CHAPTER 3

Development of a Blockchain-enabled flow integrity in the OpenFlow networks

3.1 Introduction

The adoption of SDN is going to save error-prone manual configuration by enabling dynamic automation through a central controller, thus improving network flexibility. In the OpenFlow protocol, a forwarding device matches the incoming traffic with the flow rules and takes action as specified in the matching flow rule. However, the flexibility and programmability offered by SDN controllers create a potential vulnerability where malicious actors can deceive security applications, leading to the installation of malicious flow rules on OpenFlow switches. Moreover, the attacker can exploit vulnerabilities in the control plane to modify or redirect network traffic, leading to security breaches and data leakage. The controller is the only device that can determine whether a flow rule has been tampered with. So, the attacker with a rogue controller will try to tamper with the flow rules in the switches and cause network failure.

The existing works in the literature mainly emphasize on maintaining a switch version number to preserve the integrity of flow rules. Consequently, in the event of an unauthorized flow modification attack, the switch needs to update its original flow rules from the blockchain. This leads to a substantial surge in verification time and controller overhead particularly as the frequency of attack increases. The controller latency plays a pivotal role in enhancing network responsiveness, reliability, and overall performance in SDN environments. This highlights a significant research gap in the field, necessitating the exploration of more efficient strategies to preserve the integrity of flow rules provided verification latency is minimal. Therefore, in this chapter, we plan to develop robust and effective security mechanisms to mitigate malicious flow manipulation attacks on the SDN switches.

The rest of this chapter is organized as follows. In Section 3.2, we discuss our system model and architecture of the proposed model in detail. We also give a detailed description of the SC design based on the Ethereum Private Network and the flow verification process. Section 3.3 presents the experimental result analysis and discussion of the results. Section 3.4 presents the concluding remarks and future directions.

3.2 Proposed Model

In this chapter, we introduce FTISCON (Flow Table Integrity Through Smart Contract), a comprehensive OpenFlow security mechanism to preserve the integrity of the flow rules on SDN switches that leverage blockchain technology. We created a specialized Blockchain Agent responsible for verifying flow rules before their installation on forwarding devices. This module operates independently of the SDN controller, reducing the processing burden on the controller and significantly expediting the flow rule verification process. We implement Smart Contracts as a key component of FTISCON, streamlining the process of flow rule verification through the blockchain network. These contracts establish a trust framework that safeguards against malicious manipulation of flow rules by potential attackers, ensuring the integrity of network configurations. Finally, we conduct a comprehensive simulation-based experimental study of FTISCON using the Mininet network emulator. This evaluation encompasses a range of performance metrics, including detection rate, execution time, transaction cost, and delay. Additionally, we conduct a time complexity comparison with existing security techniques BlockFlow [53], BlockSDSec [16], and FRChain [101] to highlight the advantages of the proposed model in terms of computational efficiency. These protocols specifically focus on improving the security and integrity of OpenFlow rules in Software-Defined Networks (SDNs) using blockchain-based mechanisms, which makes them directly relevant to our study. A detailed description of the proposed model is given below.

3.2.1 Architecture of the proposed FTISCON

The architecture of the proposed model consists of the following modules- an SDN Controller, a Blockchain Network, a Smart Contract (SC), and a Blockchain Agent as depicted in Figure 3.1. We briefly explain each module below-

a) **SDN Controller:** The SDN controller maintains the topology information, forwarding rules from a central point. The controller can dynamically adjust the network's routing policies as the network changes. The controller also communicates with the Blockchain node to store the flow information through the WEB3 API.

b) **Blockchain Network:** The Blockchain network consists of multiple peer-to-peer nodes that participate in the consensus for verification of OpenFlow rules. The number of Blockchain nodes depends upon the application type. The more the number of Blockchain nodes, the better the security of the system. However, in the Ethereum Private Blockchain Network, it is recommended to have a minimum bound of seven nodes as a validator. This will allow up to two nodes to be either malicious or go offline. Each Blockchain node on the network maintains a copy of the blockchain ledger and participates in the consensus process. The nodes validate the transactions by following the consensus algorithm such as Proof-of-Work(PoW), Proof-of-Stake(PoS), etc. to create the next block.

c) **Smart Contract:** The role of smart contracts in the blockchain is to enable trustless transactions and automated execution of contractual agreements. They allow for the creation of secure, tamper-proof, and transparent agreements between parties, which are executed automatically when certain conditions are met. A single SC is used for the proposed model. It maintains the rules, topology information, and logic for the verification of flow rules. All the functionalities and business logic are governed by the SC in a decentralized manner.

d) **Blockchain Agent:** The Blockchain Agent is a separate entity that is responsible for collecting the flow rules from the OpenFlow switches and forwarding them to the blockchain network to perform the verification through SC. It is also responsible for removing flows from the switches if found invalid by the blockchain. This module is kept separate from the SDN controller to avoid Single-Point-of-Failure (SPOF). Even if the controller fails, the Agent can do the verification of the flows.

3.2.2 Smart Contract Design

A smart contract is a program that automatically executes a task when predefined conditions are triggered on the blockchain. The main objective of SC is to simplify and automate the operations of flow rule verification without the need for a centralized third party. All parties can perform the transaction in a trustless manner, solely depending on the digital contract. The data

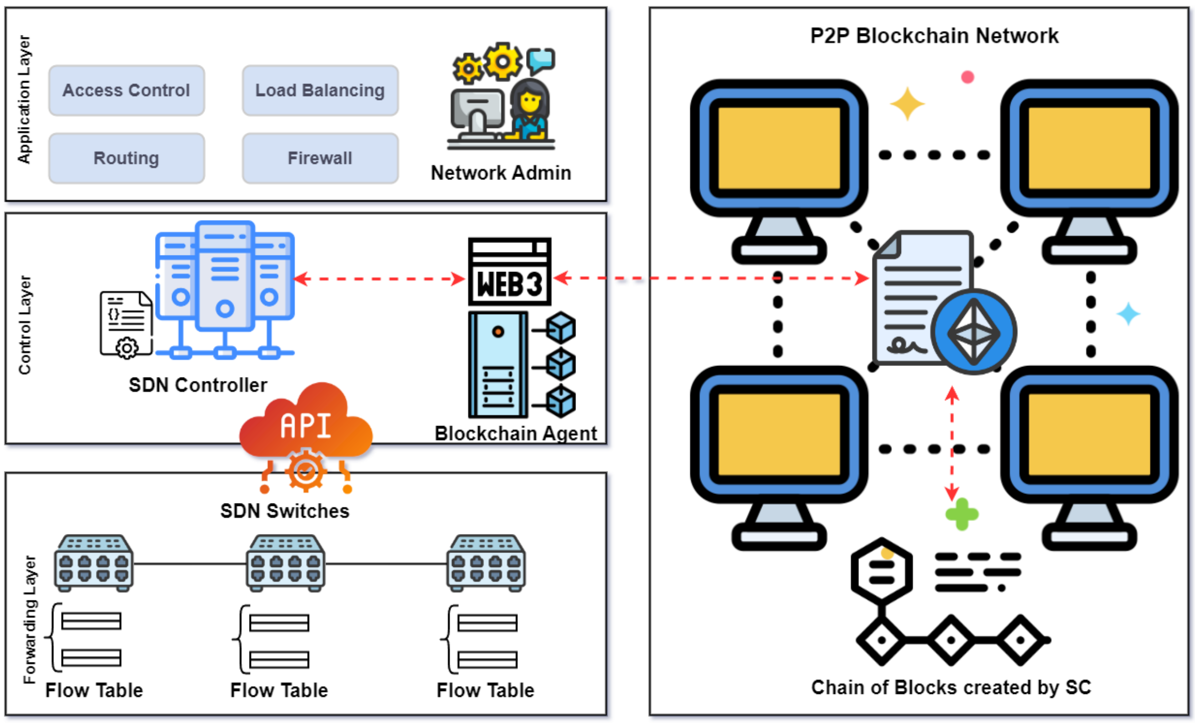


Figure 3.1: Proposed Blockchain-Enabled SDN flow security System Architecture.

stored on the blockchain is shared between the validators to provide total transparency of the transactions to all parties.

We maintain two structures for storing the topology and flow rule information. The SC allows the creation and update operation on the blockchain for both topology and flow tables through the SC functions.

The switch structure consists of the following fields in the SC- **switch_id**, **switch_name**, and **flows**. The flow rule structure has the following fields- **switch_id, flow_id, src_mac, dst_mac, protocol, inport, outport, and priority**.

The different functions available in the SC are discussed below:

a) **addSwitches:** This function is invoked by the SDN controller as soon as a new switch joins the network. It stores the dpid of the switch on the blockchain. Algorithm 1 represents the steps required for switch registration in the blockchain. Switches register themselves by providing datapath_id and transaction timestamps.

b) **addRule:** As the user tries to communicate over the network, the controller identifies the route to the destination in the form of OpenFlow rules. This function is invoked by the SDN controller to store a new flow rule on the blockchain.

c) **verifyRule:** The blockchain agent invokes this function to verify the forwarding rules. Upon receiving this function call, the blockchain peers perform the consensus to validate the transaction. Then, it performs the similarity of the hashes with the flow information received from the agent and already stored hash. If it matches the hash value, then the flow verification is successful; otherwise, an event is generated by the SC to remove the flow from the switch.

Table 3.1: Notation Table

Symbol	Definition
C	SDN Controller
BA	Blockchain Agent
S_i	OpenFlow enabled SDN switch
IP_c	IP address of SDN Controller
W_c	Ethereum Wallet address of SDN Controller
W_{ba}	Ethereum Wallet address of Blockchain Agent
ts	The time at which the transaction was initiated
src	Source MAC
dst	Destination MAC
$inport$	Ingress port number
p	Priority of the flow rule
$dpid_i$	Datapath ID of SDN Switch s_i
tn_id	Unique identifier for the transaction
$drop$	Packet is blocked by the controller
$outport$	Packet is transmitted through the given outport
$normal$	Packet is transmitted as traditional Layer 2 convention
f_{old}	Flow rule already present on the blockchain
f_{new}	New flow rule request for verification

3.2.3 System Initialization

Let $S = \{s_1, s_2, ..., s_m\}$ be the set of switches in the SDN network where each s_i consists of the $dpid$ that is used to specify the switch is being modified.

Let the set of all transactions be denoted as $T = \{tn_1, tn_2, ..., tn_m\}$, where each transaction tn_i takes place between the SDN and blockchain network. Since the flow rules are a combination of match and action fields, let us denote $M(tn_i)$ as the set of match fields in transaction tn_i.

$M(tn_i) = \{m_1, m_2, ..., m_n\}$ where each $m_i = \{src, dst, p, inport\}$. We have used these four match fields to match the packet. However, we can also use the remaining fields to match the packet. Similarly, let $A(tn_i) = \{a_1, a_2, ..., a_n\}$ be the action field specified in transaction tn_i where each $a_i = \{drop|outport|normal\}$.

Let $F = \{f_1, f_2, ..., f_n\}$ be the set of existing flow rules in the network, where f_i is the set of match field m_i and action field a_i i.e. $f = \{(m_1, a_1), (m_2, a_2), ..., (m_n, a_n)\}$.

The transaction between SDN and Blockchain can occur in three scenarios: i) During the network components registration ii) During the addition of new flow rules and iii) During the verification of flow rules.

3.2.4 Threat Model

The threat model of the proposed model is illustrated in Figure 3.2. Our threat model encompasses potential risks on the SDN switches from various perspectives. Firstly, the normal flow of communication between the SDN controller and switch is marked using the green dashed line (Figure 3.2 Circle Mark 1). Next, we account for the threat of an attacker intercepting OpenFlow messages between the SDN controller and switches to execute an attack (Figure 3.2 Circle Mark 2). Additionally, we consider the scenario where an attacker may deploy a rogue controller to orchestrate a malicious attack on the SDN switches (Figure 3.2 Circle Mark 3). Lastly, we consider the scenario of attackers exploiting various APIs to inject malicious flows (Figure 3.2 Circle Mark 4). Furthermore, we have identified and formulated three specific types of attacks that a potential attacker could carry out on the SDN switches. These are detailed as follows.

a) **Flow Modification Attack:** An attacker attempts to modify an existing flow rule on a switch by changing its action and match field. Therefore, when an attacker initiates modification transaction tn_i with modified match field m'_i and action a'_i, the flow becomes: $(m_i, a_i) \rightarrow (m'_i, a'_i)\ \ for\ i = 1, 2, ..., n$. Then, the set of modified flow rules will be-

$$f' = \{(m_1, a_1), (m'_i, a'_i), ..., (m_n, a_n)\} \tag{3.1}$$

This can be mathematically expressed as:

$\forall (m_i, a_i) \in f' : \exists (m'_j, a'_j) \in f'$ such that $(m_i = m_j)$ and $(a_i \neq a'_j)$

The above formulation states that for every original flow rule (m_i, a_i) in the OpenFlow switch, there exists a modified flow rule $(m_i = m_j)$ in the set f' such that the action field is modified.

b) **False Flow Insertion Attack:** In this attack, the attacker tries to install a false flow rule on the switch, containing a malicious action. This exploit has the potential to trigger an overflow attack.

When the attacker initiates the false flow insertion attack, a new flow rule (m'_i, a'_i) is added to the switch. Therefore, the flow (m'_i, a'_i) must be in the updated set of flow rules f' along with the existing flow rules. The updated flow table on the switch is given by the below equation:

$$f' = \{(m_1, a_1), (m_2, a_2), ..., (m_n, a_n), (m'_i, a'_i)\} \tag{3.2}$$

This can be mathematically expressed as:

$\forall (m_i, a_i) \in f' : \exists (m'_j, a'_j) \in f'$ such that $(m_i \neq m_j)$ and $(a_i \neq a'_j)$

c) **Flow Deletion Attack:** Here the attacker attempts to remove an existing flow rule from the switch. Let, the flow rule to be deleted as (m'_i, a'_i) where m'_i represents the match field and a'_i represents the action field. Therefore, when this attack is executed, the corresponding flow rule (m'_i, a'_i) must not be present in the original flow table of the switch. The updated flow table on the switch is given by the below equation:

$$f' = \{(m_1, a_1), (m_2, a_2), \ldots, (m_{n-i}, a_{n-i})\} \tag{3.3}$$

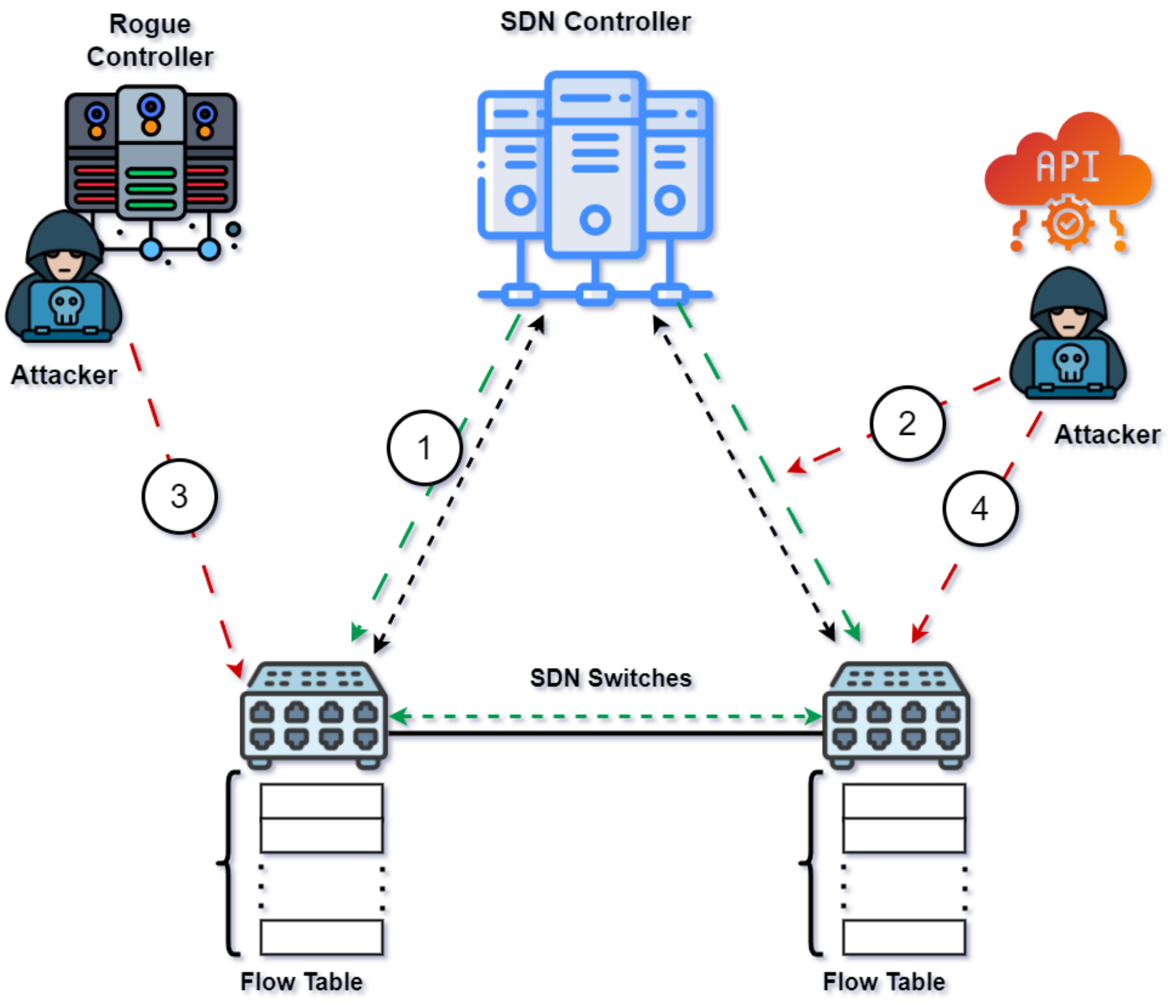


Figure 3.2: Threat Model of flow rule modification in SDN network.

3.2.5 Transaction Formulation

3.2.5.1 Network Component Registration

The SDN Controller and Blockchain Agent create their wallet address W_c and W_{ba} respectively. They register themselves on the SC using the $tn(C)$ and $tn(BA)$ respectively. Similarly, the SDN switch is added to the blockchain using the $tn(s_i)$ transaction as per the algorithm 1. The transactions are given below:

$$tn(C) = \{tn_{id}, W_c, IP_c, ts\} \tag{3.4}$$

$$tn(BA) = \{tn_{id}, W_{ba}, IP_{ba}, ts\} \tag{3.5}$$

$$tn(s_i) = \{tn_{id}, W_c, IP_c, dpid_i, ts\} \; \forall \; s_i \; \in \; S \tag{3.6}$$

The transaction $tn(C)$ and $tn(BA)$ will run only once but $tn(s_i)$ will run for each SDN switch in the network. The SC checks the duplication of the switch and then stores the s_i in the $SWList$.

Algorithm 1 addresses the process of automating the registration of OpenFlow switches on the blockchain. The smart contract (SC) verifies both the switch's dpid and the timestamp associated with the registration request. It then assesses whether the dpid is already registered by cross-referencing it with the existing dpid entries. This algorithm holds significant importance in the blockchain network's validation of flow rules, as it determines the legitimacy of switch registration.

Algorithm 1 OpenFlow Switch Registration Function

Input: Datapath ID **dpid**, Registration timestamp **ts**, Switch List in the SC **SWList**

```
1: Set status ← False
2: SC will check (dpid, ts)
3: if dpid ∈ SWList then
4:     return status
5: else
6:     status ← True
7:     return status
8: end if
```

3.2.5.2 New Flow Rule Registration

Let f_{old} be the existing flow rule on the blockchain and f_{new} be the new flow rule that needs to be registered on the blockchain. Then,

$$f_{old} = \{m_{old}, a_{old}\} \text{ and } f_{new} = \{m_{new}, a_{new}\}$$

For the addition of a new flow rule on the blockchain, the *BA* sends the transaction as:

$$tn(C) = \{tn_id, W\_c, IP\_c, f\_new, ts\} \tag{3.7}$$

When a new flow registration requests $AddFlow(tn(C))$ is received by the blockchain, the SC verifies that the flow rule specified in f_{new} does not conflict with existing flow rules in the network. Therefore, the transaction $tn(C)$ is broadcasted to the network and is verified by the SC. If the transaction is valid, the new flow rule is added to the flow table of the specified switch, and the updated state of the flow table is recorded on the blockchain.

Flow rule f_{old} and f_{new} are in conflict when the match field of both flows overlap but actions specified by the two rules are different. We can express this condition as:

$$m_{old} \cap m_{new} \neq \emptyset \tag{3.8}$$

$$a_{old} \neq a_{new} \tag{3.9}$$

The condition in equation 3.8 signifies that there is a potential conflict between f_{old} and f_{new}. Equation 3.9 signifies that the action specified by the two rules is ambiguous. Therefore, if the above two conditions are satisfied then the f_{new} will not be added to the blockchain (f_{old} and f_{new} are in conflict).

Algorithm 2 Flow Rule Registration Function

Input: Datapath ID **dpid**, New flow rule to be added $\mathbf{f_{new}}$, Controller and Blockchain Agent wallet address $\mathbf{W_c}, \mathbf{W_{ba}}$

```
1:  Set status ← $False$
2:  if $(m_{old} == m_{new}) \lor (a_{old} == a_{new})$ then
3:      return status
4:  end if
5:  Compute $h = sha256(abiencode(f_{new}, W_c, W_{ba}))$
6:  if $dpid \in$ SWList then
7:      if Rules[$h$] $\neq$ $h$ then
8:          Rules[$h$] ← $f_{new}$)
9:          status ← True
10:     end if
11: end if
12: return status
```

Algorithm 2 concentrates on automating the registration of flow rules on the blockchain network. Initially, the blockchain nodes verify the similarity between the OpenFlow match and action fields of new and old flow rules. If a conflict is detected, the transaction is rolled back.

Otherwise, the transaction is reverted as there is a conflict between the old and new flow rules. Then, the algorithm proceeds to compute the hash and securely store the flow rules on the blockchain. This implementation significantly boosts efficiency in storing new flow rules by leveraging hash calculations, ultimately minimizing verification delays.

3.2.5.3 Verification Transaction

The BA sends the request $VerifyFlow(tn(BA))$ to the blockchain for each flow present on the SDN switch to preserve the integrity and avoid any potential flow modification attack. The transaction for verification request is expressed as:

$$tn_{new}(BA) = tn_{id}, W_{ba}, IP_{ba}, f_{new}, ts\} \tag{3.10}$$

The SC receives $tn_{new}(BA)$ containing match (m_{new}) and action (a_{new}) for the verification. It then compares these fields with the transaction history (tn_{old}) associated with the flow rule to determine if there is a similar flow rule already in the blockchain ledger. The conditions are expressed as:

$$\exists!\, m_{new} \in tn_{old} \tag{3.11}$$

$$a_{old} = a_{new} \tag{3.12}$$

Equation 3.11 signifies that the match field m_{new} exists exactly once in tn_{old} and equation 3.12 signifies that both the actions are equivalent. If such a transaction exists, the function returns true, indicating that the f_{new} is similar to an f_{old}. If no such transaction exists, the function returns false.

Algorithm 3 Flow Rule Verification Function

Input: Datapath ID **dpid**, New flow rule to be added $\mathbf{f_{new}}$, Controller and Blockchain Agent wallet address $\mathbf{W_c}, \mathbf{W_{ba}}$

```
1:  Set status ← $False$
2:  if $(m_{new} \notin tn_{old}) \vee (a_{old} \neq a_{new})$ then
3:      return status
4:  end if
5:  Compute $h = sha256(abiencode(f_{new}, W_c, W_{ba}))$
6:  if $dpid \in$ SWList && $W_{ba} \in Agent\_Addr$ then
7:      if Rules[$h$] = $h$ then
8:          status ← $True$
9:      else
10:         emit event Invalid()
11:         return status
12:     end if
13: end if
14: return status
```

The algorithm 3 outlines the steps carried out by the SC function in the flow verification process. Firstly, it validates equations 3.11 and 3.12 and applies a hashing code to assess the integrity of the flow rules. If any alterations are detected in the flow, the SC function will alert the administrator, advising them to eliminate the respective flow from the OpenFlow switch by generating an SC event. This step ensures that all the OpenFlow switches have the correct flow rules installed on them and function as intended, adding an essential layer of security and reliability to the network operation.

3.2.6 Flow Rule Verification in FTISCON

The Blockchain Agent periodically collects the flow rules from the switches. For every new *PACKET_IN* message, the controller will send a copy of the Blockchain network. The algorithm 3 is executed through a Blockchain Network. Therefore, proof-of-correctness is done through the Consensus algorithm (Proof-of-Work in our case). The Blockchain nodes reach

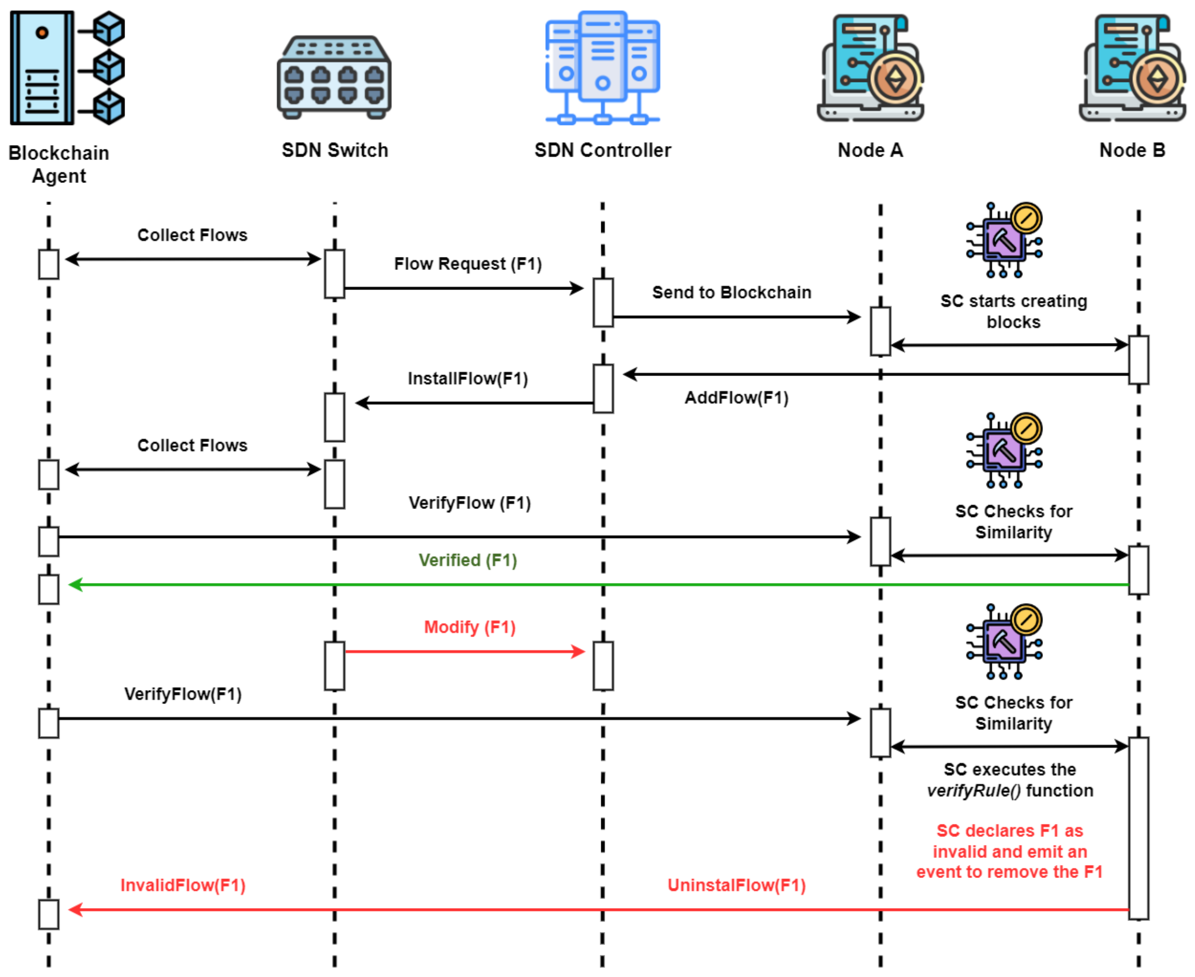


Figure 3.3: Sequence diagram of exchange messages for flow rule verification.

an agreement by following the consensus algorithm. Once the consensus is successful among the nodes, a new block is added to the chain containing flow rule information. Subsequently, the controller will send *add_flow* command to the OpenFlow switch. The Agent then calls SC function $verifyFlow(F)$ to check the similarity between the flows installed on the switch and Blockchain. If flow verification fails due to a flow modification attack, the SC will automatically emit an event to remove flow from the switch. Figure 3.3 illustrates the sequence diagram of message exchange between nodes.

In the following steps, we elaborate on the process of flow rule verification in detail:

a) Initially our Blockchain Agent collects the OpenFlow rules from the switches in a certain time interval.

b) When a particular host requests a new flow for communication, the SDN switch requests a new flow F1.

c) Simultaneously, the controller also transmits a duplicate of the flow rule to the blockchain, creating a transaction.

d) This transaction is verified by the blockchain nodes and the corresponding flow rule is stored in the blockchain.

e) Now the SDN Controller sends the flow rules for installation on the switch.

f) The Blockchain Agent initiates the *VerifyFlow(F1)* on the Blockchain network.

g) The SC checks for the similarity between the received flow F1 and the flow present on the blockchain. Once the flow is successfully verified, a positive response is sent back to the BA (This is denoted with the green line in Figure 3.3).

h) The modified flow information is sent to the SDN controller attack for the new flow verification.

i) The BA repeats the steps f and g. Once SC declares that $F1$ is invalid, a negative response is sent back to the BA (Denoted with a red line).

3.3 Experimental Results

This section describes the experimental setup, SC analysis, and evaluation based on the results obtained by the proposed model.

3.3.1 Environment Setup

This subsection provides an overview of the simulation environment for both SDN and Blockchain networks. To simulate the SDN network we utilize the Mininet framework to create virtual switches and hosts which implement Open vSwitch as a software switch. The SDN controller is implemented using a Python-based Ryu controller. The controller utilizes the OpenFlow V1.3 to communicate with the data plane. However, the proposed model is designed to be compatible with all subsequent versions of the OpenFlow, including but not limited to OpenFlow V1.3. The controller makes the network discovery using LLDP (Link Layer Discovery Protocol) and stores the global overview of the entire network. We run the Blockchain Agent on a separate virtual machine to communicate with the blockchain.

A private Blockchain network is created using the Ethereum framework. We chose private blockchains because they are much faster, cheaper, and have higher scalability. We adopted PoW as the consensus algorithm for the proposed model. The corresponding wallet accounts are created to make transactions on the network. These Blockchain nodes can execute the functions present in the SC for which it requires Ethers in their account. We implemented the SC using the Solidity programming language and deployed it on the Blockchain network. The specifications of the network components are shown in Table 3.2.

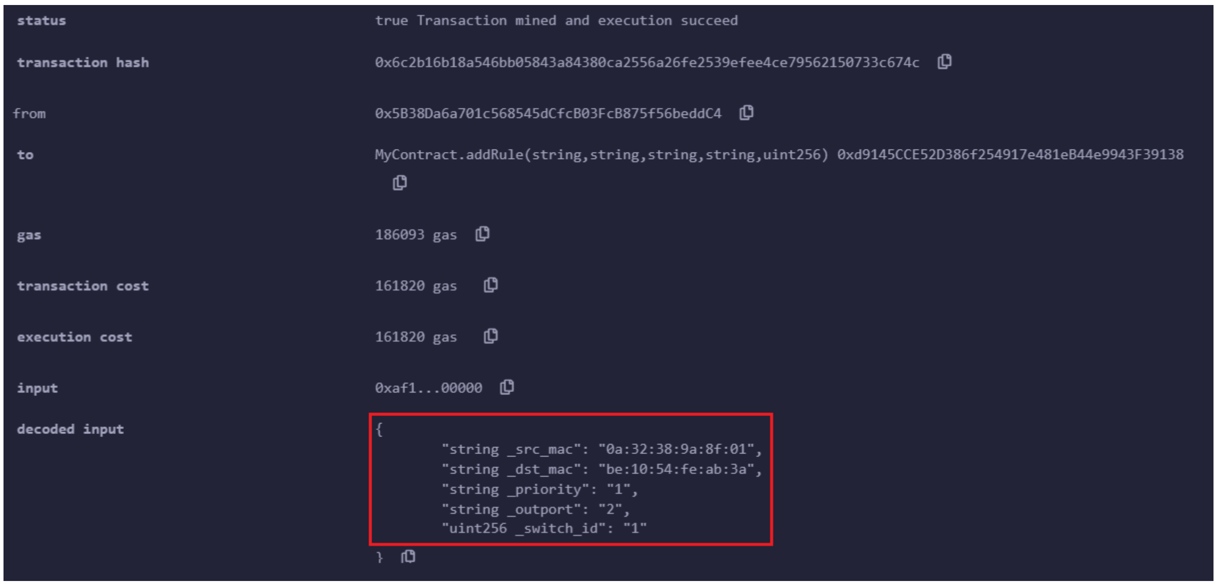
Table 3.2: Experimental Setup Table

SDN Network			
Content	**Name**	**Version**	**Quantity**
Controller	Ryu	4.34	1
South Bound API	OpenFlow	1.3	-
North Bound API	POSTMAN	v9.4	-
Switch	OpenVSwitch	2.16.x	5
End Host	Ubuntu	18.04 LTS	2
Blockchain Network			
Smart Contract	Solidity	0.4.22	1
Nodes	Geth	v1.11.4	5
Consensus	PoW	-	-

3.3.2 Smart Contract Deployment

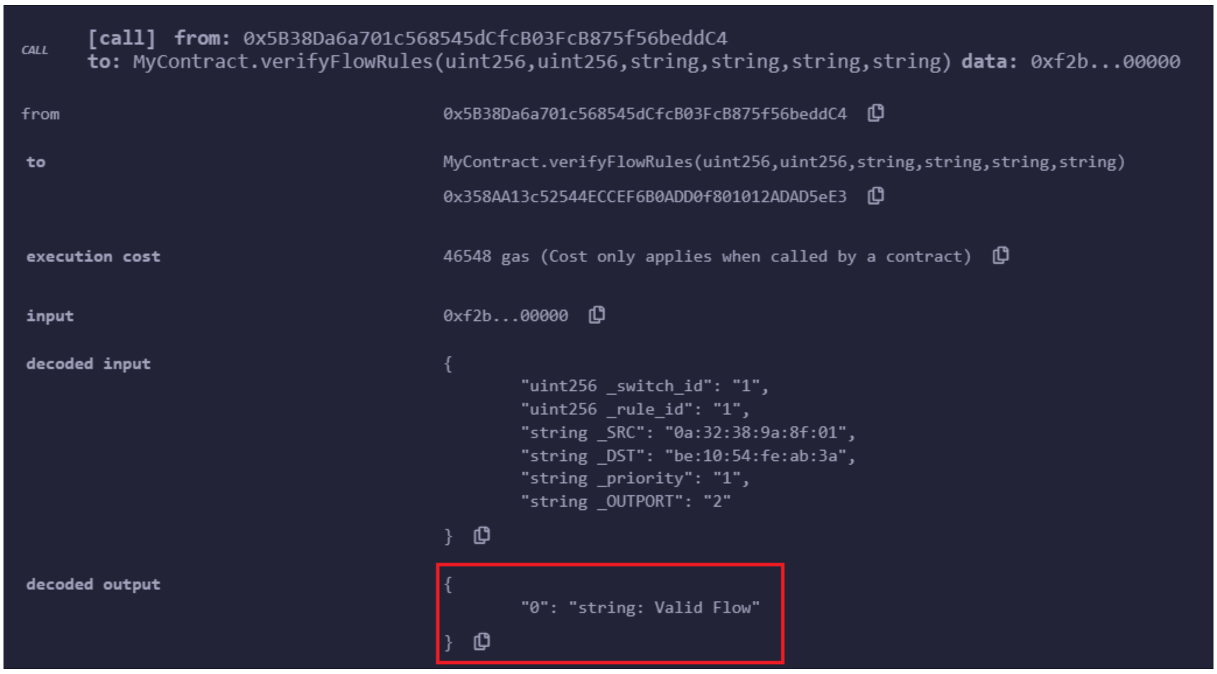
In this subsection, we discuss the deployment of SC in remix IDE(A tool to develop, deploy, and administer the SC on the browser) to visualize the transactions on the blockchain. Figure 3.4 depicts the execution of the $addRules()$ function in a SC. The SDN controller sends a transaction to the Blockchain network as soon as it detects a new switch in the network. Then the SC executes the $addRules()$ function and does the necessary authentication of the transaction.

The Blockchain Agent verifies the flow rules installed on the switch by calling the $verifyRule()$ function on the SC. Figure 3.5 depicts the successful verification of flow rules.



```
status              true Transaction mined and execution succeed
transaction hash    0x6c2b16b18a546bb05843a84380ca2556a26fe2539efee4ce79562150733c674c
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  MyContract.addRule(string,string,string,string,uint256) 0xd9145CCE52D386f254917e481eB44e9943F39138
gas                 186093 gas
transaction cost    161820 gas
execution cost      161820 gas
input               0xaf1...00000
decoded input       {
                            "string _src_mac": "0a:32:38:9a:8f:01",
                            "string _dst_mac": "be:10:54:fe:ab:3a",
                            "string _priority": "1",
                            "string _outport": "2",
                            "uint256 _switch_id": "1"
                    }
```

Figure 3.4: The execution of addRules() function in Smart Contract.

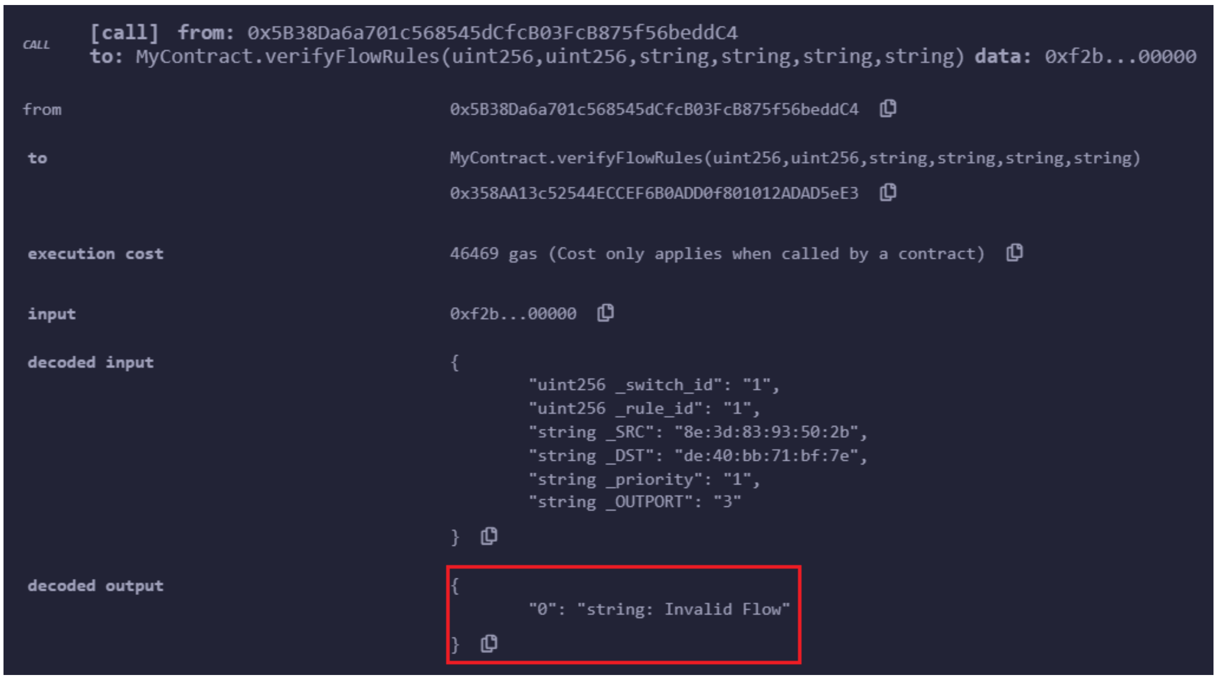


```
CALL    [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
        to: MyContract.verifyFlowRules(uint256,uint256,string,string,string,string) data: 0xf2b...00000

from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  MyContract.verifyFlowRules(uint256,uint256,string,string,string,string)
                    0x358AA13c52544ECCEF6B0ADD0f801012ADAD5eE3
execution cost      46548 gas (Cost only applies when called by a contract)
input               0xf2b...00000
decoded input       {
                            "uint256 _switch_id": "1",
                            "uint256 _rule_id": "1",
                            "string _SRC": "0a:32:38:9a:8f:01",
                            "string _DST": "be:10:54:fe:ab:3a",
                            "string _priority": "1",
                            "string _OUTPORT": "2"
                    }
decoded output      {
                            "0": "string: Valid Flow"
                    }
```

Figure 3.5: The execution of valid verifyRule() function in Smart Contract.

We modify the output port of flow_id one to three, then call the $verifyFlow()$ function. Figure 3.6 shows that the SC successfully detects the malicious flow modification.



```
CALL   [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
       to: MyContract.verifyFlowRules(uint256,uint256,string,string,string,string) data: 0xf2b...00000

from                 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

to                   MyContract.verifyFlowRules(uint256,uint256,string,string,string,string)
                     0x358AA13c52544ECCEF6B0ADD0f801012ADAD5eE3

execution cost       46469 gas (Cost only applies when called by a contract)

input                0xf2b...00000

decoded input        {
                             "uint256 _switch_id": "1",
                             "uint256 _rule_id": "1",
                             "string _SRC": "8e:3d:83:93:50:2b",
                             "string _DST": "de:40:bb:71:bf:7e",
                             "string _priority": "1",
                             "string _OUTPORT": "3"
                     }

decoded output       {
                             "0": "string: Invalid Flow"
                     }
```

Figure 3.6: The execution of invalid verifyRule() function in Smart Contract.

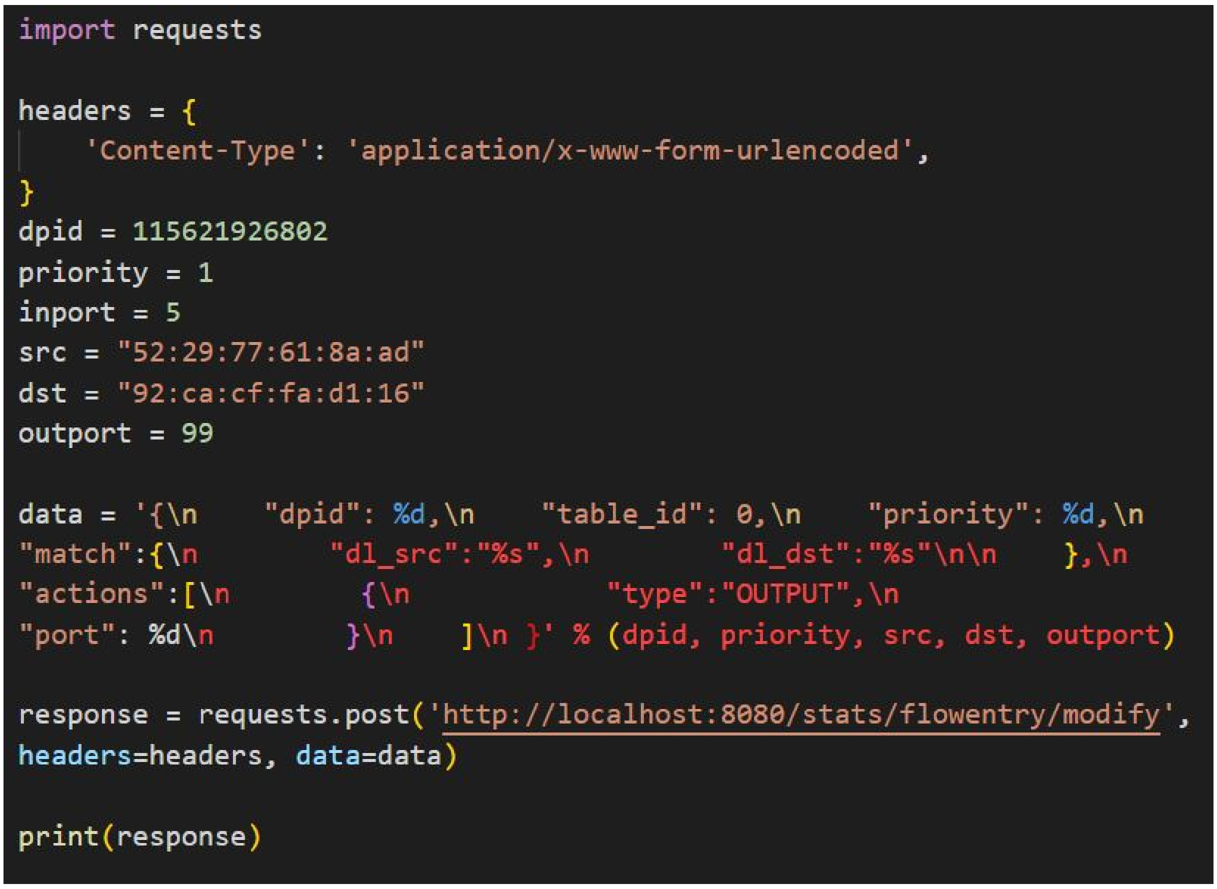
3.3.3 Performance Evaluation

In this subsection, we present the performance evaluation of the FTISCON in terms of Detection Rate, Execution Time, Delay, and Transaction Cost incurred due to the execution of SC functions. We have implemented the BlockSDSec [16], BlockFlow [53], and FRChain [101] methods and used the same configuration to compare the performances. We perform the experiment for **ten** iterations for each analysis and plot the average result.

To test the performance of FTISCON, we deploy our SC on the Ethereum Virtual Machine (EVM) running on Windows 10 Pro Dell PC Intel(R) Xeon(R) W-2145 CPU, 3.70GHz 64GB RAM machine. In the next subsection, we discuss the analysis of the proposed model.

3.3.3.1 Detection Rate Analysis

We analyze the integrity of the proposed model by modifying the flow rules explicitly by using the FLOW_MOD command from the POSTMAN API which modifies the existing flow rules on the switches. A sample of flow modification code is depicted in Figure 3.7 in which the output port of the OpenFlow switch is modified to 99. Moreover, the matching field can be made blank to modify the entire table on the switch.



```
import requests

headers = {
    'Content-Type': 'application/x-www-form-urlencoded',
}
dpid = 115621926802
priority = 1
inport = 5
src = "52:29:77:61:8a:ad"
dst = "92:ca:cf:fa:d1:16"
outport = 99

data = '{\n    "dpid": %d,\n    "table_id": 0,\n    "priority": %d,\n
"match":{\n        "dl_src":"%s",\n        "dl_dst":"%s"\n\n    },\n
"actions":[\n        {\n             "type":"OUTPUT",\n
"port": %d\n        }\n    ]\n }' % (dpid, priority, src, dst, outport)

response = requests.post('http://localhost:8080/stats/flowentry/modify',
headers=headers, data=data)

print(response)
```

Figure 3.7: A sample of synthetic flow modification attack generated from POSTMAN API.

To evaluate the accuracy of the attack detection rate we compare the number of real-time alert events generated in the SC for each flow verification transaction. Therefore, the attack detection rate of the FTISCON can be expressed as

$$DR = \frac{No.\,of\,Alert\,Event\,in\,SC}{No.\,of\,FLOW\_MOD} \times 100 \tag{3.13}$$

We applied the threat model to verify the superiority of the proposed model over the existing work and tested the ability to preserve the integrity of flow rules using the proposed model. We measure the detection rate of all four methods using the same Blockchain configuration. We tested two scenarios where the attacker manipulates the match fields and another scenario where the action field is manipulated. Table 3.3 shows that all four methods can detect the attack successfully in both cases and delete the flows from the switch. However, the time required to remove the flows from the switch in the BlockFlow method takes comparatively

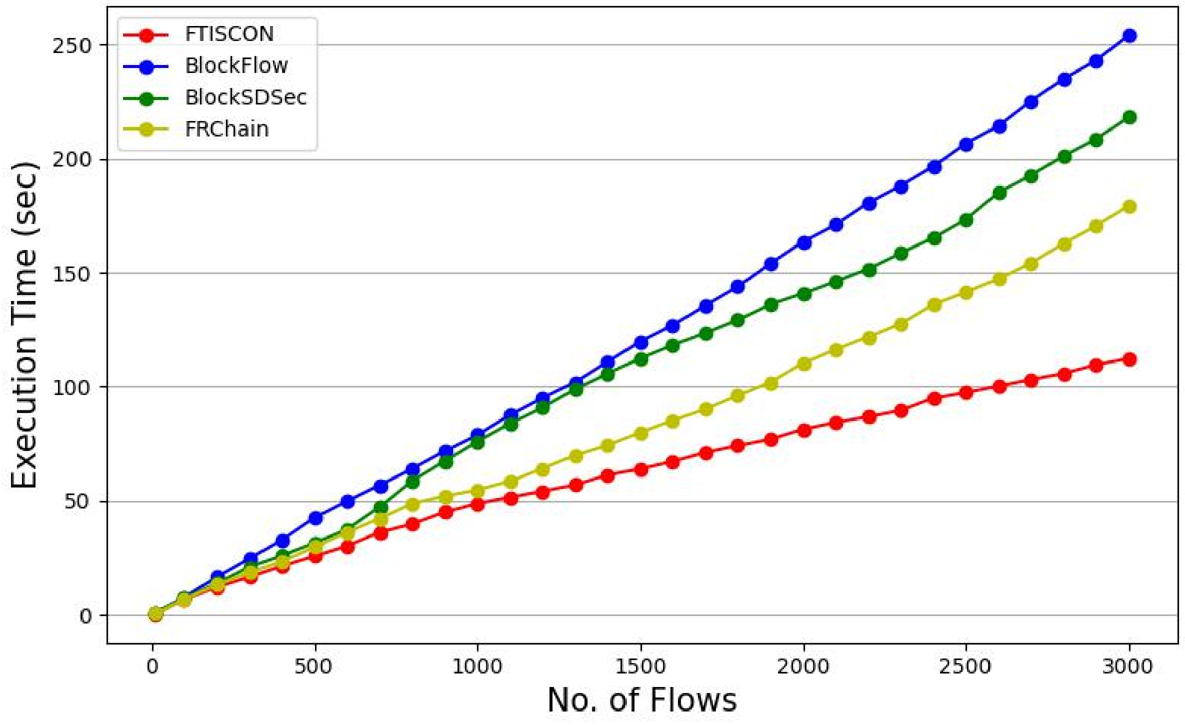


Figure 3.8: Execution time comparison with respect to the number of Flows.

more time than our method.

3.3.3.2 Execution Time Analysis

We evaluate the system performance to find out the impact of the network size on the verification of flow rules on the blockchain. We scale up the number of OpenFlow rules up to 3000 to evaluate the time required for verification. Figure 3.8 shows the results.

For a very small number of flows, all four schemes take about the same amount of time. But as the number of flows grows, the time taken by FTISCON tends to taper down a bit, while for BlockFlow it continues to grow linearly. For BlockSDSec the time taken remains close to that of BlockFlow. This is because the Controller has to delete the flow rules and re-install the entire rule set on the switch as soon as it detects a new switch version. The delay in this process makes these schemes slower. For a higher number of flows, FTISCON takes 52.36%, 48.62%, and 35.67% less execution time compared to BlockFlow, BlockSDSec, and FRChain respectively.

Table 3.3: Detection and Delay Comparison Table

Methods	Detection rate(%)	Delay (sec)		
		max	min	avg
FTISCON	100	0.09	0.02	0.04
FRChain [101]	100	1.66	0.11	0.73
BlockFlow [53]	100	1.75	0.52	1.09
BlockSDSec [16]	100	1.56	0.36	0.94

3.3.3.3 Delay Analysis

The computational delay involved in the operations on the Blockchain is as follows:

$$D = SD_i + VD_i + InstD_i \tag{3.14}$$

Where SD_i is the time required for storing flow i on the blockchain, VD_i is the verification delay of flow i and $InstD_i$ is the time required for installing flow i on the switch. We measure these delays during execution and compute the average delay for each of the schemes as shown in Table 3.3.

3.3.3.4 Transaction Cost Analysis

Another factor that needs to be analyzed in the Blockchain network is the transaction cost. We use the standard Gas Price set by Ethereum (i.e. 21000 gwei) to make the transaction.

The equation of transaction cost is expressed as:

$$Transaction\,Cost = gas\ price\ \times\ gas\ used \tag{3.15}$$

Figure 3.10 shows the result of transaction costs incurred by various SC functions. The

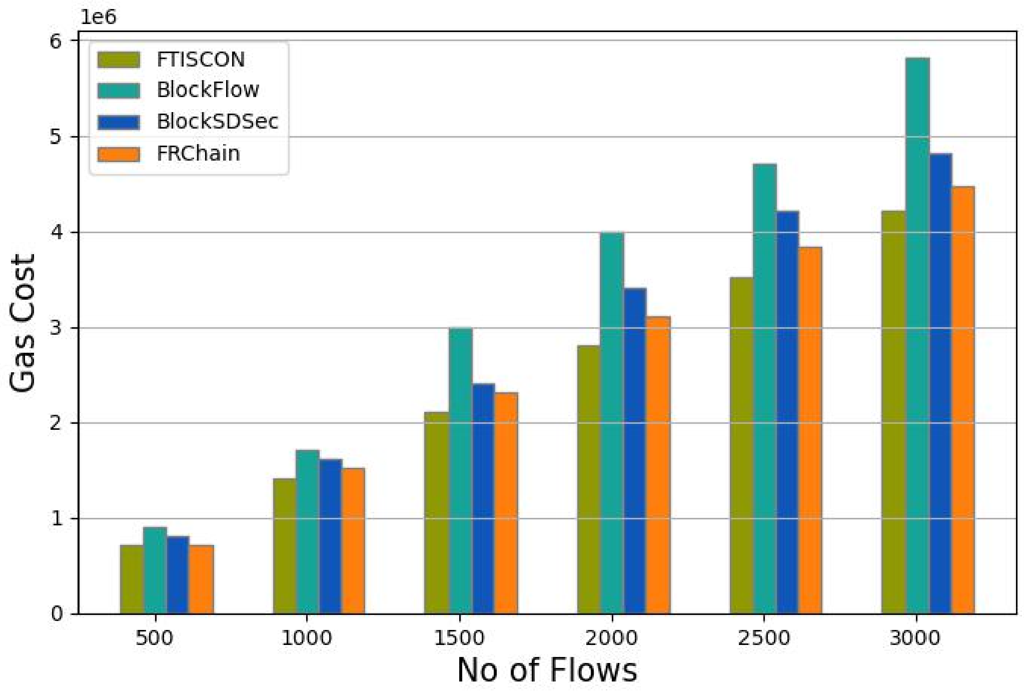


Figure 3.9: Gas Consumption comparison with respect to the number of Flows.

transaction cost for *addRule()* will vary as per the number of flow rules. However, only a mere reading of Blockchain data will not cost gas. Therefore, the methods *verifyFlow()* will not cost any fee. We plot the result of transaction cost with an increasing number of flows in Figure 3.9. From the figure, we observe that BlockFlow and BlockSDSec consume more gas compared to the proposed model. As we add more flows, the transaction cost rises proportionally. Nonetheless, the FRChain manages to consume the same amount of gas in fewer flows but it also increases with an increasing number of flows.

3.3.3.5 CPU Overhead on SDN Controller

The attacker often sends a large number of flow modification packets to overwhelm the SDN controller thereby causing it to crash or become unresponsive. The CPU overload may arise on the SDN controller due to a large number of flows entering the network in a short period. Then the SDN controller will experience a high CPU overhead due to the need to process a large number of flow updates.

Therefore, we perform an analysis of the CPU consumption of an SDN controller to identify if the controller is under heavy load and potentially being targeted by a flow modifica-

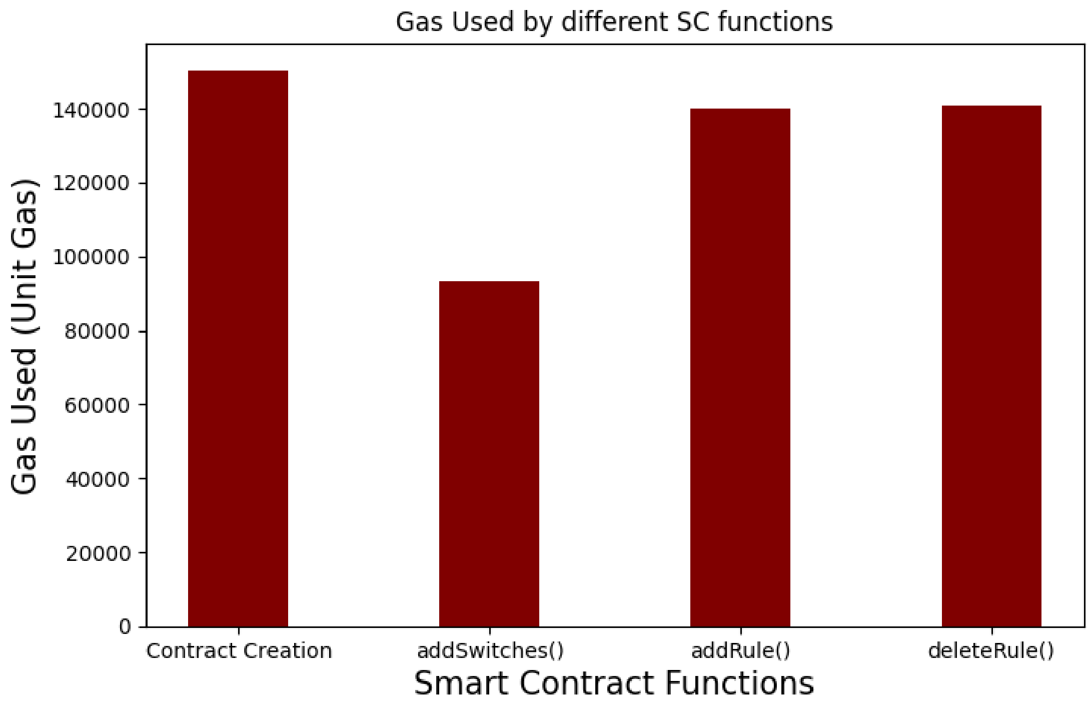


Figure 3.10: Gas Consumption comparison for different SC functions.

tion attack. We perform a Flow_Mod attack on the OpenFlow switch by injecting a synthetic flow modification attack at a different rate. We start at 100 Flow_Mod/second and the CPU consumption remains stable, as the Controller can handle the traffic using its available processing resources (refer Figure 3.11). However, after a certain point (around 500-600 packets/second), the CPU consumption starts to increase more rapidly with each incremental increase in packet_IN rate. This is because the controller is required to process more and more packets. Finally, at around 1500 packets/second, the CPU consumption reaches its saturation point where it cannot process any more packets. The graph for CPU consumption shows a curve that initially steps upward slowly, then steepens as the packet_IN rate increases.

3.3.3.6 Communication Overhead on SDN Controller

In this subsection, we analyze the communication overhead on the SDN controller due to the modification attack. Existing approaches rely on the switch version concept, which requires the reinstallation of the entire flow from the latest switch version.

Let's denote the initial number of flow rules as j and the initial version number as $v = j$. After a single flow modification attack, the version number is updated on the blockchain

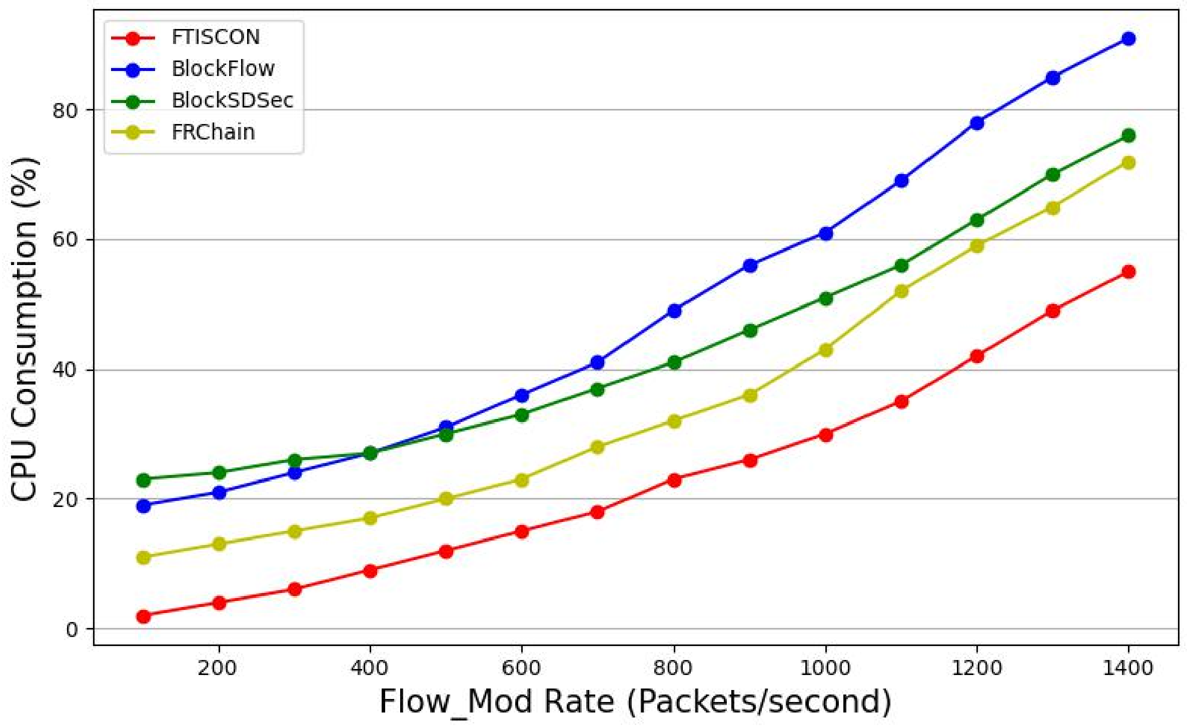


Figure 3.11: Overhead on Controller CPU with an increasing number of Flow_Mod packets.

as $v = j+1$. In the existing approach, the flow rules from previous versions need to be re-installed on the switch. This involves flushing out the existing j flow rules and reinstalling j flows. Therefore, the controller overhead is double the initial number of flow rules. If n flow modification attacks occur, the controller will have to handle the reinstallation of flow rules for each attack. Therefore, the controller overhead (O_n) after n flow modification attacks can be expressed as $(O_n) = 2.j.n$

In the proposed model, only the affected flow rules need to be reinstalled and there is no need to flush existing flows. Therefore, the controller overhead (O_n) after n flow modification attacks can be expressed as $(O_n) = n$. This shows a significant reduction in controller overhead compared to the existing approach.

3.3.4 Security Analysis

In this section, we briefly discuss the security analysis based on authenticity, integrity and resiliency against single-point-of-failure.

a) **Authenticity:** The clients need to authenticate with their public-private key pairs to communicate with the SC and participate in the verification/consensus process. Thus any unauthorised client cannot access the SC functions to tamper with the data.

b) **Integrity:** The flow information is stored on the blockchain using some hashing algorithm. Once a particular block is altered the hash of the block changes. This makes the remaining block invalid. Therefore, blockchain can preserve the integrity of the flow rules.

c) **Resiliency:** The decentralized nature of the peer-to-peer networks increases the system's robustness. As more and more nodes participate in the consensus mechanism, the likelihood of failure decreases. Therefore, the fate of the data is not controlled by a single person, group, or organization. The attack from a malicious node cannot disrupt the network entirely because the remaining nodes in the blockchain network will keep running the consensus process.

3.3.5 Complexity Analysis

In this subsection, we discuss the time and space complexity of the proposed model. The complexity of the FTISCON mainly depends on the number of flow rules present on the SDN switch because these flows are to be stored and verified. The time complexity for different operations is discussed below:

a) Flow Registration: During the registration of f_{new}, the SC first checks for flow conflict. So, it needs to iterate throughout all the existing flow rules (f_{old}). However, the proposed model utilizes the dictionary (key-value pair) structure to store the flow rules on the blockchain. The hashed value for the f_{new} is stored as the key during the flow storage on the blockchain. Therefore, the overall time for a f_{new} registration operation is $O(1)$ + $O(consensus)$. The time complexity of the consensus algorithm is difficult to express as it depends on several factors such as the computational power of the network, the difficulty level set for the mathematical puzzle, and the rate of block creation.

b) Flow Verification: For the verification of f_{new}, the SC generate the $hash(f_{new})$ before iterating on the dictionary for similarity check. Then the hashed value is then compared with the hashed of f_{old} dictionary. Therefore, the time complexity for verification of f_{new} is $O(1)$.

Table 3.4: Time complexity comparison table

Solution	**Registration**	**Verification**
BlockSDSec [16]	$O(f_{old})$	$O(f_{old}) + O(consensus)$
BlockFlow [53]	$O(f_{old})$	$O(2f_{old}) + O(consensus)$
FRChain [101]	$O(f_{old})$	$O(f_{old}) + O(consensus)$
FTISCON	$O(1)$	$O(1) + O(consensus)$

We evaluate the ability to detect malicious flow manipulation for all three attacks on a Mininet emulator. We also compared the time required for the registration and verification process of a new flow rule in BlockSDSec, BlockFlow, FRChain, and our proposed model (refer Table 3.4). The comparison table shows that the existing methods iterate through all the previous transactions before registering the flow on the blockchain. Similarly, during the verification, the existing methods perform the matching operation with the existing flow rules. Therefore, the proposed model is better in terms of time complexity when compared to existing methods for similar tasks.

3.3.6 Discussion

In this subsection, we discuss the security and effectiveness of FTISCON. The related study work shows that the SDN network cannot guarantee the complete integrity and privacy of OpenFlow rules. Some of the works in the literature use Blockchain Technology to prevent the flow rule modification attack. The adversary may install false flow rules or modify the existing flow rules present in the switches. In this work, we have proposed a model to prevent the flow rule modification attack using Blockchain Technology in which flow rules are stored on the Blockchain and verified by the Blockchain Agent.

The Blockchain Agent acts as a trusted node that communicates with the Blockchain nodes and collects flow rules from the switches. In the design of our model, we keep the SDN controller and Blockchain Agent separate from each other. This is to protect the verification process from eventual attacks on the SDN controller. The FTISCON will be able to survive even if the controller goes offline. Furthermore, the communication overhead on the controller will be less as the verification process is handled by the Blockchain Agent. This is one of the advantages of our model. However, if the Blockchain Agent is compromised, the controller will not be able to do the verification.

Blockchain Technology offers immutability, transparency, and integrity of the data through SC automation. The changes made to the EVM state will be recorded in the transaction history. Therefore, it makes it easier for the system to track down the adversary and isolate the compromised part of the network. However, the difficulty of making changes in the functionality of SCs are issue [71]. If any error or loophole occurs in the run-time, it is almost impossible or expensive to correct.

The performance analysis result shows that FTISCON is significantly better in terms of execution time and transaction cost. This is due to the separation of Controller and Agent. The experimental result shows that FTISCON possesses a reduction in execution time as one of our primary objectives. However, the SDN controller has to look if any duplicate flows are being stored on the blockchain and this is done to reduce the unnecessary filling up of Blockchain storage as well as to reduce transaction cost. Thus making it economically acceptable in the real scenario. On the other hand, the BlockFlow [53] needs to invoke *deleteRule()* function multiple times. This causes the model to pay more gas fees to make these transactions.

The proposed model is deployed and tested in a Private local Blockchain network to make the transaction processing faster. However, there are various other methods of deployment and testing available such as Testnet, and Mainnet.

The current Ethereum 1.0 platform uses heavyweight Ethash PoW as a consensus algorithm. Since PoW is based on the ability to compute hashes per second, it becomes slow with the increasing complexity level. Another vulnerability of PoW is the 51% attack where the attacker gains control over more than half of the total miners. By adopting PoW in our work we sought to ensure a more direct and relevant comparison with existing benchmarks and established systems. This approach was selected to ensure that our findings and conclusions are applicable to real-world scenarios and in line with the prevailing industry standards. However, we also acknowledge that there are alternative consensus mechanisms, such as Proof of Stake (PoS), Delegated Proof of Stake (DPoS), and Practical Byzantine Fault Tolerance (PBFT), which offer distinct advantages in terms of energy efficiency, security, and scalability. The development of Ethereum 2.0 will make it faster as it will implement the Proof-of-Stake to validate the transactions where it replaces the miners with validators. Ethereum 2.0 distributes the transaction data within the Ethereum network which increases the throughput of the network substantially.

The SDN and blockchain operate fundamentally on two different paradigms. Therefore, the integration of both of these technologies presents a set of technical challenges. However, the rise of Web3 facilitates a smooth ecosystem to build a decentralized application through Smart Contracts. It provides transparency to view and verify the transaction on the blockchain. Since the flow verification is automated through the SC, the proposed model also inherits the standing challenges of SC. Additionally, protecting the system from various attacks, both on the blockchain and SDN infrastructure, is a paramount challenge. Since the proposed model is tested on a private blockchain network, the number of consensus nodes is small. However, as the number of consensus nodes increases, the scalability issue of the system will grow.

3.4 Conclusion

In this chapter, we presented FTISCON, a scheme to protect flow rule modification attacks on OpenFlow switches in SDN using blockchain and incorporating an SC implementation for flow rule verification. A proof-of-concept implementation of the scheme is carried out using Ethereum private blockchain. Informal analysis based on the attack detection and experimental studies on running time and transaction cost is presented that validate the potential of FTISCON. The Performance analysis shows that the proposed model can significantly reduce the execution time, transaction cost, and delay incurred due to the flow verification compared to BlockFlow, BlockSDSec, and FRChain.