SecSDN: A Novel Architecture for a Secure SDN

Parjanya Vyas and R. K. Shyamasundar
Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai 400076, India

Keywords: Software Defined Networking, SDN Security, Network Security.

Abstract: Security of SDN has been an important focus of research. Attempts to uncover security vulnerabilities in SDN points to two major causes: (i) Inherent assumption of switches being severely limited in intelligence, (ii) Lack of authentication in the communication between controllers and switches. The assumption that switches have limited intelligence, and can only do the task of packet forwarding, further leads to the inference of switches never being actively corrupt or operated by malicious entities. While such an assumption is reasonable for SDN data centers operated within the bounds of a single organization, it is incorrect for larger scaled inter-networking. In this paper, we propose SecSDN, an architecture and a protocol using repetitive hashing to authenticate the communicating parties, securely verify consistency of flow tables residing inside the switches and detect their malicious behaviour within a predefined constant time frame. Using such a technique, we arrive at an infrastructure that can securely perform functions as envisaged in SDN. We establish the correctness of SecSDN and the simulations show that the overhead incurred is virtually non-existent.

1 INTRODUCTION

Software Defined Networking (SDN) offers a flexible, programmable, and adaptable architectural framework to a spectrum of innovations that can result in widespread practical implementations. SDN has been tested and deployed successfully in many of the major and widespread data centers (Yap, ; Bidkar, 2014). Kreutz et al. provide an excellent summary of SDN and its related research efforts in (et al., 2015).

Controller is the most important component of SDN providing a clear and strict separation of data and control planes. Forwarding switches are deemed to be devices that perform simple forwarding functions as per the flow rules in the flow table. Flow tables are populated by commands issued by the controller and interpreted by the forwarding devices. While such simplicity in functionality is sufficient in a well orchestrated system, it becomes difficult to realize trust among distributed components that are invariably distrusted. Lack of authentication mechanism adds a significant complexity to this problem. Feghali et al., (Feghali et al., 2015), study various security problems due to authentication and the assumptions discussed above. Official OpenFlow specification describes TLS authentication to be an optional feature leading to several manufacturers ignoring it. Switches are usually treated as "dumb forwarding devices" – thus refraining from implementing the crucial security mechanism.

Identifying a malfunctioning or malicious switch in a large scale SDN is an arduous task. Most commonly exercised approach to identify malfunctioning switch is to dump flow tables of all the suspected switches and manually check them for inconsistencies. A compromised switch with adversarial intentions would easily be able to deviate from the protocol. Serious SDN security issues have been nicely summarized in (Scott-Hayward et al., 2013). Major security problems like man-in-the-middle (MitM) attack and forging identity arise due to lack of proper authentication between the controller and switches.

In this paper, we propose an enhanced OpenFlow protocol and a controller architecture keeping the rationale invariant. The proposed architecture ensures a robust 2-way authentication in the communication between the controller and the OpenFlow switch. It further enables verification of the correct implementation of flow table commands by ensuring flow table consistency. These are achieved without losing out on performance that is key to SDN popularity and acceptance. In our architecture, we introduce the notion of 'verification hash' – that is computed using *repetitive hashing*. Verification hash serves dual purpose - It (i) ensures that a data plane switch follows the controller commands correctly maintaining consistency of flow

Vyas, P. and Shyamasundar, R.
SecSDN: A Novel Architecture for a Secure SDN.
DOI: 10.5220/0010575505870594
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 587-594
ISBN: 978-989-758-524-1

tables and (ii) authenticates both the communicating parties - controller and data plane switch. We also provide security guarantees of the solution.

2 SDN SECURITY: THREATS AND CAUSES

Current applications of SDN assume that the network is inside the domain of a single secure organization. The controller and the forwarding switches are trusted and do not need authentication. The data plane devices are assumed to be simple forwarding devices with limited intelligence and computing capabilities. Under these assumptions, SDN could work securely as the possibility of identity forgery and device corruption is no longer extant. Therefore, authentication for communications is kept optional even in the latest version of OpenFlow specification (ONF, 2019) - justified by the dire need of scalability and reduced latency.

In real life, many of these assumptions do not hold. SDNs are implemented in data-centers that define inter-network communication interfaces. In other words, the assumption of the network operating inside the canopy of a single organization fails. Rich feature sets provided by data plane devices contradict the assumption of their limited intelligence. For such inter-networking communications where the involved parties are mutually untrusted, a robust 2-way authentication and verification is necessary to ensure security and integrity of the network.

Scott-Hayward S. et al.,(Scott-Hayward et al., 2013) have explored various security threats possible in current SDN architectures. Brooks et al.,(Brooks and Yang, 2015), present a practical MiTM against OpenDayLight (ODL) SDN controller. Antikainen M. et al.,(Antikainen et al., 2014), show how a compromised OpenFlow switch can be used to attack an SDN. Liyanage et al.,(Liyanage, 2015), review security challenges faced by future Software Defined Mobile Networks (SDMN) and propose an architecture to solve these problems. Khurshid et al., (Khurshid et al., 2013), propose a framework to verify and maintain network-wide invariants in real time in a SDN. A separate multi-layered security module along with IPSec tunnelling has been proposed in (Liyanage, 2015). (Hussein et al., 2016) proposes an entire separate plane dedicated for security. A database defined network (DDN) called 'Ravel'(Wang et al., 2016) has been proposed to implement SDN using databases. To secure the DDN, access controls have been defined in (Glaeser and Wang, 2016) using row level database security.

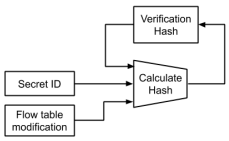


Figure 1: Verification hash using repetitive hashing.

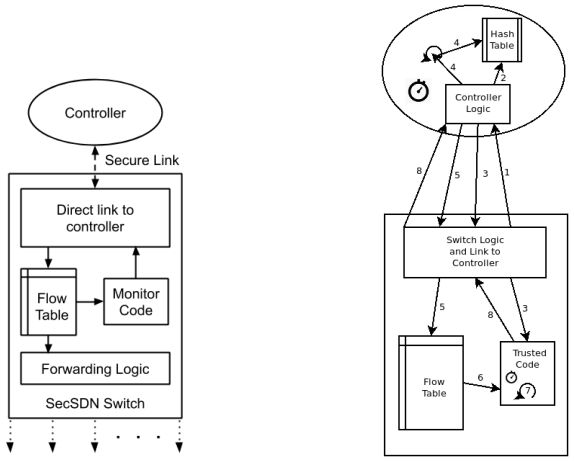
Various solutions to overcome security vulnerabilities of SDN have been realized through insertion of non-trivial modules between control plane and data plane or implementing a wrapper for OpenFlow switches. While each solves a particular issue, it generally takes a long time to detect inconsistency between flow table of a switch and the view of its controller. There is a non-negligible observable overhead resulting in degradation of performance of the switch, controller, and/or the network. Finally, interceptors introduced generally contradict the SDN philosophy of clean separation between control and data plane operations. In summary, the two main issues of security in SDN are: (i) Lack of a robust 2-way authentication protocol. (ii) Actual flow table residing inside the switches, and its view represented in the controller can vary over time. Such inconsistencies are logically incorrect and can be leveraged by attackers to inject malicious flow rules.

We overcome these issues using the concept of verification hash calculated using repetitive hashing depicted in Figure 1. Verification hash provides a robust authentication cum integrity mechanism whenever a data-plane switch and controller communicate. The concept of repetitive hashing ensures consistency of flow tables to prevent any unwanted injections of malicious flow rules in the flow tables of honest switches. Keeping in line with this spirit, we propose "SecSDN", as a novel architecture and a protocol that uses repetitive hashing to ensure security of SDN.

3 SecSDN: A NOVEL ARCHITECTURE

3.1 Realizing Flow Table Integrity

We impose a compulsory acknowledgement scheme on forwarding switches by modifying the existing OpenFlow protocol to verify the consistency of flow tables used by forwarding switches. Controller always maintains a hash of the current flow table of the switch. As and when the flow table changes are done, the controller updates the hash. The controller also keeps track of the changes done in the flow table by the switch, through a trusted monitoring code. Whenever a change in flow table is detected, the monitoring



(a) Switch architecture (b) Protocol Interactions

Figure 2: SecSDN Architecture and Protocol Interactions.

code simply calculates a hash of these latest changes and sends it to the controller. The verification is done by comparing the hash sent by the monitoring code and the hash of the latest flow table present in the controller. The actual expression used to calculate and refresh the hashes is described in section 3.3.

As a byproduct, our hashing mechanism, achieves a 2-way authentication mechanism to authenticate communicating entities - the switch and the controller. By using hashes stored in the controller and switches as server and client nonce, and the identity of the switch as a shared secret, a structure similar to the standard authentication protocol is realized without using any additional packet or a cryptographic mechanism. We argue that the protocol provides a foolproof authentication using which, the *Authentication Property*, is ensured (cf. Theorem 1 in Section 4).

3.2 SecSDN Architecture

To overcome the problem of authentication and consistency verification, a concise way of acknowledgement in terms of a simple proof of correct execution is required. Consistency essentially means the view of the flow table residing in the switch and the view of the flow table present in the controller are the same. For succinctly representing the state of a data object such as the flow table, a fixed length hash can be used. We use such a hash as an acknowledgement by the switches to verify their correct execution of controller commands.

We extend the existing OpenFlow south-bound interface by adding a single field in the packet structure of all the message types used in south-bound

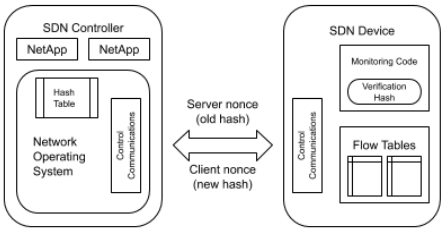


Figure 3: The SecSDN protocol diagram.

interface. We add an extra field called a 'verification hash' in all the types of OpenFlow packets that are from switch to controllerThese packets mainly include types, 'packet-in', 'FlowRemoved', 'GetConfigRes', etc., that are all the message types for communication from switch to controller. Additionally, all the switches in the network will have a small trusted code that can access the flow table directly. The trusted code can either be installed by a trusted manufacturer, or as stated in the protocol later, can be installed by the controller as the result of first communication between the switch and the controller. The broad proposed architecture called SecSDN is shown in Figure 2a.

3.3 Protocol Description

Figure 3 depicts a schematic diagram of SecSDN protocol that improves upon the existing OpenFlow protocol shown in (et al., 2015). Figure 2b shows interactions in the protocol used in SecSDN. SecSDN protocol in described detail along with interactions between the controller and a switch.

SecSDN utilizes the concept of repetitive hashing to represent the flow table states. As shown in Figure 3, every data-plane switch maintains a monitoring code, that contains a verification hash. The controller maintains a hash table that stores all the verification hashes corresponding to the data-plane devices existing in the network. The initial hash is calculated using Formula 1.

$$initial\_hash = Hash(secret\_id) \quad (1)$$

As and when the flow table changes in the switch, the monitoring code updates the recent hash using expression 2.

$$new\_hash = Hash(sec\_id + chngd\_flow\_rule + old\_hash) \quad (2)$$

Whenever the controller needs to send a packet to a switch, it places the old hash present in its flow table with the packet intended for the switch. This hash serves the purpose of a server nonce. The switch places the newly calculated (using the expression 2) verification hash in the response packet. This hash

servers two purposes: (i) it is used for verification of the flow table by the controller and (ii) it serves as a client nonce in the authentication mechanism. Whenever the controller receives a message from the switch, it verifies that the hash present in the packet matches with the hash from the hash table. In case the hashes do not match, the switch is reported as a malfunctioning switch to the networking application (NetApp) and necessary steps are taken as defined by the NetApp.

Interpretation of labelled messages in Figure 2b:
1. A switch is newly connected in the network and establishes a TCP channel with the controller normally.
2. Controller maintains a verification hash corresponding to the latest flow table change for every switch in the network. The initial value of the hash would be the hash of the switch's unique and secret id, which is not known to anyone else apart from the controller and the trusted code. 3. Inside the 'FeatureReq' message that is the first message received by the switch, the controller provides a signed trusted code, that is to be installed on the switch. The trusted code is obfuscated to hide all the secret information and prevent tampering. It is implied that the switch responds with the 'FeatureRes' message that would additionally contain the secure acknowledgement that the code is successfully installed and is not tampered with.
4. Whenever the controller wants to send a packet to the switch for changing its flow rule, the hash corresponding to that switch is updated using Formula 2. The controller also initiates a count-down for checking how long has it been till the change in the hash is acknowledged.
5. The controller sends command for modifying the flow table and switch changes the same accordingly.
6. Trusted code residing in the switch keeps monitoring the flow table and maintains the latest hash.
7. As soon as it detects any change in the flow table made by the switch, it updates its hash in the same way as above. The code also initiates a count-down to measure how long has it been till it has acknowledged the change in hash.
8. Whenever a switch wants to communicate with the controller using one of the message types ('packet-in', 'FlowRemoved', 'GetConfigRes', etc), after creating the packet, it queries the latest hash from the trusted code and puts it in the additional field of 'verification hash' as shown in Figure 2a. Timeout for the acknowledgement is reset when this message is sent to the controller. If the count-down timer goes off without any message being sent to the controller, then a special acknowledgement message containing the updated hash is sent to the controller for verifying the correctness of the flow table by the trusted code.
9. Whenever the controller receives a packet from the switch, before performing the intended task, it first matches the hash residing in the packet with the hash maintained by itself for the particular switch and resets the acknowledgement timer.
10. If the hashes match, the normal operations proceed, else a malfunctioning or adversarial switch is detected and appropriate steps are taken as defined by the policy of the controller.
11. If the acknowledgment timer in the controller goes of without receiving the acknowledgement for the change, then appropriate steps as defined by the management policy are taken to check what is wrong with the switch.

The protocol ensures two important properties that lay the foundation of the security guarantees provided by the architecture. The first property is called *Compulsory Acknowledgement*, which says "Whenever the recent hash is recalculated by an honest OpenFlow switch, an acknowledgement is always sent to the controller within a constant timeframe". This is ensured by the timers presented in the protocol above. The detailed description and proof of this theorem are presented as Theorem 2 in Section 4.

Second, the most important property central to our architecture is the property of *Threat Detection* - "An OpenFlow switch that does not follow the controller commands would always be caught within a constant time frame". This property ensures that a malicious, malfunctioning or misconfigured switch is always detected within a constant amount of time. This time is dictated by the countdown values set in the controller and switch timers. These values also serve as a security parameter to tune in between the performance and security provided by the protocol. Theorem 4 given in Section 4, articulates threat detection and it's proof.

4 SecSDN SECURITY CHARACTERIZATION

Assumptions. SecSDN is secure under the assumption that the SDN controller and the designed TrustCode is secure; this is a natural assumption (similar to assuming the kernel of an OS is secure). We elaborate these assumptions to convince the reader of the naturalness of the assumptions.

Assumption 1 *Controller is Fully Trusted***:** Primary objective of SecSDN is to secure the controller from potentially malicious forwarding switches by securing the south-bound interface assuming the controller is trusted. This is quite a natural assumption in the context of SDN rationale.

Assumption 2 *Monitoring Code is Trusted and Cannot be Tampered***:** The monitoring code is installed by the controller in the forwarding switches as part of their first interaction. The trusted controller installs the trusted monitoring code.

In a sense, the switches present in the network are potentially corruptible entities and include them in our threat model. The monitoring code resides inside the switch, and for the trust of monitoring code to hold, its' integrity must be intact. Therefore, the monitoring code needs to be tamper proof.

Threat Model. As described in Section 4, we completely trust the controller and the monitoring code provided by the controller. The attacks (if any) in SecSDN are thus possible through (i) a switch connected to the SDN; the switch might try and deviate from the protocol, or (ii) eavesdropping communications between the controller and the switch. Thus, attackers are capable of sniffing, hijacking, tampering or replaying packets from/to the switch. Note that the attackers are not capable of breaking crypto-primitives.

Security Characterization.

Theorem 1. *Authentication: A third party entity cannot assume the identity of a genuine OpenFlow switch or the SDN controller once the communication channel is established.*

Proof. Proof depends on two assumptions (i) Secrecy of the switch identity and (ii) Security of the hashing algorithm. Assuming that the communication channel is set up and the hashes are correctly initialized in the OpenFlow switch and the controller, the recent hash itself works as a valid nonce. Properly authenticated client-server communication protocol that provably prevents man-in-the-middle (MiTM) and other such attacks uses client and server nonces.Here, the nonce protect the client and the server from replay attacks by making the message unique. Property of hashing algorithm prevents the output of the freshly calculated hash being repeated. In SecSDN, whenever controller issues a command to the switch (Step 5 in Section 3.3), the current hash of the switch is used as the server_nonce. During acknowledgement (Step 8 in Section 3.3) to the controller the newly refreshed hash is used as the client_nonce in the authentication mechanism. Finally, the secret identity of the switch is used as the secret input to the hash function.

From the above argument, it follows that the message structure and protocol that SecSDN uses, comprise of all the necessary components and functions for a two-way authenticated communication. The security of such mechanism can always be guaranteed if (i) nonces are uniformly random and (ii) they have negligible repeat probability. These follow from the properties of the hash functions that the uniform randomness is guaranteed by the security of the hash function and every time the hash is recalculated, it uses the old hash as an input, making the input, and by definition the output, unique. Hence, the probability that a nonce is repeated is negligible. Therefore, the communication is always authenticated, and a third party cannot assume the identity of an OpenFlow switch or the controller. □

Authentication ensures that a third party cannot forge identity of a genuine SDN controller or switch. It further assures that during an ongoing communication between a controller and a switch, it is not possible for an unauthorized malicious entity to launch MiTM.

Theorem 2. *Compulsory Acknowledgement: Whenever the recent hash is recalculated by an honest OpenFlow switch, an acknowledgement is always sent to the controller within a bounded time.*

Proof. Here, we use the assumption of reliability of the communication channel and the trusted nature of the monitoring code. Whenever a hash is recalculated in a switch (Step 7 in Section 3.3), a timer is started by the monitoring code. There are two possible scenarios: (i) If there is a message to be sent to the controller by the switch before the timer goes off, then the recent hash is piggybacked with this message and sent to the controller as a piggybacked acknowledgement (Step 8 in Section 3.3). (ii) If the timer goes off, without any message sent to the controller by the switch, then monitoring code creates a special acknowledgement packet along with the recent hash and sends it to the controller (Step 8 in Sec. 3.3).

Hence, in both the cases above, an acknowledgement is always sent to the controller within a constant time frame dictated by the timer residing in the monitoring code. □

Compulsory acknowledgement property assures that changes done in the flow table by an honest switch is always sent to the controller as an acknowledgement. The acknowledgement might be delayed by a bounded time to improve the overall network performance, but eventually it will always be sent.

Theorem 3. *Consistency: Recent hash maintained inside the monitoring code of an honest OpenFlow switch and the one maintained inside the controller would eventually always match within a bounded time.*

Proof. The hashing function used in the controller and the switches is the same. An initial value of the recent hash is calculated by hashing the secret identity (after Step 3 shown in Section 3.3), which is provided

by the controller. Hence, the initial hash calculated by the controller (during Step 2 in Section 3.3) and the switch would be exactly the same. The recent hash resides inside the monitoring code and is isolated from the environment. It can be modified only by the monitoring code. Hence, the only time the hash changes is when it is changed by the monitoring code (Step 7 in Section 3.3). The trusted and verified nature of the monitoring code along with the authentication property proved in Theorem 1 forbids any entity other than the authenticated controller to change the recent hash. Hence, the hash only changes when the flow table is required to be changed, which in turn is commanded by the authenticated SDN controller.

Whenever the controller issues a command to change the flow table, it recalculates it's own recent hash using the expression defined in step 4 of the protocol description. This is exactly the same expression used by the monitoring code to recalculate the recent hash in the switch when a change in the flow table is detected. According to Theorem 2, the recalculated recent hash will always be sent to the controller for an acknowledgement; hence, both the versions of the hash will match, as soon as the modification is complete in both - the controller and the switch.

Steps above cover all possibilities when the most recent hash is changed either in controller or switch. Given the reliability of the communication channel, hashes in the controller and switch will always match and hence the theorem. □

Consistency property ensures that the actual flow table residing inside the switch and its view present in the controller match. Recent hash is a unique representation of the state of the flow table and assuring the matching of hashes, automatically verifies that the actual flow table and its view in the controller are same.

Theorem 4. *Threat Detection: A switch that does not follow the controller commands will always be caught within a bounded time.*

Proof. A dishonest switch can only evade the controller commands in three possible ways: (1) It modifies the flow table wrongly (during Step 5 of Section 3.3), (2) It discards the controller command altogether (The latter part of Step 5 in Section 3.3 is dropped), and (3) It prevents the monitoring code from communicating with the controller (The latter part of Step 5 in Section 3.3 is dropped).

If the flow table is modified differently, then the hash calculated by the monitoring code will not match the hash calculated by the controller, due to the property of the hashing function and the uniqueness guaranteed by the hashing formula. The switch, in this case, is caught almost instantaneously.

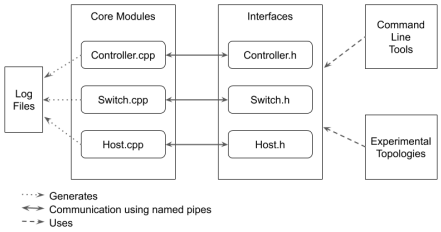


Figure 4: A schematic diagram of secure SDN simulator.

If it discards the controller command without modifying the flow table (scenario 2) or prevents outgoing communication from the monitoring code (scenario 3), the timer started in the controller will eventually go off without receiving an acknowledgement from the switch. Hence, the switch will be caught in the time frame defined by the controller timer, that is constant.

A scenario where the outgoing communication from the monitoring code is captured and corrupted or modified is not possible because the authentication mechanism requires a valid hash to be presented to the controller along with the acknowledgement. This valid hash can only be calculated if the switch identity is known, that is assumed to be a secret. Hence, the integrity of messages between the controller and the switch cannot be broken without immediate detection. □

Threat detection property states that a malicious or malfunctioning switch that wrongly populates a flow table is eventually always detected within a bounded time.

5 PERFORMANCE EVALUATION

We have built a proof of concept of SecSDN as a secure SDN simulator. SecSDN simulator works in two different modes: (i) normal and (ii) secure. In normal mode, the authentication and verification mechanisms described in the earlier sections are disabled. In secure mode, the protocol described in Section 3.3 is activated and the consistency of the flow table is verified using SHA256 hash. A schematic diagram depicting the architecture is shown in Figure 4.

We use appropriate metrics defined in (Isaia and Guan, 2016) for benchmarking our simulator. We evaluate our implementation using three different topologies with variable number of nodes (N) or switches with N = 10, 50 and 100 for both 'normal' and 'secure' cases. Topology 1 shown in Figure 5a corresponds to measuring performance when there is a bottleneck link. Variable number of messages that

Table 1: Performance measurements for Topology T1.

	N=10		N=50		N=100	
	Normal	Secure	Normal	Secure	Normal	Secure
Total flows	10	10	50	50	100	100
Total time to add all flows	434	655	2497	2824	4250	5392
Total network data sent to add all flows	272	912	1474	4674	3074	9474
Total packets sent to add all flows	20	20	100	100	200	200
Total messages sent	10	10	50	50	100	100
Total message propagation time	2599	2564	11852	11599	24972	23287
Average flow length	4	4	4	4	4	4
Average time to add one flow	43.4	65.5	49.94	56.48	42.5	53.92
Average network data sent to add one flow	27.2	91.2	29.48	93.48	30.74	94.74
Average packets sent to add one flow	2	2	2	2	2	2
Average message propagation time	259.9	256.4	237.04	231.98	249.72	232.87

Table 2: Performance measurements for Topology T2.

	N=10		N=50		N=100	
	Normal	Secure	Normal	Secure	Normal	Secure
Total flows	19	19	99	99	199	199
Total time to add all flows	1463	2319	27908	51183	113587	173597
Total network data sent to add all flows	1529	6233	39099	127003	165846	501750
Total packets sent to add all flows	109	109	2549	2549	10099	10099
Total messages sent	19	19	99	99	199	199
Total message propagation time	3096	3248	44476	40892	164229	154357
Average flow length	7	7	27	27	52	52
Average time to add one flow	77	122.05	281.9	517	570.79	872.347
Average network data sent to add one flow	80.47	328.05	394.94	1282.86	833.4	2521.36
Average packets sent to add one flow	5.74	5.74	25.75	25.75	50.75	50.75
Average message propagation time	162.95	170.95	449.25	413.05	825.27	775.66

Table 3: Performance measurements for Topology T3.

	N=10		N=50		N=100	
	Normal	Secure	Normal	Secure	Normal	Secure
Total flows	9	9	49	49	99	99
Total time to add all flows	345	537	1768	2491	3658	5173
Total network data sent to add all flows	118	406	717	2285	1468	4636
Total packets sent to add all flows	9	9	49	49	99	99
Total messages sent	9	9	49	49	99	99
Total message propagation time	1592	1529	8535	7482	17333	17260
Average flow length	3	3	3	3	3	3
Average time to add one flow	38.33	59.67	36.08	50.84	36.95	52.25
Average network data sent to add one flow	13.11	45.11	14.63	46.63	14.83	46.83
Average packets sent to add one flow	1	1	1	1	1	1
Average message propagation time	176.89	169.89	174.18	152.69	175.08	174.34

use the bottleneck link are sent simultaneously. The performance of SecSDN is measured for 10, 50 and 100 messages for this scenario. Topology 2 shown in Figure 5b corresponds to measuring performance when the topology is 'linear' and there are flows of varied lengths present in the network. In the experiments with value of N = 10, 19 flows with minimum length of 3 to maximum length of 12 are present. Similarly for N = 50, 99 flows of length 3 to 52 and for N = 100, 199 flows of length 3 to 102 are present in the network. Topology 3 shown in Figure 5c corresponds to measuring performance when the topology is 'star' and there is maximum load on a single switch. In this case, all flows are of length 3 but all of them passes through the same switch. In the experiments with N = 10, 50 and 100 a single switch maintains connection with the controller and 10, 50 and 100 hosts and manages 9, 49 and 99 flows respectively.

Topologies are shown in the figures 5a, 5b, 5c and their experimental results are shown in tables 1, 2, 3. Time is measured in microseconds (μs) unit and the network data is measured in bytes (B).

5.1 Discussion

Figure 6 and Figure 7 show bar charts depicting comparison of average time to add one flow and comparison of average message propagation time for a single message of 512 bytes in the topologies described earlier. As seen from the results in the Table 4 and chart shown in Figure 7, the message propagation and message forwarding are completely independent of the secure nature of the topology. The protocol only affects the setup phase of the network when flows are being initialized. The piggybacking used in the protocol design is highly effective as clearly seen from the fact that number of network packets required for initializing flows remains exactly the same for SDN and SecSDN setups.

Table 4: Overhead incurred in SecSDN vs SDN.

Topology	Total switch count	Overhead Factors			
		Flow setup time	Flow setup data	Flow setup packets	Message propagation time
T1	20	0.51	2.35	0	-0.01
	100	0.13	2.17	0	-0.02
	200	0.27	2.08	0	-0.07
T2	147	0.59	3.08	0	0.05
	2747	0.83	2.25	0	-0.08
	10497	0.53	2.03	0	-0.06
T3	9	0.56	2.44	0	-0.04
	49	0.41	2.19	0	-0.12
	99	0.41	2.16	0	0

As the number of flows and the average length of the individual flows increase, the number of network bytes sent to initialize these flows increase in SecSDN. This is so since every flow table modification in a switch requires an acknowledgement with recalculated hash sent to the controller. SHA256 is used to generate a 32 bytes hash per flow table modification. Hence, every time a flow is added, all the switches present in the path of the flow needs to send an extra 32 bytes to the controller as an ack. This means, as and when the flow length and the number of flows increase, the network bytes sent to add these flows would also increase with a factor of 32 bytes per switch in all the flows. But compared to the real time data with size of thousands of megabytes that usually flows in the networks, this amount is negligible in a large scale SDN topology. As a result of increase in data bytes while initializing flows, time to initialize flows will also increase linearly as seen from Figure 6. Every time a flow is initialized, the controller as well as the switch need to calculate a new SHA256 hash for every switch in the flow path. The additional hash computations, and the time taken to transmit the 32 bytes hash would result in additional time for flow initialization.

6 CONCLUSION AND FUTURE WORK

In this paper, we have envisaged SecSDN as an efficient architecture using an enhanced OpenFlow protocol that authenticates and securely verifies consistency of flow tables in switches across SDN using repetitive hashing. It is shown that SecSDN realizes security in SDN via a simple robust authentication,

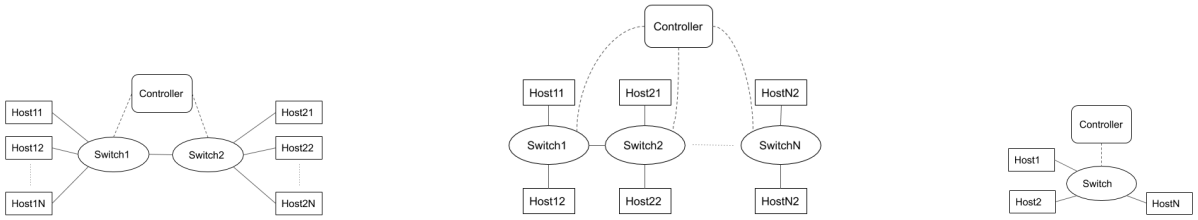


Figure 5: Topologies chosen for experimental evaluation.

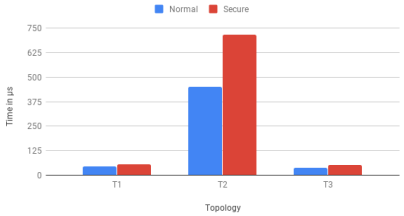


Figure 6: Average time to add one flow in SecSDN vs SDN.

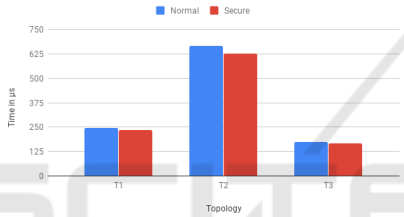


Figure 7: Avg message propagation time: SecSDN vs SDN.

maintaining a clean separation between control and data plane operations. It detects a malicious switch in a constant time - bound by a timer value - that serves as a security parameter to tune between desired security and performance. Our initial simulations show that the overhead incurred is virtually non-existent – thus showing the power of SecSDN both in theory and practice. In short, SecSDN provides a good architectural basis for building secure SDNs.

To sum up, SecSDN architecture provides a promising proven concept towards building a secure SDN as compared to other approaches.

ACKNOWLEDGMENTS

The work was done at the Information Security Research and Development Centre (ISRDC) of Indian Institute of Technology Bombay, sponsored by the Ministry of Electronics and Information Technology, GOI.

REFERENCES

Antikainen, M., Aura, T., and Särelä, M. (2014). Spook in your network: Attacking an sdn with a compromised openflow switch. In *Secure IT Systems*, pages 229–244. Springer Int. Publisher.

Bidkar, S. e. a. (2014). Field trial of a software defined network (sdn) using carrier ethernet and segment routing in a tier-1 provider. In *IEEE Global Communications Conference*, pages 2166–2172.

Brooks, M. and Yang, B. (2015). A man-in-the-middle attack against opendaylight sdn controller. In *4th ACM Conf. on Research in Information Technology*, RIIT '15, pages 45–49.

et al., D. K. (2015). Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76.

Feghali, A., Kilany, R., and Chamoun, M. (2015). Sdn security problems and solutions analysis. In *Int. Conf. on Protocol Engineering (ICPE) and Int. Conf. on New Technologies of Distributed Systems (NTDS)*, pages 1–5.

Glaeser, N. and Wang, A. (2016). Access control for a database-defined network. In *2016 IEEE 37th Sarnoff Symposium*, pages 1–2.

Hussein, A., Elhajj, I. H., Chehab, A., and Kayssi, A. (2016). Sdn security plane: An architecture for resilient security services. In *2016 IEEE Int. Conf. on Cloud Engineering Workshop (IC2EW)*, pages 54–59.

Isaia, P. and Guan, L. (2016). Performance benchmarking of sdn experimental platforms. In *2016 IEEE NetSoft*, pages 116–120.

Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. (2013). Veriflow: Verifying network-wide invariants in real time. In *10th {USENIX} {NSDI} 13*, pages 15–27.

Liyanage, M. e. a. (2015). Security for future software defined mobile networks. In *9th Int. Conf. on Next Generation Mobile Applications, Services and Technologies*, pages 256–264.

ONF (2019). Sdn technical specifications.

Scott-Hayward, S., O'Callaghan, G., and Sezer, S. (2013). Sdn security: A survey. In *2013 IEEE SDN For Future Networks and Services (SDN4FNS)*, pages 1–7.

Wang, A., Mei, X., Croft, J., Caesar, M., and Godfrey, B. (2016). Ravel: A database-defined network. In *Proc. of the Symposium on SDN Research*, page 5. ACM.

Yap, K.-K. e. a. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proc. ACM SIGCOMM 2017*, pages 432–445.