For translation, some optimal design of a new controller will be sketched. For verification purpose, some intermediate code representation has to be decided so any existing model checker can be used.

VII. Acknowledgment

Our work is supported by DEiTY Government of India project, ISEA-II in the Department of Computer Science and Engineering at Malaviya National Institute of Technology, Jaipur.

References

[1] Cisco White paper. *Software-Defined Networking: Why We Like It and How We Are Building On It*. 2013. URL: http://www.cisco.com/c/dam/en_us/solutions/industries/docs/gov/cis13090_sdn_sled_white_paper.pdf.

[2] HP Technical white paper. *Software-defined networking and network virtualization*. 2014. URL: http://h20195.www2.hp.com/V2/GetPDF.aspx/4AA5-0092ENW.pdf.

[3] *OpenFlow Switch Specification, Version 1.5.0*, pp. 75–76. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.

[4] Open Networking Foundation. *OpenFlow Switch Specification*. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[5] Christopher Monsanto et al. "Composing Software-defined Networks". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 1–14.

[6] Hyojoon Kim et al. "Kinetic: Verifiable Dynamic Network Control". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015, pp. 59–72.

[7] Christopher Monsanto et al. "A Compiler and Runtime System for Network Programming Languages". In: *SIGPLAN Not.* 47.1 (Jan. 2012), pp. 217–230.

[8] Shriram Krishnamurthi. "Tierless Programming and Reasoning for Networks". In: *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*. PLAS'15. Prague, Czech Republic: ACM, 2015, pp. 42–42.

[9] Nate Foster et al. "Frenetic: A Network Programming Language". In: *SIGPLAN Not.* 46.9 (Sept. 2011), pp. 279–291.

[10] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. "Procera: A Language for High-level Reactive Network Control". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: ACM, 2012, pp. 43–48.

[11] Katta Naga Praveen, Jennifer Rexford, and David Walker. *Logic Programming for Software-Defined Networks*. URL: https://www.cs.princeton.edu/~dpw/papers/xldi-2012.pdf.

[12] S. Scott-Hayward, G. O'Callaghan, and S. Sezer. "Sdn Security: A Survey". In: *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. Nov. 2013, pp. 1–7.

[13] Kevin Benton, L. Jean Camp, and Chris Small. "OpenFlow Vulnerability Assessment". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN '13. 2013, pp. 151–152.

[14] Diego Kreutz, Fernando M.V. Ramos, and Paulo Verissimo. "Towards Secure and Dependable Software-defined Networks". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN '13. 2013, pp. 55–60.

[15] *Open vSwitch*. URL: http://openvswitch.org/.

[16] *Mininet:An Instant Virtual Network on your Laptop (or other PC)*. URL: http://mininet.org/.

[17] *HP5406R Switch: Product documentation*. URL: https://www.hpe.com/h20195/v2/default.aspx?cc=emea_africa&lc=en&oid=7430783.