CHAPTER 4

Development of a Blockchain-enabled Multi-controller Architecture in SDN Network

4.1 Introduction

Due to the growing use case of IoT applications in the field such as Healthcare, Industry, and Autonomous Vehicles, the number of active IoT devices is expected to reach 13.1 billion by the end of 2022. However, due to the lack of unified IoT regulation, the current IoT ecosystem is vulnerable to various attacks on data privacy, DDoS attacks etc. This requires continuous monitoring and security analysis of end-to-end communication of IoT devices. Integrating Software-Defined Networking (SDN) with the Internet of Things (IoT) simplifies the management of IoT devices; however, it introduces security challenges. Adversaries may manipulate forwarding rules to redirect communication, compromising user security. The main problem arises from the single point of failure in SDN due to its centralized control architecture. When the master controller becomes unresponsive or fails, the entire network can become unstable or even unusable if no immediate recovery mechanism is in place. This brings the requirement of a Multicontroller environment to make the network agile and scalable. The SDN architecture also provides a mechanism to handle large networks by introducing multiple SDN controllers in the network. This work suggests a way to make SDN controllers more reliable by using multiple controllers, which helps ensure that services remain available even if one controller fails. Once the Master controller collapses, the remaining Equal controllers vote for the surety of controller failure then the most suitable equal controller is assigned as the master controller. The voting consensus is done through the Blockchain SC. Our proposed model considers response time and resource utilization of equal controllers, ensuring the most suitable controller assumes the role of master controller. Additionally, this paper proposes a mechanism to secure the forwarding path using Blockchain Technology. We introduced a Smart Agent that periodically performs the verification task by collecting the forwarding rules from the SDN switches. In the next

subsection, we elaborate on the working model of the proposed model.

The rest of the chapter is organized as follows: In Section 4.2, we describe the proposed model in detail. In Section 4.3, we discuss the performance evaluation using various parameters. In section 4.4, we provide a discussion on the proposed model and results obtained through the experiment. Finally, we present the concluding remarks in Section 4.5.

4.2 Proposed Model

In our solution, we implement a distributed SDN architecture called SDBlock-IoT, to safeguard the integrity of flow rules and provide more stability for the network services. The distributed but logically connected equal controllers maintain awareness of the master controller's status by regularly sending PING requests. If the master controller fails to respond, the correctness of the controller failure is confirmed through a voting consensus on the SC, in which all equal controllers participate. Our approach takes into account the response time and resource utilization of equal controllers to vote for the new master controller. Subsequently, we assess the scores of each equal controller and designate the one with the highest score as the new master controller.

The second aspect of our work involves validating OpenFlow rules through the SC. Flow rules are hashed and recorded on the blockchain. Flow verification occurs during new flow requests and periodically by the Smart Agent. In a new flow verification request, the agent computes the hash code of the flow rules and calls the function in the SC. The SC conducts consensus among blockchain nodes to verify if the hashed code is registered on the SC. If the hashed code is already registered, access is granted for the flow request; otherwise, the agent removes the flow from the forwarding devices. Further details on the operational model of our proposed work are provided in the following section.

4.2.1 Architecture

This subsection presents our proposed solution called SDBlock-IoT to safeguard the integrity of OpenFlow rules installed on the forwarding devices in an SDN-enabled IoT network. As shown in Figure 4.1, we have organized the SDBlock-IoT into three distinct layers: the application, control, and forwarding layers.

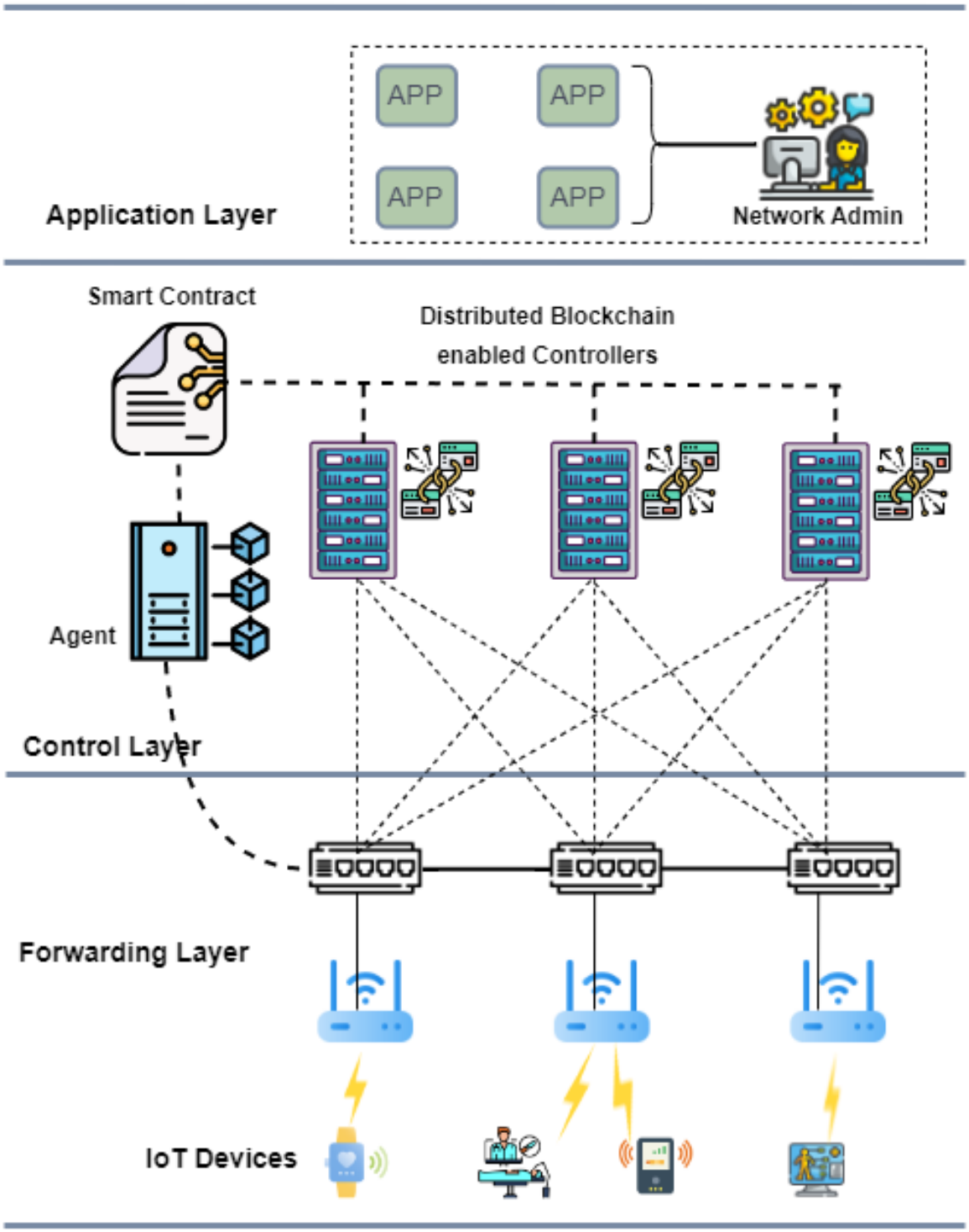

Figure 4.1: Proposed distributed SDBlock-IoT Architecture.

At the forefront, the application layer takes charge of overseeing and managing the diverse network applications in operation. Here, the network administrator's role becomes pivotal, involving tasks such as configuring network policies to ensure optimal functionality and overseeing the entire life cycle of deployed applications. This includes but is not limited to tasks

such as ensuring proper configuration settings, troubleshooting issues that may arise during application execution, and implementing any necessary updates or modifications to meet evolving requirements.

The control layer operates in tandem with the application layer, acting as the central hub for network management. It includes distributed but logically centralized SDN controllers to provide more efficient availability of network services. These controllers are equipped with a specialized capability to interact with the blockchain network during the consensus process of flow verification.

The interaction between the blockchain network and SDN involves a unique interface known as Web3, which is managed by the Smart Agent operating independently of the distributed controllers. When there's a requirement to set up a new flow on the SDN switches, the dedicated Smart Agent gathers the OpenFlow rules and transmits them to the Smart Contract through a transaction. After successful verification of the flow, the SDN controller directs the implementation of flow rules on the forwarding devices.

In the proposed work, the Smart Contract (SC) stands as a crucial piece of code imbued with business logic. Its primary role is to autonomously facilitate the consensus process among these controllers in instances where the master controller falters in managing the network. Beyond this, the SC undertakes the responsibility of scrutinizing forwarding rules, leveraging the assistance of a Smart Agent to ensure the integrity and validity of the network's operations. This ensures a seamless transition and reliable network performance even in the face of potential master controller failures.

The forwarding layer plays a crucial role in transmitting data packets according to the configured policies. In this layer, SDN-enabled switches and IoT gateways are situated to handle the forwarding of IoT traffic within the network. These switches are under the management and control of distributed SDN controllers, which install OpenFlow rules on them. Consequently, the traffic is directed by the OpenFlow rules defined on the switches. In the next subsection, we formally define the system model of the proposed work.

Table 4.1: Symbol Table

Symbol	**Description**
C_m	Master Controller
$EQL[IP_e, IP_e, ..., IP_e]$	List of Equal Controllers
IP_c	IP address of Equal Controller
IP_m	IP address of Master Controller
V_c	Vote count
H_c	Hashed code of flow rule
$dpid$	Datapath ID
$hash\_mapp[]$	Mapping between H_c and $dpid$
$Rules[]$	List of OpenFlow rules
$VCL[]$	Vote count list

4.2.2 Formulation of proposed SDBlock-IoT

Let $C = \{c_1, c_2, ..., c_n\}$ be the set of controllers equipped with blockchain public-private key pairs (Pub_{c_i}, Pri_{c_i}) to make the transaction. Let $S = \{s_1, s_2, ..., s_m\}$ be the set of SDN switches present in the forwarding layer of the IoT network. Let, M_{new} be the initial master controller that takes control of all switches S. Then, all other controllers (excluding the M_{new}) are in the equal controller. This can be mathematically expressed as-

$$\exists M_{new} \in C \text{ such that } \forall c_i \in C - \{M_{new}\}$$

Every packet/event that is received on the master controller is replicated to all other Equal controllers. These Equal controllers parallelly log the topology information on their memory. However, they are prevented from making transactions such as adding switches and hosts on the blockchain. Because these transactions will simply cost them an unnecessary gas fee.

Upon receiving a new connectivity request, the SDN controller configures the switch by installing flow rules $R = \{r_1, r_2, ..., r_k\}$, which are essential for processing packets. The flow

rules, identified by their unique attributes such as dpid, in_port, src, dst, priority, out_port, and protocol, undergo hashing using the *sha256* algorithm. Simultaneously, these hashed flow rules are synchronized and recorded on the blockchain as hashed_code $H = \{h_1, h_2, ..., h_k\}$. This ensures the integrity and traceability of the network rules through the secure and decentralized nature of blockchain technology.

In a real-world situation, an attacker can carry out three distinct types of attacks on a network: malicious addition, deletion, and modification of flow rules. Consequently, following such an attack, the state of the flow table on the switch could be altered to any one of the following conditions:

a) Unauthorized Flow Addition: The attacker may illegitimately introduce new flow rules into the switch's flow table, potentially leading to the redirection or interception of network traffic.

$R' = \{r_1, r_2, ..., r_{k+1}, r_{k+2}, ...r_{k+j}\}$

($j >= 1$, Maliciously adding j flow rules)

b) Malicious Flow Deletion: Another form of attack involves the malicious removal of existing flow rules from the switch's table. This act could disrupt normal network operations and cause service outages.

$R' = \{r_1, r_2, ..., r_{k-j}\}$

($j >= 1$, Maliciously deleting j flow rules)

c) Malicious Flow Modification: The attacker may tamper with existing flow rules, modifying their parameters or destinations. This type of attack aims to manipulate the flow of network traffic for various purposes.

$R' = \{r_1, r_2, ..., r'_{k-j}, r'_{k-j+1}, ...r'_k\}$

($j >= 1$, Maliciously modifying j flow rules)

4.2.3 Consensus Process for Master Controller Update

The process for updating the new master controller involves two steps: Detection of master controller failure and controller consensus for the selection of a new master controller. We explain these two steps in the following subsection.

4.2.3.1 Detection of Failure

Detecting a failure in the master controller is crucial for a swift transition to backup controllers in SDN-enabled networks. The master controller plays a vital role in sustaining network connectivity, and any disruption needs to be swiftly addressed to minimize downtime. To address this, the proposed work performs consensus among controllers to decide whether the Master controller is truly unavailable or not and assigns a new Master among the active controllers. Therefore, the equal controllers need to check the aliveness of the Master controller constantly. There are mainly two situations that can arise on the master controller. The master controller is completely down or it has changed its role to slave. Therefore, a new Master controller needs to be placed from the active equal controllers to manage the network.

Initially, the network admin generates a list of public-private key pairs $E_1(pub, pri)$, $E_2(pub, pri)$,..., $E_n(pub, pri)$ for all the controllers for making the transaction on the blockchain. This information is stored on the blockchain through the SC function *RegisterController(E_i(pub, pri))* for further communication on the blockchain network. Next, the equal controllers keep monitoring the aliveness status of the master controller by sending ping requests. The Master controller replies with an ICMP packet if the controller is alive. Otherwise, the requesting controller sends an update request/vote count (V_c) on the SC. Similarly, the other controllers also send the update request.

Each equal controller monitors the aliveness status of the master controller. The voting of the update request is done based on the following two conditions:

a) If $Alive(M_{old}) == True$, the master controller is alive and no action is taken.

b) If $Alive(M_{old}) == False$, the equal controller sends a vote for a master controller failure.

After the vote collection phase, a decision is made to find the status of the master controller. However, if a compromised equal controller lies and the vote count does not reach the total number of equal controllers, it could potentially disrupt the election process for determining the new master controller. In such a scenario, the consensus system may not be able to reach an agreement on the new master controller. Therefore, we incorporated threshold (Θ) within the smart contract. By setting thresholds for the number of valid votes required to make a decision, we ensure that the consensus process remains robust even in the presence of malicious or compromised controllers. If the false vote crosses the threshold, the consensus process

is halted, and appropriate actions, such as retrying the election process or invoking a fallback mechanism are taken.

We set up the threshold value as follows:

$$\Theta = \frac{1}{3} \times N_e \tag{4.1}$$

where Θ is the threshold and N_e is the total number of equal controllers. By allowing for the possibility of one-third of the equal controllers failing, the network can uphold its operational reliability, even if several failures occur simultaneously. When only a fraction of controllers report the master controller as failed, it might suggest isolated problems or deliberate tampering. Therefore, taking action only when this threshold is met helps to minimize the chances of unnecessary alerts due to false alarms, while still promptly addressing genuine issues.

Let V_c be the vote count received from the equal controllers regarding the master controller's status. Let $VCL[]$ be the list of votes for the corresponding equal controllers. This list is maintained by the smart contract to keep track of the compromised equal controllers. So, the final vote count can be expressed as

$$V_c = \sum_{i=1}^{N_e} VCL[i] \tag{4.2}$$

We divide the decision based on four scenarios during the detection of master controller failure. These are discussed below-

a) *All Equal Controllers Report Master Controller as Alive:* In this scenario, all equal controllers have successfully detected the aliveness of the master controller. This indicates a normal operational state where the network functions as expected. No further action is needed as all controllers are in sync with the master. This condition can be expressed as: $(V_c == N_e)$

b) *All Equal Controllers Report Master Controller as Failed:* If all equal controllers unan-

imously report that the master controller is not alive, it signifies a failure in the master controller. In response, the network initiates a consensus process to select a new master controller. This ensures that the system remains functional even in the event of master controller failure. This condition can be expressed as: ($V_c == 0$)

c) *Some Equal Controllers Report Master Controller as Failed:* If the vote count indicates that less than or equal to one-third of the total equal controllers report the master controller as failed, it suggests a potential compromise or malfunction in some equal controllers. This could be due to connectivity problems or tampering attempts. To mitigate the risk of tampering, the system initiates a re-voting process and notifies the network administrator to redeploy a new set of equal controllers with fresh configurations and keys. This condition can be expressed as: ($V_c <= \Theta$)

d) *Majority of Equal Controllers Report False Information:* When the vote count exceeds one-third of the total equal controllers reporting the master controller as failed, it indicates a critical situation where a significant portion of the system may be compromised or malfunctioning. In such an extreme scenario, the system cannot rely on re-voting and must take immediate action. An event is emitted to notify the network administrator, prompting them to redeploy both the master controller and a new set of equal controllers with updated configurations and keys to restore system integrity. This condition can be expressed as: ($V_c > \Theta$)

In Algorithm 4, we outlined the failure detection procedure. If the master controller is identified as failed, we increment the vote count by one, indicating that the master controller is inactive. Once the vote is given by all the equal controllers, the smart contract executes the consensus function for selecting a new master controller. This crucial step allows the system to respond promptly to the detection of a failed master controller, ensuring the reliability of the overall system.

In the next subsection, we delve into the consensus process used to designate a new master controller. This decision is determined by assessing the response time and resource utilization of the equal controllers. This critical procedure ensures that the most suitable controller assumes the role of master based on its performance metrics. Additionally, we explore the factors influencing this selection and how they contribute to the overall efficiency and stability of the system.

Algorithm 4 Master controller failure detection

Input: $\mathbf{m}$: Total number of controllers, $\mathbf{N_e}$: Total number of equal controllers, $\mathbf{V_c}$: Vote Count, **Alive**(**i**): Aliveness status of controller i, **VCL**[]: Vote count list

Output: **VCL**[] Vote Count List

Generate key pairs: $\mathbf{E_1}(\mathbf{pub},\mathbf{pri}),\mathbf{E_2}(\mathbf{pub},\mathbf{pri}),...,\mathbf{E_n}(\mathbf{pub},\mathbf{pri})$
Register the Controllers: **RegisterController**($\mathbf{E_i}(\mathbf{pub},\mathbf{pri})$)
Assign threshold: $\Theta \leftarrow (\frac{1}{3}$ x $\mathbf{N_e})$
Initialize $V_c \leftarrow 0$
for $i \in EQL.length$ **do**
    if **Alive**($\mathbf{m_{old}}$) == **True then**
        $VCL[i] \leftarrow 1$
        $V_c \leftarrow V_c + 1$
    else
        $VCL[i] \leftarrow 0$
    end if
end for
// Vote collection and Decision making
if $\mathbf{V_c}$ == $\mathbf{N_e}$ **then**
    No Action: Controller is Alive
end if
if $\mathbf{V_c}$ == **0 then**
    ExecuteConsensus(**VCL**[])
end if
if $\mathbf{V_c}$ <= Θ **then**
    Initiate **Re** − **voting**
    emit **NotifyAdmin**()
else
    Initiate **Re** − **voting**
    emit **NotifyAdmin**() to redeploy equal controller
    return VCL[]
end if

4.2.3.2 Consensus for Update

To mitigate the issue of single-point-of-failure, our proposed solution deploys multiple equal controllers that actively monitor the status of the master controller. Every equal controller keeps track of both response time and resource utilization. Subsequently, a comprehensive assessment of these two factors is conducted to determine the ultimate score, which plays a crucial role in selecting the next master controller. This evaluation process ensures that the controller with optimal performance in terms of response time and resource utilization emerges as the preferred choice for the role of the master controller.

Response time reflects the latency between controller and network devices. A controller with lower response time is closer (network-wise) to the devices and can react faster to events, thereby ensuring efficient management of the network. Resource utilization indicates the current load on the controller in terms of CPU, memory, or bandwidth usage. Selecting a controller with lower utilization ensures that the new master is not already overburdened and has sufficient resources to manage the additional responsibilities. These two parameters ensure that the newly selected master controller is not only reachable quickly but also capable of handling control tasks without degrading performance or risking overload. This approach enhances the overall resiliency and reliability of the network.

Let us formalize the consensus method to decide on the master controller among the equal controllers considering these two criteria. Let $RespTime[i]$ denote the response time of Controller i when pinging the master controller, and ResUtil[i] represents the resource utilization (such as CPU and memory) of equal controller i. The calculated score for equal controller i is denoted as $Score[i]$. Now, we define specific scoring functions for each controller considering the scaling and weighting factors. The scoring functions take into consideration the response time ($RespTime[i]$) and resource utilization ($ResUtil[i]$) metrics to assess and rank the performance of each equal controller.

a) Scaling factors: Let $K1$ and $K2$ be the scaling factors for response time and resource utilization, respectively. The scaling factor plays a crucial role in determining how response time and resource utilization contribute to the overall score. Essentially, the scaling factor influences the significance of each parameter in the decision-making process. It's important to note that a higher value for the scaling factor implies a greater emphasis on response time when calculating the score. Therefore, we expect the value of $K1$ to be greater than that of $K2$,

highlighting the prioritization of response time in the evaluation process.

b) Weighting factors: Let α and β be the weighting factors for response time and resource utilization, respectively. The weighting factor determines the relative importance of response time compared to resource utilization and other factors. A higher weighting factor for response time indicates that it should have a stronger influence on the final score. Therefore, the value of α is expected to be greater than the value of β.

We use a linear scaling approach to determine the score for response time and resource utilization. The mathematical formulation is as follows:

$$ScoreRespTime[i] = K1/(RespTime[i] + \varepsilon) \tag{4.3}$$

$$ScoreResUtil[i] = K2/(ResUtil[i] + \varepsilon) \tag{4.4}$$

where $K1 > K2$ (Provides more priority to response time) and ε is a small constant to avoid division by zero.

Let us now combine the factors by weighting them appropriately to calculate the overall score for each controller.

$$Score[i] = \alpha * ScoreRespTime[i] + \beta * ScoreResUtil[i] \tag{4.5}$$

Finally, we can select the controller with the highest score as the new master controller.

$$MasterController = argmax(Score[i]) \tag{4.6}$$

In Algorithm 5, we elaborate on the consensus process for updating the master controller. Once all equal controllers detect the master controller as being down, the SC examines

response time and CPU utilization for the final score assessment. Following the reassignment of the new master, the equal controller is removed from the equal controller list to prevent ambiguity in subsequent consensus procedures. Consequently, the vote count is reset to zero. This ensures a streamlined and unambiguous transition in the event of a master controller failure.

Algorithm 5 Consensus for Master controller update

Input: $\mathbf{IP_e}$: IP address of equal controller, **EQL**[] : List of equal controllers, **RespTime**[] : List of response time, **ResUtil**[] : List of resource utilization, $\mathbf{K1}, \mathbf{K2}$: Scaling factor constants, α, β : Weighting factor constant, $\mathbf{V_c}$: Vote Count of IP_e

Output: currentMaster : Current Master

Initialize $score[]$ to keep score for each controller.
if $V_c == EQL.length$ **then** // master controller is down
    for $i \in EQL[]$ **do**
        $ScoreRespTime[i] \leftarrow K1/(RespTime[i] + \varepsilon)$
        $ScoreResUtil[i] \leftarrow K2/(ResUtil[i] + \varepsilon)$
        $Score[i] \leftarrow \alpha * ScoreRespTime[i] + \beta * ScoreResUtil[i]$
    end for
    $index \leftarrow max(score[])$
    $currentMaster \leftarrow EQL[index]$
    Reset the vote count $V_c \leftarrow 0$
    $EQL[IP_e].pop()$ // Remove the IP_e from $EQL[]$
    return $currentMaster$
end if

4.2.3.3 Smart Contract Design

At the initial deployment of the SDN controller, the SC registers the IP address of the current Master controller (IP_m) and the list of equal ($EQL[IP_e, IP_e, ..., IP_e]$) controllers. The SC has defined functions related to controller consensus. These are- *addMaster(IP_m)*, *addEqual(IP_e)*, *updateMaster(IP_e)*, and *updateEqual(IP_e)*. Functions *addMaster(IP_m)* and *addEqual(IP_e)* are invoked by the SDN controller to store the controller IP address on the blockchain. When Master update request *updateMaster(IP_e)* is received, the SC counts the number of votes received

so far from the controllers. Once the vote count reaches the total number of equal controllers, the SC decides to assign one of the active controllers as a new Master. Next, *updateEqual(IP_e)* function is executed to remove the current master IP address from the equal controller list which is no longer an equal controller.

In Figure 4.2, we depict the process of controller consensus and the communication sequence involved in selecting a new master controller. To illustrate this, let's consider the scenario with one network administrator, a smart contract, one master controller, and two equal controllers. The sequence diagram showcases the exchange of messages among these entities during the master controller selection process.

In the initial phase, the network administrator deploys the Smart Contract (SC) on the blockchain network. Subsequently, the SC address is integrated into both the master and equal controllers. The SC facilitates the registration of the IP addresses of the master (IP_m) and equal controllers (IP_{e1}, IP_{e2}) using the $addMaster()$ and $addEqual()$ functions. During step 3, the SC employs the *onlyAdmin* modifier, as discussed in subsection 4.2.6, to validate the registered IP addresses and enlists them in the admin list.

Now, let's consider a scenario in step 4 where the master controller experiences a failure, rendering it unresponsive to ping requests from IP_{e1} and IP_{e2}. In response, step 5 involves the SC checking the *voteCount* for each update request, such as $updateMaster(IP_{e1})$ and $updateMaster(IP_{e2})$. Subsequently, in step 6, the SC initiates the consensus process for updating the master controller, a topic explored in detail in subsection 4.2.3.2. Following the successful update of the master controller, the *voteCount* is reset, and the equal controller list undergoes necessary updates.

In the next subsection, we delve into the second aspect of the proposed model, focusing on the protection of forwarding rules within an SDN-enabled IoT network. In this part of the discussion, we will not only explore the safeguarding measures but also delve into pertinent details surrounding the management and security of forwarding rules in the context of SDN-enabled IoT networks.

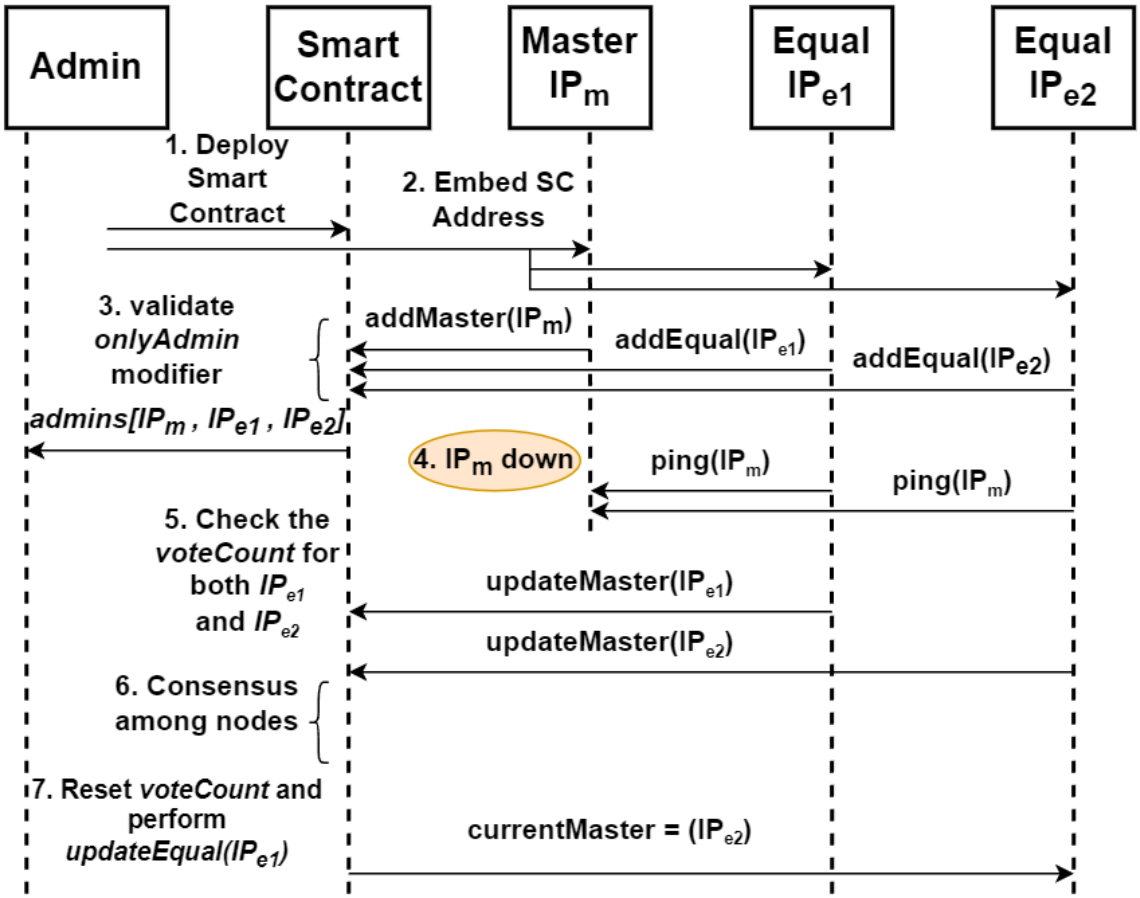

Figure 4.2: Controller Consensus Sequence Diagram.

4.2.4 Forwarding Rule Management in the Blockchain

To perform the verification of OpenFlow rules, we register the flow rules on the blockchain through the SC. In this work, we use the on-chain method to register the flow rules. The forwarding devices cannot directly communicate with the SC. Therefore, the SDN controller acts as an intermediary between them. The controller uses *Web3.py* library to communicate with the SC. We discuss the SC design for flow rule management below.

We define a structure on the SC to register the flow rules as shown in Algorithm 6. The structure contains *dpid* of the switch and *hashed_code* (H_c). The H_c is assigned as the key to reducing the time required for flow rule verification. Because the iterative method and linear search will gradually consume more time as the number of flow rules increases. The algorithm checks for duplicate flows on the blockchain before registering the flows by checking the hashed value of the new flow. If it matches then the transaction is reverted to its previous

state. This function is invoked multiple times as the Master changes. Therefore, H_c will help to find the duplicate flow rules on the blockchain. This will also minimize the gas cost required for the transaction. If there is no duplicate flow entry, the SC will register the flow rule on the blockchain.

Algorithm 6 Flow rule management in Smart Contract

Input: $\mathbf{H_c}$: Hashed flow rule, $\mathbf{dpid_i}$: Datapath ID of the switch i

Output: **txn** : Transaction hash from the Blockchain

1: **if** $hash\_mapp[H_c] == dpid_i$ **then**
2: Revert transaction $txn \leftarrow revert()$
3: Return txn
4: **else**
5: $txn = Rules(H_c, dpid_i)$ // *Register the rule*
6: $hash\_mapp[H_c] \leftarrow dpid_i$
7: return txn
8: **end if**

4.2.5 Flow Verification by Smart Agent

Flow rules on the forwarding devices need to be verified to see if the flow has been modified by any external adversaries. In this letter, we utilize the blockchain SC to verify the flow rules to avoid false flow rule injection and modification attacks.

A Smart Agent is brought in to handle the verification task separately from the SDN controller, without using complex technical language. The flow verification is performed twice, one at the time of the first flow request and another by the Agent. The Agent is fed with the information regarding the Contract address and ABI (Application Binary Interface) to execute the functions on the SC. Then flow rules are collected periodically from all the switches on the network. The Agent computes the hash code of the flow rules and invokes the *verifyFlow(H_c)* function. The Blockchain nodes perform the consensus by taking the hashed code of the flow rules with the code present in the SC. If the H_c matches the hashed code present on the blockchain then the connectivity request will be granted access. Otherwise, the Agent will remove that particular false or modify that flow from the switch. The process of flow rule verification is

presented in Algorithm 7.

Algorithm 7 Flow rule verification by Smart Agent

Input: r : Flow rule to be verified

Output: status

1: Compute $H_c \leftarrow sha256(abiencode(r))$
2: **if** $hash\_mapp[H_c]$ **then**
3: $status \leftarrow$ True
4: **else**
5: $status \leftarrow$ False
6: emit event $Invalid()$
7: **end if**
8: return $status$

4.2.6 Use of Admin modifier in Smart Contract

The modifiers in Ethereum SC check the conditions before executing the function. If the function does not satisfy the requirements, the function execution stops. The modifier serves as a validation check for SC functions. Therefore, we employ this concept in our SC to restrict the execution of certain functions from unauthorized wallet accounts. The account associated with the owner of the SC will be the first admin wallet address $admins[wa_1]$. When an SDN controller is deployed, their account is added to the $admins[]$ list. Therefore, an admin can only add another account to the $admins[]$ list. We defined a modifier *onlyAdmin* and placed it before all the functions in the SC. So, the SC checks if the wallet address is allowed to execute the function otherwise the transaction is reverted to its previous state. However, the caller will have to pay gas costs even if the execution is not complete. This way unauthorized controllers will not be able to take control of the network. We present the onlyAdmin modifier in Algorithm 8.

Algorithm 8 onlyAdmin modifier in Smart Contract

Input: msg.sender : Wallet address of sender, **admin**[] : List of Admins

Output: admin : Status of Admin

```
Set admin ← False
for i = 0; to i < admins.length do
    if msg.sender == admins[i] then
        admin ← True
        break
    end if
end for
if !admin then
    revert()
end if
```

4.3 Performance Evaluation and Results

4.3.1 Physical Environment Setup

In Figure 4.3, four virtual machines (VMs) are initialized and successfully communicate with each other. The SC is deployed on the Ethereum Private blockchain from VM4, and the contract address is shared with controller VM1 (Master), controller VM2 (Equal), and controller VM3 (Equal). To emulate the distributed multi-controller environment, we connect two Hardware switches (Allied Telesis OpenFlow switch) to all the SDN controllers. The Agent on VM4 continuously verifies flow rules through the SC. We set up the Mosquitto MQTT broker on the VM4 to receive the IoT traffic from the IoT sensors. Each IoT sensor publishes a message to the MQTT broker. For each IoT sensor device appropriate flow rules are installed on the switches by the SDN controller. We also used the DITG network traffic generator to test the performance with an increasing number of OpenFlow rules. The device specification for the network topology is presented in Table 4.2.

Table 4.2: The Device Specification Table

Devices	Specifications
Operating System	Ubuntu 20.04.1 LTS
RAM	4 G
Core	Intel Xeon(R) W-2145
CPU	3.70GHz x 4
OF version	OpenFlow 1.3
Hardware Switch	AT-X230-28GP PoE
MQTT Broker	Mosquitto 5.0
DITG	3.2

4.3.2 Experimental Results

In this subsection, we describe the numerical performance evaluation results in physical hardware devices. We compare the results with three other existing methods (BlockFlow [53], BlockSDSec [16] and FRChain [101]) that are close to our method.

4.3.2.1 Latency for New Controller Selection

We measure the time required for the selection of a new Master controller to take control of the network from the failure. Once the Master controller is unavailable, the remaining Equal controllers start voting on the SC for the new master. Since all events are replicated to all the controllers there is no need for a temporary buffer to store the events. Due to this, even when the Master controller crashes the events are delivered at least once.

We experimented by sending continuous traffic in the SDN-IoT network. In the middle, we shut down the Master VM and recorded the time for which there was no traffic on the network. The latency of the SDBlock-IoT mainly depends on detecting the Master failure, the execution time for the consensus algorithm for the new Master, and *OFPRoleRequest()* instruction to SDN switches. We performed ten iterations of this experiment to measure the latency for the new master controller selection (see Table 4.3). With frequent controller failures, the time for a new master controller selection increases exponentially. The average latency for

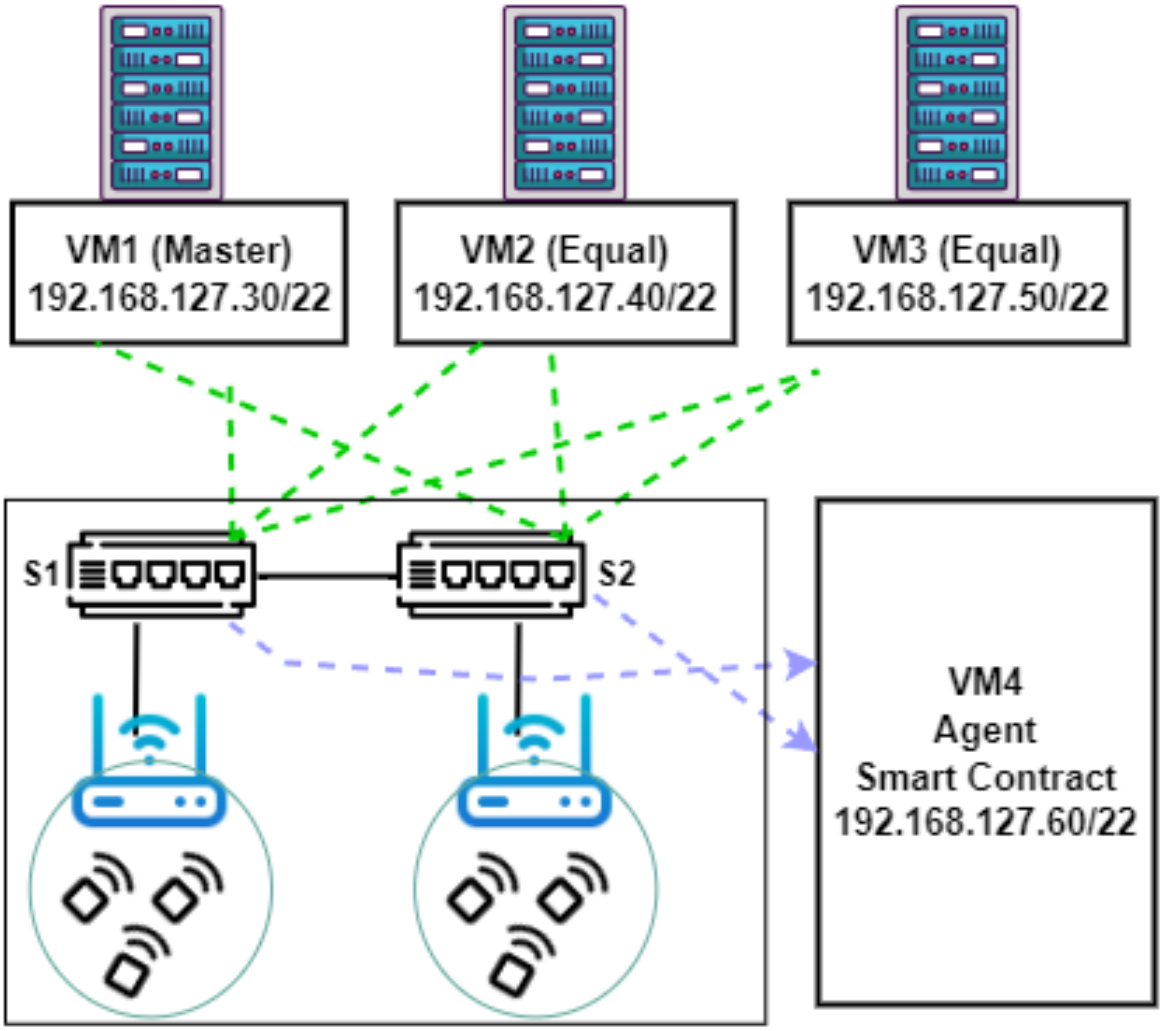

Figure 4.3: Network topology used in our experiment.

Master controller recovery is **52.87ms** with a standard deviation of **1.31ms**.

We also tested the performance of the proposed method to observe the time for controller recovery due to flow modification attacks. The comparison of the result is depicted in Figure 4.4. From the figure, we observe that the controller update time increases linearly with the increasing number of flow mod attacks. However, the proposed method significantly enhances the responsiveness of network service availability by providing a minimum update time compared to existing methods.

Table 4.3: Latency for new Master controller Selection

Iteration	Average time (ms)			
	Detection	**Selection**	**OFPRoleRequest**	**Total**
1	16.81	25.71	12.39	54.95
2	14.45	25.18	13.10	52.73
3	15.51	25.36	12.97	53.84
4	14.22	23.99	13.51	51.72
5	16.85	25.06	11.53	53.44
6	15.12	26.78	13.12	55.02
7	14.49	24.37	13.04	51.90
8	14.12	23.63	13.36	51.11
9	14.70	24.08	13.50	52.28
10	15.52	24.21	12.01	51.74

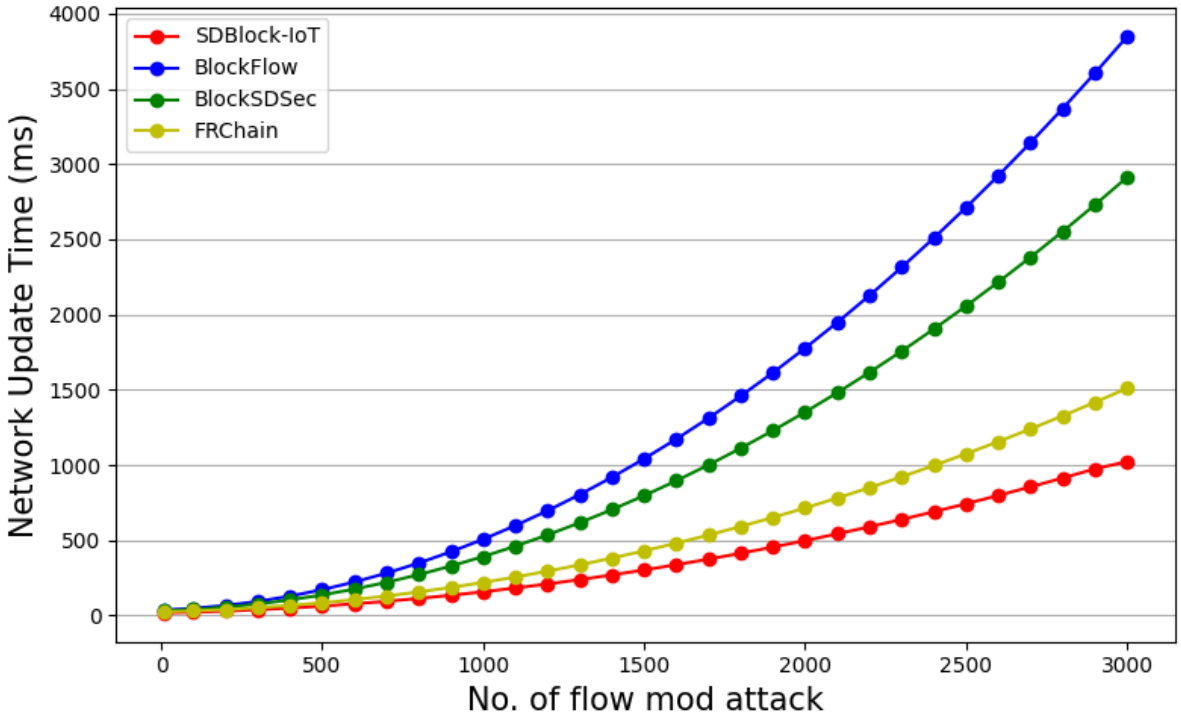

Figure 4.4: Latency for controller recovery due to flow mod attack.

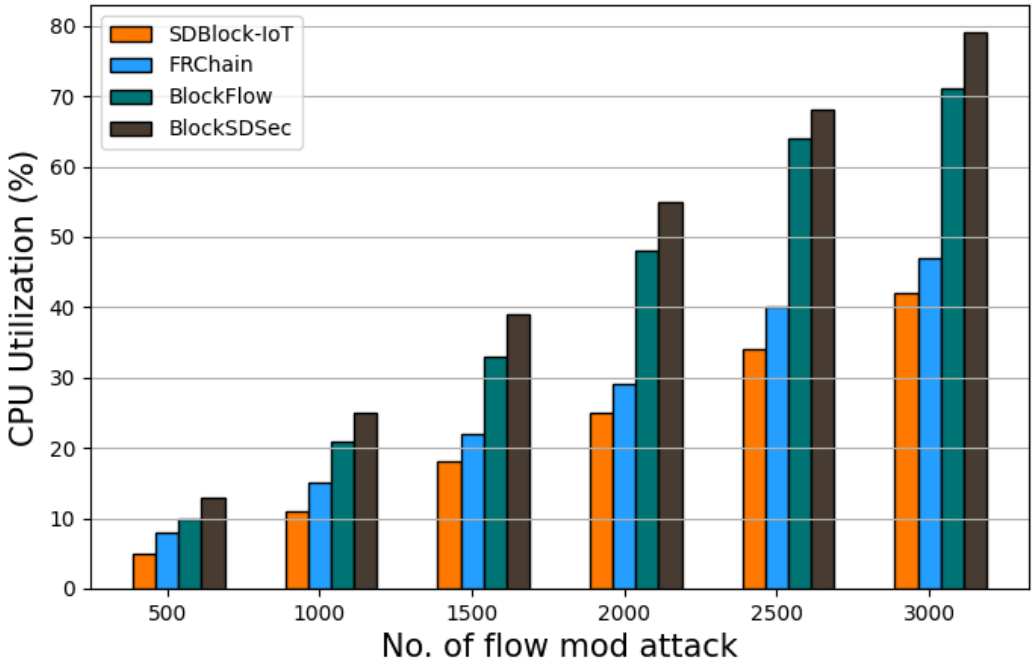

Figure 4.5: Controller overhead on different numbers of flow modification attacks.

4.3.2.2 CPU Utilization of Master Controller

The CPU Utilization of the master controller during the flow modification attack is depicted in Figure 4.5. A different number of flow modification attacks is performed for 120 seconds to observe the effect on the CPU of the controller. From the figure, we observe that with the increased attack rate, the CPU consumption also climbs up. However, the proposed method provides adequate protection against flow modification attacks.

4.3.2.3 Flow Verification Latency

Another experiment is conducted to measure the verification time for malicious flow modification attacks. We inject an attack script on the SDN switch from an external module to maliciously add, delete, and modify the flow rules. Initially, we install a fixed dummy flow i.e. $k = 3000$ on the switch.

The agent regularly gathers information about how data flows through the switches and executes the *verifyFlow()* function on the SC. For different values of j, we tested the verification time of the flow rules. We perform all three attacks (add, delete, and modify) on the forwarding

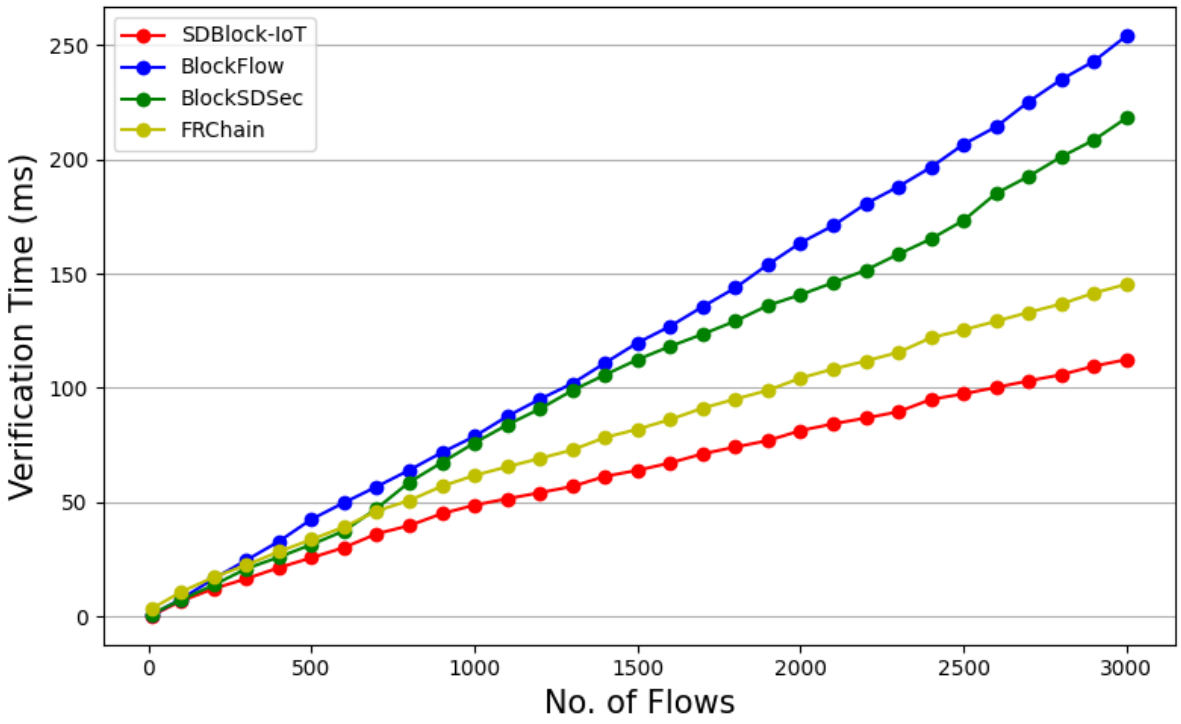

Figure 4.6: Flow verification latency with an increasing number of flow rules.

device. Figure 4.6 shows that the proposed method can precisely detect the modification attack with a relatively low average verification time.

4.3.2.4 Transaction Cost

One of the most important factors in the system while using blockchain is the transaction cost. The amount of gas required for the transaction is computed as the product of the gas price and the unit of gas used. We set the standard gas price of 21000*gwei* for our experiment.

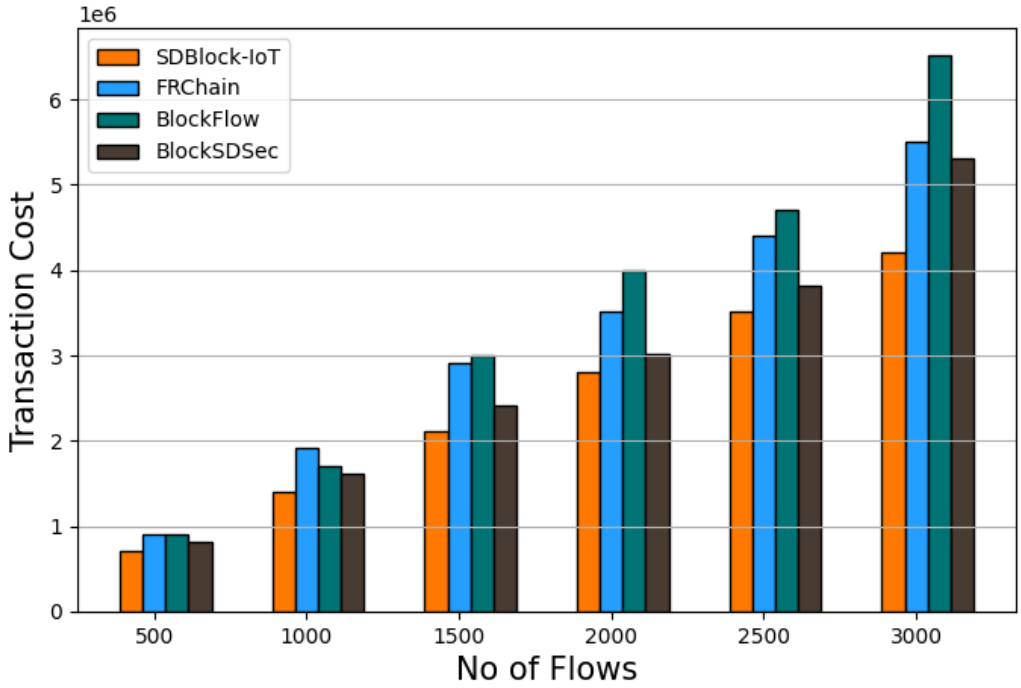

Figure 4.7: Transaction Cost for SC function execution.

Figure 4.7 shows the amount of total average transaction cost on different numbers of flow rules. The result shows that the proposed method requires less cost compared to the other three methods. The use of multiple controllers will not cost any extra transaction costs on the blockchain network. Because only the Master controller can modify the machine state.

These are the results obtained from the two Allied Telesis hardware switches. We have conducted further experiments to assess the performance of a large number of flow rules. To mimic real-world scenarios of large-scale networks, we set up a virtual network using Mininet, consisting of 20 Open vSwitch (OVS) switches. Through a custom Python script, we deployed 500 dummy flow rules on each switch within this virtual environment. This comprehensive test aimed to simulate the challenges posed by managing a vast number of flow rules across numerous switches.

We performed additional experiments on this virtual topology to measure the flow verification latency and latency for controller recovery.

Firstly, we measure the flow verification latency during the flow modification attack. In the Figure 4.8, it is evident that the proposed model exhibits a consistent linear increase as the number of flow modification attacks rises, and it demonstrates quicker processing times

compared to existing models. This indicates the robustness and efficiency of the proposed approach in handling varying attack scenarios. Furthermore, the results obtained from the hardware switch and the OVS switch implementations yield similar results, highlighting the consistency and reliability of our findings across different system configurations. This suggests that our proposed model is not only effective but also adaptable to large-scale network infrastructures.

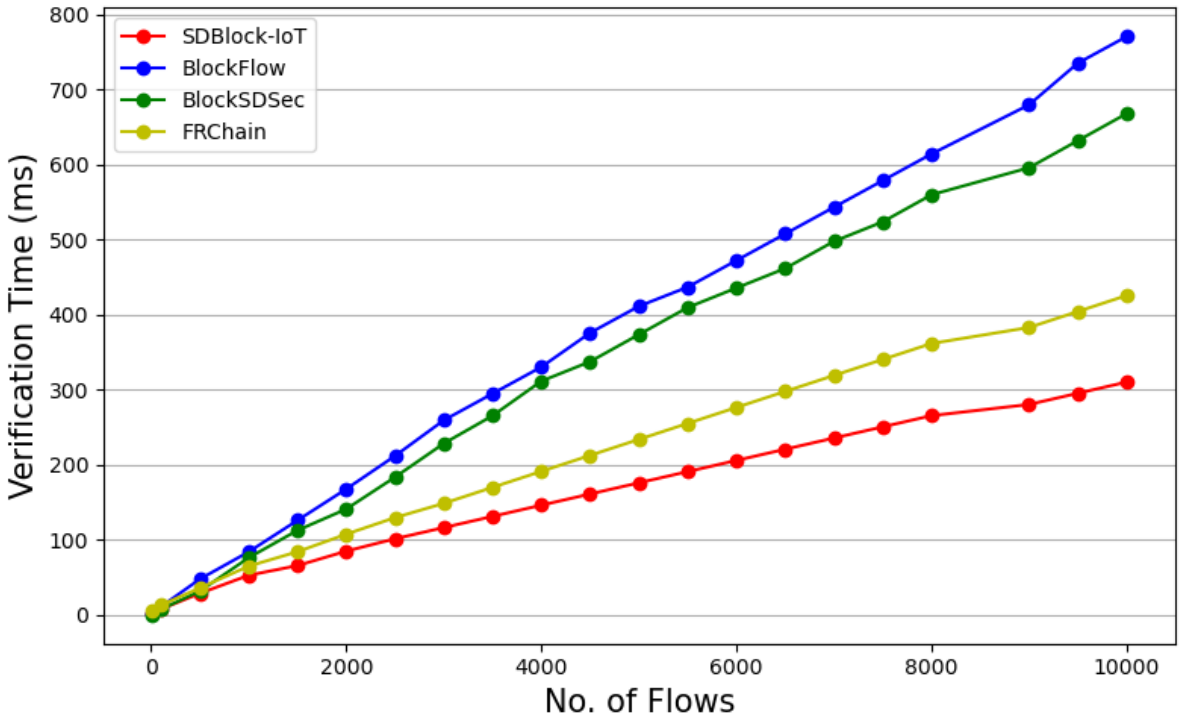

Figure 4.8: Flow verification latency with an increasing number of flow rules on simulated network.

Similarly, we have experimented to measure the latency for controller recovery following flow modification attacks within a simulated network environment. The findings revealed a consistent linear escalation in the network update time across the simulated network (Shown in Figure 4.9). Intriguingly, when compared to the simulated setup, the results obtained from the hardware switch exhibited a steeper upward trajectory.

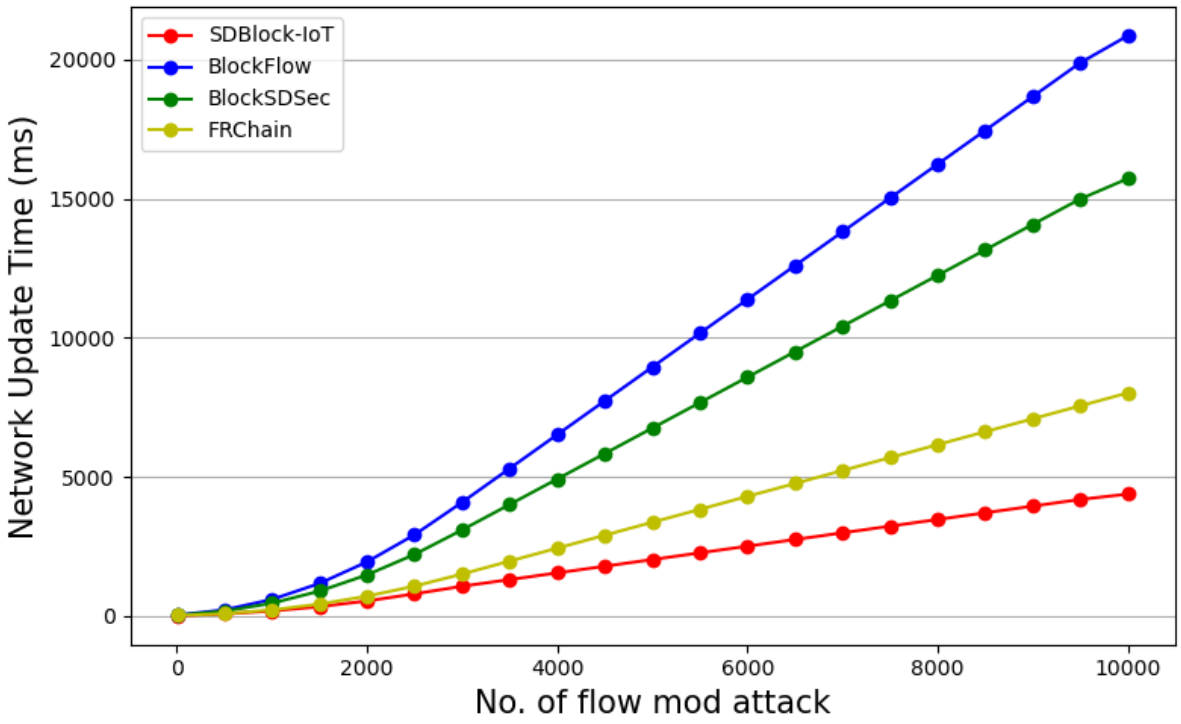

Figure 4.9: Latency for controller recovery due to flow mod attack on the simulated network.

4.4 Discussion

Our proposed model incorporates a distributed multi controller architecture, which enhances the resiliency of the system by distributing control responsibilities across multiple controllers. This distributed approach mitigates the risk of single points of failure and improves fault tolerance, ensuring robust operation even in the face of controller failures or attacks. We have devised a voting algorithm through the Smart Contract that can detect the master controller failure even in the presence of a compromised equal controller. To mitigate the impact of misbehaving controllers, we incorporated threshold-based consensus within the smart contract. Setting a threshold at 1/3 of the total equal controllers balances the need for prompt detection of abnormal behavior with the risk of false positives. A lower threshold might trigger alarms too frequently, leading to unnecessary interventions or disruptions in normal operation. Conversely, a higher threshold might delay detection of genuine issues, allowing them to escalate. Achieving absolute security against attacks or failures often comes with significant resource costs, both in terms of computational overhead and infrastructure redundancy. Therefore, we allow for a certain degree of fault tolerance to balance the need for resilience with resource constraints.

In our proposed model, all controllers are required to register themselves on the blockchain using their public-private key pairs for making transactions. This registration process ensures the authenticity and integrity of transactions initiated by controllers. Each controller's public key is associated with its identity on the blockchain, facilitating secure communication and transaction validation. By leveraging distributed decision-making mechanisms, we enhance the resilience and robustness of our scheme against potential attacks. In addition to that, the updated voting mechanism will give the probability of the controller being attacked. The fourth condition of the master controller failure detection algorithm takes action if the voting does not satisfy the given threshold. That is if more than $\frac{1}{3}$ of the equal controllers give a false vote there is a chance that some of the equal controllers are hacked.

On the other hand, the integrity of flow rules is preserved through the blockchain. The existing methods maintain a switch version for each state change, leading to increased overhead as the entire flow rule needs to be fetched from the blockchain upon detection of manipulation. In contrast, our proposed model strategically reinstalls only the recently changed flow rules, rather than the entire flow, from the blockchain. This selective reinstallation approach significantly reduces the time required for verification (Figure 4.6), as only the affected flow rules need to be retrieved and verified. For every transaction in the blockchain, there is computational work needs to be done by the blockchain nodes to verify the flow rules. The proposed model selectively re-installs the original flow rules and thus minimizes unnecessary transaction costs (Figure 4.7) and controller overhead (Figure 4.5) associated with fetching and reinstalling unchanged flow rules. This streamlined process optimizes the efficiency of flow verification, particularly in scenarios with frequent state changes or large-scale networks.

Additionally, the FRChain initiates a voting mechanism once an unmatched flow rule is detected. If there is an illegal transaction in the block then a negative vote is given by the controller. This voting result determines whether the flow table is safe or not. However, FRChain does not perform well when the number of nodes in the network increases as it takes longer to complete the voting process. Our approach directly compares the hash codes stored on the blockchain with those from the switches. Therefore, this significantly reduces the time required for decision-making, as retrieving hash codes from the blockchain dictionary incurs constant time overhead. These are the reason that makes the proposed model superior to the existing models.

4.5 Conclusion

We proposed SDBlock-IoT, a scheme to provide security to forwarding rules on OpenFlow switches for IoT devices. We integrate blockchain technology to verify the flow rules through SC. The proposed method achieves nominal verification time while preserving the integrity of flow rules. In addition, a multi-controller architecture is designed to eliminate the single point of failure where other Equal controllers take control of the network with minimal latency. We carried out an extensive experimental performance analysis based on network update time, latency for flow verification, CPU overhead on the controller due to attack, and finally, transaction cost. The real-world implementation of the proposed method shows superior performance in comparison to existing methods.