الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

An Intrusion Detection System in Sdn-enabled IoT Networks

Faculty of Informatics Engineering Department of
Systems Security and Computer Networking

Prepared by : Nagham Ashqar & Moustafa Seifo

Abstract:

This project investigates the security of MQTT-based IoT networks by designing and implementing an intrusion detection and mitigation framework based on Software-Defined Networking (SDN). The proposed approach leverages the centralized control and global visibility provided by SDN to dynamically detect malicious traffic patterns and enforce mitigation policies at the network level. Instead of relying solely on traditional host-based defenses, the system integrates a network-based intrusion detection system (IDS) to monitor MQTT traffic in real time.

In the proposed architecture, Suricata is employed as a signature-based IDS to analyze MQTT traffic and detect known attack patterns and protocol violations, including publish flooding and malformed MQTT control packets. Upon detecting a security alert, the SDN controller reacts by dynamically installing flow rules in the OpenFlow switch to block malicious traffic originating from the detected source for a predefined period of time. This tight integration between the IDS and the SDN controller enables automated and rapid mitigation without manual intervention.

The system is implemented and evaluated in a virtualized testbed using Mininet, Open vSwitch, the Ryu SDN controller, and a Mosquitto MQTT broker. Experimental results demonstrate that the proposed solution is capable of detecting and mitigating several MQTT-specific attacks in real time, effectively reducing their impact on the broker and the network. The project highlights the feasibility and effectiveness of combining SDN with signature-based intrusion detection to enhance the security of MQTT-based IoT environments, while also identifying potential challenges and directions for future improvements.

Contents

1. Introduction

The Internet of Things (IoT) has transformed modern computing by enabling seamless connectivity between everyday devices—from smart home appliances to industrial sensors—facilitating automation, remote monitoring, and data-driven services. However, this rapid expansion introduces profound security challenges, as the scale and heterogeneity of IoT ecosystems create numerous attack surfaces that adversaries can exploit. Many IoT devices are resource-constrained and lack robust built-in security, resulting in vulnerabilities that traditional network defenses struggle to address. This combination of high connectivity and weak security makes IoT environments susceptible to attacks such as DDoS, unauthorized access, and message manipulation, threatening critical infrastructures like smart utilities and healthcare systems [1].

Software-Defined Networking (SDN) has emerged as a promising architectural paradigm to enhance the security and manageability of complex network environments, including IoT. By decoupling the control plane from the data plane, SDN provides centralized network intelligence, global visibility, and programmable control over traffic flows. This flexibility allows security functions—such as intrusion detection and policy enforcement—to be abstracted from individual devices and managed centrally, enabling rapid responses to malicious activity. For IoT, SDN's centralized control supports scalable, fine-grained security management that is adaptable to diverse traffic patterns and lightweight protocols like MQTT, which is widely used in IoT communications yet vulnerable to both known and emerging threats [1].

To address these security needs, this project implements a signature-based Intrusion Detection System (IDS) using Suricata**,** deployed within the SDN control layer to provide real-time, deterministic detection of known attacks. Signature-based detection offers high accuracy and low false positives for recognized threat patterns, making it a reliable first line of defense in operational IoT networks. By leveraging SDN's centralized visibility, Suricata inspects MQTT traffic flows and triggers automated mitigations—such as blocking malicious clients—through dynamic flow rule insertion. This integration not only enhances real-time security responsiveness but also demonstrates a practical, deployable model for securing protocol-specific IoT communications.

While signature-based systems excel at detecting known attacks, they cannot identify novel or evolving threats. In this context, the integration of adaptive detection methods—such as machine learning—remains a recommended direction for future enhancement, enabling the system to learn from traffic behavior and detect anomalies beyond predefined signatures. Nevertheless, the current implementation prioritizes stability, interpretability, and real-time performance, establishing a foundational security layer that is both scalable and immediately applicable to MQTT-based IoT environments under SDN control.

1.1 Project Objectives

This project aims to design and implement a practical, SDN-based intrusion detection framework for MQTT-based IoT networks. By integrating signature-based detection with programmable network control, the framework enables real-time traffic monitoring and automated security enforcement, establishing a deployable foundation for IoT security.

To achieve this overarching goal, this research is guided by the following specific objectives:

1.1.1 Design and Implementation of an SDN-Based IoT Network Using the MQTT Protocol

The first objective of this project is to design and implement an Internet of Things (IoT) network environment based on the Software-Defined Networking (SDN) paradigm, utilizing the MQTT protocol as the primary communication mechanism between network entities. The network is emulated using the Mininet platform, where the essential MQTT components—including the broker, publisher, and subscriber—are deployed within a centralized topology connected through an OpenFlow-enabled switch. This architecture enables the separation of the control plane from the data plane, providing enhanced flexibility, programmability, and centralized management of network traffic, which is particularly suitable for dynamic and heterogeneous IoT environments.

1.1.2 Implementation and Tuning of a Signature-Based Intrusion Detection System for MQTT Traffic

The second objective is to implement and configure a signature-based Intrusion Detection System (IDS) specifically tailored for MQTT traffic. The open-source tool Suricata will be deployed and tuned with custom rule-sets designed to recognize known attack patterns prevalent in IoT communications, such as connection floods, malformed packets, and brute-force attempts. The system's performance will be validated using live emulated traffic, ensuring accurate detection and low false positives for defined threats. This objective establishes a reliable, deterministic security layer as the project's operational core.

1.1.3 Integration of the Signature-Based IDS Within an SDN Controller for Real-Time Monitoring and Mitigation

The third objective is to integrate the configured signature-based IDS with an SDN controller (RYU) to enable centralized, real-time traffic analysis and automated response. The controller will be extended to forward traffic to the IDS for inspection and to programmatically enforce security policies based on alerts. Upon detecting malicious activity, the controller will dynamically install OpenFlow rules to block or isolate offending traffic flows. This integration demonstrates a practical, closed-loop security model that combines SDN's programmability with efficient signature-based detection for immediate threat mitigation in IoT networks. Furthermore, this modular architecture provides a foundational platform upon which adaptive detection methods, such as machine learning models, could be integrated in future work to address novel and evolving threats.

1.2 Challenges

The design and implementation of intrusion detection systems in SDN-IoT environments, particularly for MQTT traffic, are associated with a number of technical and operational challenges that must be carefully addressed.

1.2.1 Protocol and Traffic Complexity

Securing MQTT-based IoT communications is inherently difficult due to the lightweight design of MQTT, which prioritizes efficiency over security. MQTT lacks strong authentication, encryption, and integrity checks at the protocol level, making it vulnerable to both volumetric and subtle application-layer attacks such as SlowITe and message manipulation. This challenge is compounded by the fact that MQTT traffic patterns, especially under low-rate attacks, can closely resemble legitimate traffic, rendering traditional signature-based intrusion detection ineffective [25]. Furthermore, IoT systems often exhibit highly heterogeneous device behaviors, varying QoS levels, retain flags, and dup flags that complicate the distinction between benign and malicious flows in real time [25].

1.2.2 Inherent Limitation of Signature-Based Detection:

The core security mechanism relies on pre-defined signatures. This approach guarantees high accuracy and low false positives for known attacks but cannot generalize to detect novel (zero-day) attack variants or sophisticated anomalies that deviate from the rule set. This trade-off between reliability and adaptability is a fundamental characteristic of the implemented system.

1.2.3 Dataset Limitations and Machine Learning Generalization:

Machine learning–driven IDS approaches depend heavily on high-quality, representative datasets. However, many publicly available datasets are either synthetic or generated in controlled environments, which do not fully capture the diversity and noise of real IoT deployments [3][9]. This mismatch between training data and live traffic leads to significant generalization problems, increasing false positive and false negative rates in operational settings [2][6]. In addition, class balancing and feature selection techniques applied during offline training may distort natural traffic distributions, further degrading performance when models are integrated into SDN controllers [1][9]. The lack of large, labeled, protocol-specific datasets for MQTT traffic remains a fundamental bottleneck in developing robust learning-based detection mechanisms.

1.2.4 Real-World Deployment and Performance Constraints

Even when models demonstrate high accuracy in controlled experiments, deploying them in real SDN-IoT environments presents practical challenges. Resource constraints, network dynamics, and controller processing overhead can significantly affect IDS performance [1][5][11]. Furthermore, the effectiveness of ML/DL models depends on continuous adaptation to evolving attack patterns, yet many existing proposals lack mechanisms for real-time updating or online learning [7][14]. This gap between offline model evaluation and real-time operational effectiveness remains a critical barrier to fully robust intrusion detection in MQTT-centric IoT networks

1.3 Research Study

This section reviews contemporary research at the intersection of IoT security, Software-Defined Networking (SDN), and intrusion detection systems (IDS), establishing the context and justification for the proposed project.

1.3.1 IoT Security

The explosive growth of IoT connectivity—now reaching on the order of tens of billions of devices—dramatically expands the network attack surface. Each new IoT device (sensor, appliance, controller, etc.) can be a weak point: many are low-cost and resource-constrained, often shipped with outdated firmware, default credentials, or minimal encryption. The result is a prolific target environment: compromised devices (e.g. cameras, routers) become “entry gateways” for attackers to infiltrate networks, steal data, or disrupt services. In practice, this vulnerability has enabled large-scale IoT botnets (Mirai and its variants) and a string of high-profile attacks. For instance, Mirai hijacked insecure IP cameras and routers for DDoS attacks, and incidents like the Ring doorbell hacks or industrial IoT breaches (e.g. the Jeep Cherokee hack, Colonial Pipeline ransomware) illustrate how unprotected IoT devices can threaten critical infrastructure [4].

- Expanded attack surface: Every connected sensor or actuator adds potential entry points.
- Weak device security: IoT devices often lack rigorous authentication or timely updates, making default-password attacks and exploits commonplace.
- Real-world impact: Compromised IoT nodes have powered botnets (Mirai) and enabled network-wide disruptions, highlighting the urgent need for robust IoT defense

1.3.2 SDN-Based Centralized Control in IoT

Traditional IoT networks lack a unified management plane. Software-Defined Networking (SDN) offers a solution by decoupling the control plane from data forwarding, placing a central controller “brain” over the network. This centralized control is highly relevant to IoT: the SDN controller can maintain a global view of all IoT traffic and quickly enforce security policies. For example, SDN enables real-time network monitoring and anomaly detection across all links[1]. It also supports dynamic traffic management and fine-grained policy enforcement – the controller can isolate suspicious flows or quarantine devices on-the-fly[1]. By using SDN in an IoT setting, we gain unified visibility and programmability: we can push flow rules, collect statistics, and reconfigure the network centrally to mitigate attacks. In short, SDN’s programmable centralized architecture makes it well-suited to manage the complex, heterogeneous IoT domain and respond to threats quickly [1][5].

1.3.3 ML-Based IDS vs. Traditional IDS

Conventional Traditional intrusion detection systems (IDS) rely on signature/rule-based methods: they match network traffic against known attack patterns. Open-source IDS like Suricata and Snort embody this approach. Suricata, in particular, is a high-performance, multi-threaded IDS/IPS that fully supports Snort's rule format [31]. It can process tens of gigabits per second on commodity hardware, making it practical for real-time network defense. Signature-based IDS are accurate for known threats and impose low overhead, but they require continuous rule updates and cannot detect zero-day attacks. By contrast, machine learning (ML)–based IDS learn traffic patterns and can flag novel anomalies without explicit signatures. ML/AI systems have advantages in adaptability and detection of new or obfuscated attacks, as many surveys note. However, ML models demand extensive training data and computational resources.

Our project, by design, focuses exclusively on rule-based IDS using Suricata (no ML). The rationale is to leverage Suricata's mature signature engine and low-latency performance, targeting MQTT-specific behaviors. Suricata's open-source engine also allows integration of custom protocol parsing and strict validations [31]. In short, we adopt Suricata because it can immediately detect defined MQTT exploits at line rate, without the need for model training.

1.3.4 SDN in IoT

Researchers agree that SDN is a natural fit for IoT security. SDN's centralized controller can enforce network-wide policies in heterogeneous IoT deployments. For example, Al Hayajneh *et al.* (2020) proposed an SDN-based architecture to mitigate man-in-the-middle and other attacks in HTTP-based IoT systems, showing that the SDN controller enables rapid threat response and granular filtering [1]. However, the proposed architecture primarily focuses on HTTP-based communication and does not address lightweight IoT protocols such as MQTT, which limits its applicability to modern IoT environments.

Similarly, recent surveys note that SDN "provides centralized control, dynamic traffic management, [and] comprehensive network visibility" in IoT networks, which jointly enhance security[4]. In practice, SDN controllers like RYU can act as the network "brain" for IoT: they collect flow statistics from switches/emulated IoT links (e.g., via Mininet) and apply policies globally. These features (centralized monitoring, on-demand flow rules, isolation between network segments) allow SDN-enabled IoT systems to detect anomalies and quarantine threats more effectively than legacy networks [4][1]. Also, a representative and well-structured SDN-based security architecture for IoT is proposed by Karmakar et al. (2020), where the SDN controller acts as a policy decision authority for the entire IoT network [8]. In their architecture, IoT devices (sensors and actuators) connect to the network through OpenFlow-enabled gateways, which are controlled by a centralized SDN controller. The controller maintains a global view of the network topology, device identities, and traffic flows, enabling it to enforce fine-grained security policies dynamically. On the other hand, while the architecture effectively establishes centralized policy enforcement and device-level access control, it does not incorporate intelligent intrusion detection mechanisms capable of analyzing traffic behavior or identifying sophisticated network-based attacks. Recent studies have also explored SDN-based architectural enhancements for IoT environments. For example, Bedhief et al. proposed an SDN–Docker-based IoT architecture that leverages centralized SDN control and containerized services to improve scalability, flexibility, and traffic management, while providing a suitable foundation for deploying advanced security mechanisms in IoT networks [13].

1.3.5 Rule-based IDS (Suricata) in SDN-IoT

The SDN controller collects flow data and can forward traffic or alerts to Suricata for inspection. Recent work has shown how Suricata can be extended to IoT protocols: notably, Husnain *et al.* enhanced Suricata with a custom MQTT parsing engine to strictly validate MQTT packet fields [31]. Their approach catches protocol violations (wrong lengths, missing fields, etc.) and tags malicious payloads before reaching devices. On top of that, Suricata’s rule engine can match protocol-specific keywords extracted by the parser. For example, rules were defined to detect MQTT flooding: one rule drops packets when *CONNECT* requests from one source exceed 10 per minute, and another drops excess *PUBLISH* packets above 100 per minute. In practice, this means Suricata can block known attack patterns on-the-fly. As Husnain *et al.* report, “when an adversary tries to send multiple connection requests to an MQTT broker, rule 1 will be triggered, and the Suricata rule engine will drop these malicious packets… Likewise, when an adversary attempts to flood publish packets, the Suricata rule engine will detect and drop this malicious attempt”.

By combining SDN and Suricata, we gain centralized enforcement: the controller can reroute suspicious flows (via OpenFlow rules) while Suricata provides deep packet inspection. This structure allows real-time reaction: the SDN controller applies policies based on Suricata alerts (e.g., isolating a device, rerouting traffic). In summary, a Suricata-based IDS in our SDN-IoT setup offers a practical, high-speed defense layer. We leverage Suricata’s proven, signature-based capabilities to catch MQTT-specific exploits (as in) and use SDN for network-wide policy control.

Table 1.1- A comparative table for recent works

Ref No.	Year	Context / Environment	Security Focus	Approach Used	Key Contributions	Limitations (Relevant to This Work)
[17]	2023	SDN-enabled IoT network (simulation)	Network intrusion detection	Deep learning–based IDS integrated with SDN controller	Demonstrated improved detection accuracy by leveraging SDN control and DL models	ML-based, requires offline training, high computational overhead, not MQTT-aware
[18]	2022	SDN-IoT testbed	DDoS & MiTM mitigation	Belief-Based Secure Correlation with SDN flow control	Enabled real-time mitigation through centralized SDN policies	Limited to specific attacks; no protocol-level inspection such as MQTT
[24]	2021	Conceptual IoT security framework	IoT intrusion detection	IDS taxonomy and reference architecture	Provided structured classification of IoT IDS techniques and architectures	No implementation, evaluation, or protocol-specific analysis
[25]	2020	MQTT-based IoT environment	Denial-of-Service attacks	Experimental attack analysis (SlowITe)	Identified stealthy DoS attack exploiting MQTT keep-alive mechanism	No IDS or mitigation mechanism proposed
[26]	2026	Experimental IoT MQTT setup	Man-in-the-Middle attacks	Protocol-level message manipulation	Demonstrated MQTT message interception and manipulation vulnerabilities	No automated detection or IDS integration
[31]	2022	IoT network with MQTT broker	MQTT vulnerabilities	Rule-based IDS using Suricata with MQTT parsing	Extended Suricata to support MQTT inspection and custom attack rules	No SDN-based control or mitigation
[30]	2020	General network traffic environment	Traffic anomaly detection	Deep Packet Inspection (DPI)	Demonstrated DPI effectiveness for detecting anomalous traffic patterns	Not IoT- or MQTT-specific; lacks SDN integration
[23]	2023	IoT-enabled digital logistics	Communication security threats	Threat modeling methodology	Identified IoT communication attack surfaces and threats	Conceptual only; no IDS implementation
[27]	2022	IoT protocol comparison study	Communication protocol security	Analytical comparison of MQTT and AMQP	Highlighted lightweight design and security trade-offs of MQTT	No attack modeling or IDS analysis
[32]	2020	Simulated IoT MQTT network	IDS evaluation	Dataset generation (MQTT-IoT-IDS2020)	Provided labeled MQTT traffic dataset for IDS research	Dataset only; no detection or mitigation mechanism

The comparative analysis highlights that recent IDS research in IoT environments increasingly emphasizes protocol-aware detection and realistic traffic modeling. While several works demonstrate the effectiveness of rule-based IDS such as Suricata for detecting MQTT-specific attacks through signature and protocol validation mechanisms, most implementations remain limited to standalone IDS deployments without integration into SDN control frameworks. Additionally, many studies focus on detection only, lacking real-time mitigation or centralized enforcement capabilities. These observations reinforce the need for a practical SDN-enabled, rule-based IDS that leverages MQTT-aware rules and operates within a live control loop, as pursued in this project.

1.3.6 Research gap

Despite Existing SDN-IoT security literature largely emphasizes ML/DL models or generic network traffic. Few studies specifically explore rule-based MQTT detection within an SDN framework. For instance, Husnain *et al.* demonstrate MQTT rules in Suricata, but many similar works focus on detection only, without integrating live mitigation in an SDN loop. In general, published IDS solutions often stop at offline experiments on datasets; they rarely become real-time modules within an SDN controller. Moreover, MQTT itself has seen limited IDS research despite its prevalence in IoT. Many IoT security gaps remain: the need for protocol-aware rulesets, real-time deployment, and closed-loop response in SDN environments.

Our project addresses these gaps by designing and implementing a live SDN-IoT testbed (using RYU, Mosquitto, etc.) that employs Suricata as a rule-based IDS. We will write and tune MQTT-specific Suricata rules (inspired by works like [19]) and deploy them in real network flows. The SDN controller will harness Suricata alerts to enforce policies immediately (e.g. dropping or rerouting offending traffic). By focusing on a practical, rule-based solution, we bridge the divide between theoretical IoT security proposals and a deployable system. In essence, we combine centralized SDN control, protocol-aware Suricata rules, and realistic MQTT traffic (MQTTset) to build a live, MQTT-centric IDS – demonstrating a clear path from literature to a working defensive mechanism.

2. Theoretical Foundation

2.1 Introduction

This chapter provides theoretical and architectural treatment of the problem space for intrusion detection in SDN-enabled IoT networks that use MQTT for messaging and machine learning (ML) for detection. Its goal is to give a rigorous foundation to build on in subsequent experimental chapters. The chapter is organized into the following major sections: Internet of Things (IoT) architecture and security constraints; MQTT protocol semantics and security implications; IDS taxonomies and detection paradigms; software-defined networking (SDN) fundamentals; the Mininet + Ryu research and prototyping stack; detailed ML considerations for network IDS; and finally datasets with emphasis on MQTTset and other IoT datasets used in IDS research. Each section explains essential concepts, derives design implications for IDS, and points to representative references.

2.2 Internet of Things (IoT)

The Internet of Things, commonly referred to as IoT, is a transformative technological concept that has reshaped the way we connect, communicate, and interact with the world around us. At its core, IoT is a network of interconnected physical objects, or "things," embedded with various sensors, software, and other technologies, which enable them to collect and exchange data with other devices and systems over the internet. These objects can range from everyday household appliances and industrial machines to wearable devices and vehicles. IoT brings these diverse entities together to create a seamless, data-driven ecosystem where information flows freely and interactions occur without human intervention [19].

2.2.1 Scope, Heterogeneity, and Layered Architectures

Typical architectural models partition IoT into layers that map to different responsibilities (perception, network, processing, application, and management/business) [22], and show the flow of telemetry and control information from sensors to cloud/analytics platforms. Devices vary widely — from constrained microcontrollers with tiny RAM and intermittent connectivity, to more capable edge gateways and cloud services.

2.2.2 Resource Constraints and Security Consequences

IoT nodes are frequently resource-constrained in computational power, memory, energy, and networking capability. This amplifies design trade-offs: heavy cryptography or continuous packet capture is often infeasible on devices themselves, so IDS solutions usually relocate intensive computation to gateways, fog nodes, or cloud services. These layers also mediate protocols (for example, a constrained device may use a lightweight data transport like CoAP or MQTT-SN), which affects what features an IDS can observe. The multiplicity of communication patterns — publish/subscribe in MQTT, client/server in HTTP, and lightweight sensor profiles — underscores the need for protocol-aware detection strategies.

2.2.3 Attack Surface and Threat Modeling

The IoT attack surface is broad, encompassing physical tampering, insecure local networks, weak or absent authentication, and application-layer protocol exploitation. Common adversarial objectives include data breach (eavesdropping), integrity compromise (message spoofing/injection), denial-of-service (DoS/DDoS), and device co-option for botnets.

2.2.4 IoT Threat Model and Attack Taxonomy

Internet of Things (IoT) ecosystems introduce unique threat models due to their heterogeneity, resource constraints, and pervasive deployment [23][24]. Attackers may be classified as external (remote) or internal (insider), and as passive (eavesdropping, data collection) or active (disruptive attacks) [24]. Threat modeling approaches (e.g. STRIDE) applied to IoT reveal vulnerabilities at multiple layers: physical (e.g. device tampering), link (e.g. flooding/jamming), network (e.g. routing attacks, IP spoofing), and application (e.g. protocol misuse) [23][24]. Common IoT attack categories include those that target availability (e.g. Denial of Service), confidentiality/integrity (e.g. Man-in-the-Middle), identity (e.g. spoofing), and protocol-specific misuse. These can be further organized as passive *vs.* active and internal *vs.* external threats.

Table 2.1- IoT Attack Types

Attack Category	**Description**	**Examples**
Denial of Service (DoS)/Distributed DoS (DDoS)	Flooding or overwhelming resources (network, CPU, memory) to make services unavailable[25]. Often distributed via botnets.	*Mirai*-family botnets launching volumetric TCP/UDP floods from compromised IoT devices[24]. Shodan-identified open MQTT brokers used in flooding experiments[25].
Man-in-the-Middle (MitM)	Intercepting or altering communications between devices without their	*MQTT MitM:* Unencrypted MQTT links can be intercepted and modified (shown via Polymorph

Attack Category	Description	Examples
	knowledge[23][26]. Violates confidentiality/integrity.	tool)[26]. Eavesdropping on wireless sensor data [23].
Spoofing (Identity Forgery)	Impersonating a legitimate device or node by falsifying identifiers (e.g. IP/MAC addresses, credentials)[23][24].	IP/MAC spoofing on an IoT LAN; ARP spoofing to redirect traffic[23]. Crafting fake sensor packets to appear from trusted device.
Flooding (Traffic Injection)	Sending excessive data/requests at high rate to congest links or queues. A form of DoS but often at specific protocol level (e.g. network-layer flooding).	ICMP/UDP floods targeting an IoT gateway; MQTT flooding by rapid subscribe/publish cycles.
MQTT-Specific Attacks	Protocol misuse exploiting MQTT features: topic abuse (wildcards), retained-message abuse, QoS flaws, broker attacks, unauthorized publications.	Wildcard subscription ("#") leaks all topics; using retained messages for outdated state injection; exploiting QoS 2 handshake to overflow broker; taking over a broker via credential/default weaknesses.

IoT attack surfaces include physical (e.g. sensor node tampering), link (e.g. radio jamming or flooding at MAC layer), network (routing attacks, flooding), transport (SYN floods, UDP storms) and application layers (protocol abuses, malware installation). IoT threats are especially severe because compromised devices can serve as both victims and unwitting attackers (e.g. exploited in DDoS)[24]. Overall, this threat model highlights that in SDN-enabled IoT, detection systems must consider multi-layer attacks: from radio jamming and brute-force on low-power links up to advanced MITM and application-layer protocol exploits.

Denial-of-Service (DoS/DDoS) attacks are prevalent in IoT. For example, the Mirai botnet in 2016 leveraged insecure IoT cameras to conduct one of the largest DDoS attacks ever recorded[24]. Mirai's technique—infecting embedded devices to coordinate massive TCP/SYN floods—illustrates how IoT-specific botnets can execute volumetric attacks. Similarly, flooding of network links (e.g. ARP or ICMP floods) can paralyze IoT gateways. Man-in-the-Middle attacks exploit unsecured links; studies show that without TLS, MQTT packets between publishers and subscribers can be captured and modified[26]. Spoofing attacks (e.g. MAC/IP spoofing) allow attackers to masquerade as legitimate IoT nodes[23]. Finally, MQTT-specific attacks exploit protocol semantics: unsubscribed (wildcard) subscriptions can expose all traffic, and setting retained flags on messages can force brokers to serve malicious or stale data. These examples underscore the need for a comprehensive IoT attack taxonomy that includes both generic network threats and those unique to IoT/MQTT environments.

2.3 MQTT Protocol

The MQTT (Message Queuing Telemetry Transport) protocol is a lightweight message protocol based on the publish/subscribe principle, designed for communication between remote devices where a small code footprint is required and network bandwidth is limited. In other words, MQTT is a machine-to-machine (M2M) / Internet of Things connectivity protocol. It is lightweight and can support the smallest devices with minimal available resources, yet it is reliable enough to guarantee that important messages always reach their destination [27].

2.3.1 Key Principles

The MQTT protocol is built using several key principles aimed at ensuring message delivery while minimizing their volume:

1) Publish/Subscribe

The publish/subscribe principle defines the mechanism for transferring messages between the source and the recipient of information. According to it, the source publishes information, indicating the topic (category) to which it belongs, and the recipient subscribes to the message topics that interest them. The information source is commonly called a publisher, and the recipient is called a subscriber. When implementing this principle, the publisher does not necessarily need to know the subscriber's location and vice versa, and the transmission of messages from one publisher to many subscribers is ensured.

2) Topics and Subscriptions

In MQTT, messages are published with the indication of topics, which can be considered as names of subject areas. To receive certain messages, subscribers subscribe to the corresponding topics. A subscription can be set by explicitly specifying specific topics for which messages need to be received, or it can be defined using wildcard characters (e.g., #) to receive messages on various related topics.

3) Quality of Service Levels

The MQTT protocol defines three quality of service (QoS) levels for message delivery. The higher the quality level, the more effort the server applies to ensure message delivery. Higher quality levels ensure more reliable message delivery, but they can consume more network bandwidth or lead to delays in message delivery.

4) Message Retention

In the MQTT protocol, the server retains a message even after sending it to all current subscribers of the corresponding topic. In case a new subscriber subscribes to this topic, all previously retained messages are sent to them.

5) Persistent Connections and Session Cleanup

When a client connects to a server via the MQTT protocol, the client sets the value of the clean session flag parameter. This parameter determines the need to delete (clean up) the client's subscriptions when they disconnect from the server (end of session). If the value is true, all of the client's subscriptions are deleted; if false, the connection is considered persistent, and all of the client's subscriptions are preserved after any disconnection. With a value of false, messages received during the client's disconnection period and having a high quality of service level are retained and delivered to the client after the connection is restored. Using the session cleanup parameter is not mandatory.

6) Will Messages

When a client connects to a server, it can inform the server that it has a will message that should be published on a specific topic(s) in case of unexpected client disconnection. The will message is especially useful for alert or security settings where the system must immediately know when a remote sensor has lost connection to the network.

2.3.2 Protocol Publisher/Subscriber Logic

Publishers own the information and initiate communication with the broker via a unique ID. They send information at specified intervals and log their actions. Subscribers, on the other hand, initiate communication, listen for incoming information, and respond to it if it's what they need. They also log their actions for troubleshooting purposes.

2.3.3 Data Integrity in the MQTT Protocol (Quality of Service)

Data integrity in MQTT is upheld through its Quality of Service (QoS) levels.

There are three QoS levels:

1) Level 0: Doesn't guarantee message delivery, as in Figure 2.1.

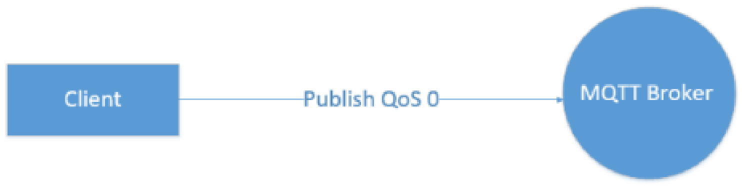


Figure 2.1- Quality of Service level 0: delivery at most once.

2) Level 1: Ensure messages are delivered at least once, with retransmission if necessary, as in figure 2.2.

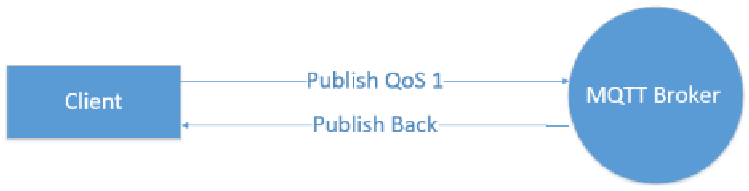


Figure 2.2- Quality of Service level 1: delivery at least once.

3) Level 2: Guarantees that each message is delivered exactly once but is the slowest option and involves a detailed confirmation process, as in figure 2.3.

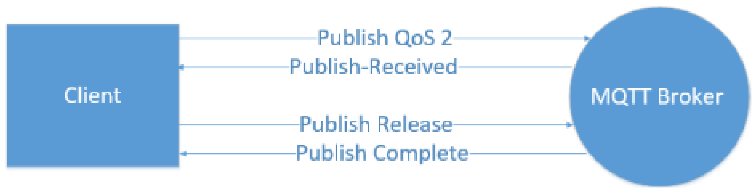


Figure 2.3- Quality of Service level 2: delivery exactly once

To achieve Level 2, a sender and recipient go through a series of acknowledgments. The sender initially sends a Publish packet, and the recipient responds with a Publish-Received packet. If the sender doesn't receive this acknowledgment; it resends the Publish packet. Once acknowledged, the sender sends a Publish-Release packet, and the recipient replies with a Publish-Complete packet. This process ensures that the message is delivered and received with confirmation. If a packet is lost, the sender is responsible for retransmitting it [28].

2.3.4 MQTT Architecture

MQTT Architecture consists of two main components:

- Client: Establishes network connections to the MQTT broker and can function as both a publisher and subscriber. It can publish messages, subscribe to topics, unsubscribe from topics, and disconnect from the broker.

- Broker: Controls the distribution of information, receiving messages from publishers, filtering them, determining recipients, and forwarding messages to subscribers. The broker also handles client requests, including subscriptions and unsubscriptions. This architecture enables efficient communication between devices and is integral to the operation of MQTT in IoT scenarios.

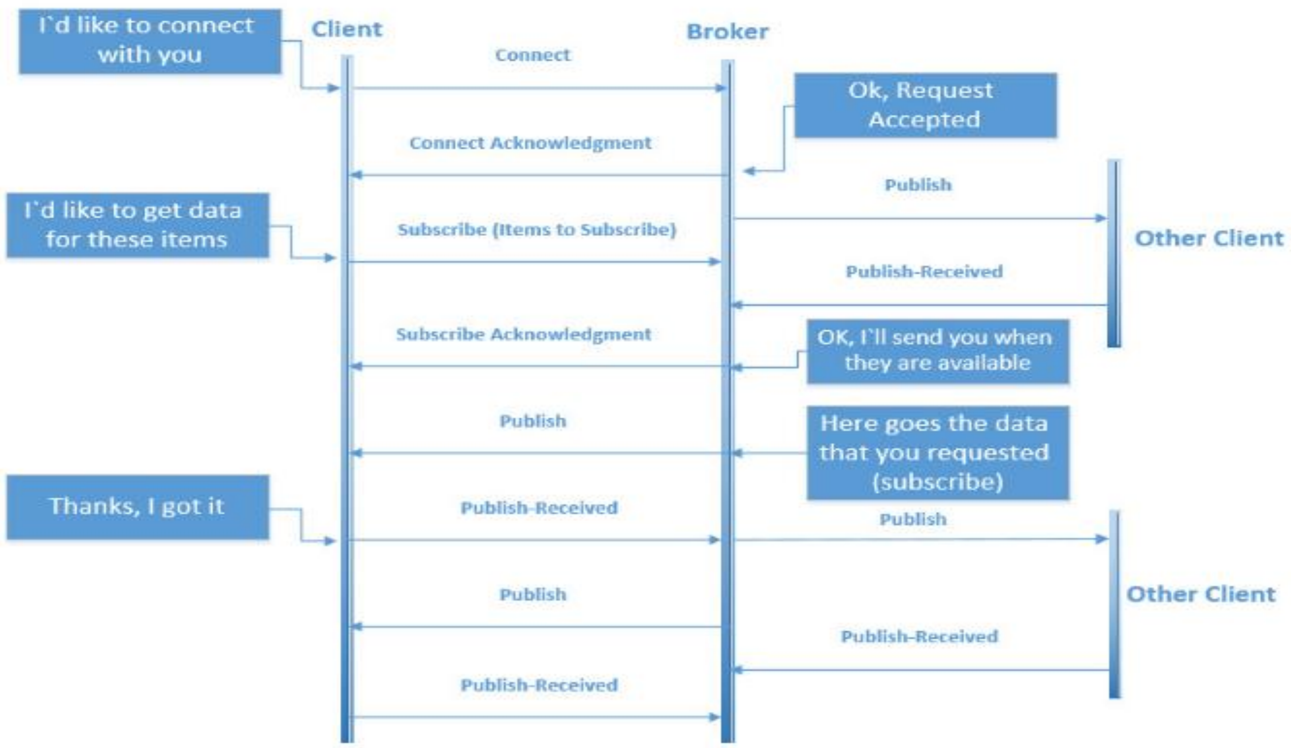


Figure 2.4- Working of MQTT

Figure 2.4 illustrates the core MQTT publish/subscribe communication flow between a client, a broker, and a subscribing client, including connection establishment and message acknowledgment [28].

2.3.5 MQTT Broker

In MQTT, the central element is the broker, a crucial component for establishing MQTT networks. Brokers can be hosted online in the cloud or locally on a private device. Many brokers are open source, allowing users to obtain credentials, install, and set them up on their devices. This sets up a broker with a specific IP, making it part of the private local network, akin to a SCADA server, where clients interact with the server.

MQTT brokers come in two types: public and private. Public brokers allow any device to publish and subscribe to topics without privacy restrictions. However, they are not recommended for production use and are better suited for learning or experimentation. In contrast, private brokers are more secure. Only devices authorized by the user can publish and subscribe to the broker's topics. Private brokers are suitable for both production and prototyping purposes [28].

2.3.6 Security Properties

According to Ferrari, to attain a deeper understanding of MQTT's security, it is essential to consider security from the network layer to the application layer. Notably, selecting an MQTT broker designed for security is crucial. While open-source brokers may suffice for home automation, professional deployments demand heightened security considerations.

2.3.6.1 Problem Statement

MQTT plays a pivotal role in critical global deployments for device connectivity. Security is a paramount concern for such deployments. MQTT, being a layer 7 protocol, relies on underlying layers, and it is typically used in conjunction with TLS to establish encrypted communication channels between devices and brokers Despite these security measures, MQTT deployments invariably emphasize user authentication and authorization mechanisms. These mechanisms ensure that only authorized devices can connect to the broker. Devices must present valid credentials, certificates, and authorization to publish and subscribe to specific topics and perform designated actions. Nevertheless, MQTT's security is not without its vulnerabilities, and there is evidence to support this claim [28].

2.3.6.2 Problem Analysis

Despite MQTT's numerous advantages, it may not be suitable for all scenarios, as observed in

Some of the protocol's drawbacks include:

1. Slower transmit cycles when compared to the Constrained Application Protocol (CoAP), which can be critical for systems with a substantial number of devices
2. Challenges in resource discovery, given that MQTT employs a flexible topic subscription system, whereas CoAP employs a stable resource discovery system
3. Limited security encryption. MQTT primarily lacks encryption, despite TLS/SSL usage.
4. Scalability concerns, particularly in establishing globally scalable networks when compared to alternative protocols.

While MQTT remains a fitting solution for many applications, addressing its security and interoperability challenges necessitates consultation with Internet of Things development experts [28].

2.3.6.3 Problem Explanation

MQTT communication may become unreliable, and a publisher might disconnect without warning. In anticipation of such situations, a publisher can set up a "last will and testament." This message is stored on the broker and is dispatched to subscribers in the event of an unanticipated disconnection. The message typically contains information to identify the disconnected publisher and guide appropriate actions. MQTT was initially designed to facilitate efficient data transmission over unreliable and costly communication lines. As a result, security received limited attention during its design and implementation. Nevertheless, some security options exist, albeit with increased data transmission and storage requirements. Network security can be an effective measure, where the network itself is secured, making the transmission of unencrypted MQTT data irrelevant. Notably, security breaches must arise from within the network, such as through malicious actors or network intrusions.

MQTT allows the use of usernames and passwords for establishing connections with brokers. Regrettably, in pursuit of lightweight communication, these usernames and passwords are transmitted in plain text. This approach was acceptable in 1999 but is insecure in today's context, where wireless network communications can be easily intercepted. While usernames and passwords can prevent unintentional connections; they offer little protection against malicious actors. The commonly employed SSL/TLS, which runs atop TCP/IP, introduces significant overhead to the otherwise lightweight MQTT communication.

MQTT communication between the client and broker involves the generation of TCP/IP packets by the client, which are processed to generate specific outputs sent to the broker. The broker, upon receiving these packets, responds with TCP/IP packets, and the client deciphers the resulting output and forwards it to the mapper for abstraction. However, MQTT brokers may exhibit non-deterministic behavior due to factors such as latency or timeouts [28].

2.3.6.4 Problem Conclusion

One primary source of security and privacy issues on the Internet of Things is the prevalence of insecure default configurations. A default misconfigured MQTT server poses significant risks, as anyone with access can intercept all messages transmitted through it due to MQTT's use of wildcards. Alarmingly, many MQTT servers are poorly configured and accessible without authentication on the public internet. For instance, the world's first Internet of Things search engine uncovered almost 67,000 exposed MQTT servers, most of which lack authentication.

The MQTT protocol allows anyone to subscribe to broadcasted topics without authentication, making it highly vulnerable. Fundamental security principles for the Internet of Things include data confidentiality, availability, integrity, and privacy.

The current implementation of MQTT primarily provides identity, authentication, and authorization within its security mechanisms. However, relying on TCP/IP for sending/receiving data is insecure, even though TCP/IP packets are generated and sent to the broker. MQTT clients can subscribe to topic filters, publish messages to topic names, and receive messages from subscribed topics. Notably, the topic filter allows the use of wildcard characters for simultaneous subscriptions or subscriptions to multiple topics.

Despite MQTT's widespread use, it was not originally designed with a strong emphasis on security, as evidenced by its standing as the second most popular Internet of Things messaging protocol. The substantial growth of MQTT's usage demands the development of a secure version that accounts for resource-constrained devices. Therefore, prioritizing MQTT security requirements are essential [28].

2.4 CoAP

Constrained Application Protocol is a lightweight M2M protocol from the IETF CoRE (Constrained RESTful Environments) Working Group. CoAP supports both request/response and resource/observe (a variant of publish/subscribe) architecture. CoAP is mainly developed to interoperate with HTTP and the RESTful Web through simple proxies. Unlike MQTT, CoAP uses Universal Resource Identifier (URI) instead of topics. Publisher publishes data to the URI and subscriber subscribes to a particular resource indicated by the URI. When a publisher publishes new data to the URI, then all the subscribers are notified about the new value as indicated by the URI. CoAP is a binary protocol and normally requires fixed header of 4-bytes with small message payloads up to maximum size dependent on the web server or the programming technology. CoAP uses UDP as a transport protocol and DTLS for security. Thus, clients and servers communicate through connectionless datagrams with less reliability. However, it uses “confirmable” or “non-confirmable” messages to provide two different levels of QoS. Where, confirmable messages must be acknowledged by the receiver with an ACK packet and non- confirmable messages are not. CoAP offers more functionality than MQTT such as it supports content negotiation to express a preferred representation of a resource; this allows client and server to evolve independently, adding new representations without affecting each other [29].

2.5 AMQP

AMQP is a lightweight M2M protocol, which was de- veloped by John O’Hara at JPMorgan Chase in London, UK in 2003. It is a corporate messaging protocol designed for reliability, security, provisioning and interoperability. AMQP supports both request/response and publish/subscribe architecture. It offers a wide range of features related to messaging such as a reliable queuing, topic-based publish- and-subscribe messaging, flexible routing and transactions. AMQP communication system requires that either the pub- lisher or consumer creates an “exchange” with a given name and then broadcasts that name. Publishers and consumers use the name of this exchange to discover each other. Subsequently, a consumer creates a “queue” and attaches it to the exchange at the same time. Messages received by the exchange have to be matched to the queue via a process called “binding”. AMQP exchanges messages in various ways: directly, in fanout form, by topic, or based on headers. AMQP is a binary protocol and normally requires fixed header of 8-bytes with small message payloads up to maximum size dependent on the broker/server or the programming technology. AMQP uses TCP as a default transport protocol and TLS/SSL and SASL for security. Thus, the communication between client and broker is a connection-oriented. Reliability is one of the core features of AMQP, and it offers two preliminary levels of Quality of Service (QoS) for delivery of messages: Unsettle Format (not reliable) and Settle Format (reliable) [29].

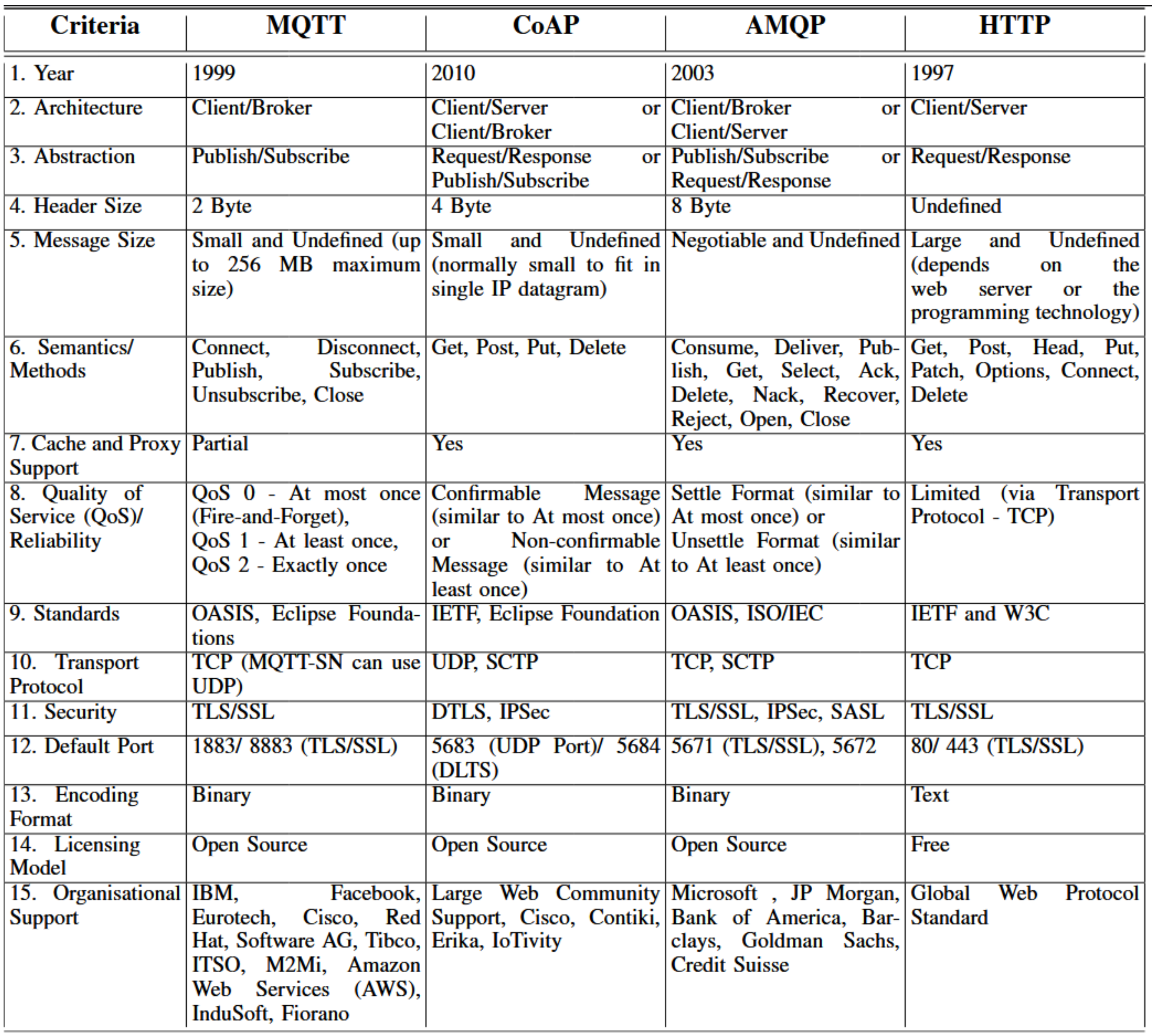
2.6 HTTP

Hyper Text Transfer Protocol is predominantly a web messaging protocol, which was originally developed by Tim Berners-Lee. Later, it was de- veloped by IETF and W3C jointly and first published as a standard protocol in 1997. HTTP supports request/response RESTful Web architecture. Analogous to CoAP, HTTP uses Universal Resource Identifier (URI) instead of topics. Server sends data through the URI and client receives data through particular URI. HTTP is a text-based protocol and it does not define the size of header and message payloads rather it depends on the web server or the programming technology. HTTP uses TCP as a default transport protocol and TLS/SSL for security. Thus, communication between client and server is a connection-oriented. It does not explicitly define QoS and requires additional support for it. HTTP is a globally accepted web messaging standard offers several features such as persistent connections, request pipelining, and chunked transfer encoding [29].

2.7 Analysis of Messaging Protocols for IoT Systems: HTTP, COAP, AMQP and MQTT

These messaging protocols are very extensive and different from each other because they have been evolved through different processes and needs. Also, their precise and relative comparisons depend on the types of IoT systems, devices, resources, applications, and specific conditions and requirements of the system.

Table 2.2- Comparative Analysis of Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP, and HTTP[29]



Criteria	MQTT	CoAP	AMQP	HTTP
1. Year	1999	2010	2003	1997
2. Architecture	Client/Broker	Client/Server or Client/Broker	Client/Broker or Client/Server	Client/Server
3. Abstraction	Publish/Subscribe	Request/Response or Publish/Subscribe	Publish/Subscribe or Request/Response	Request/Response
4. Header Size	2 Byte	4 Byte	8 Byte	Undefined
5. Message Size	Small and Undefined (up to 256 MB maximum size)	Small and Undefined (normally small to fit in single IP datagram)	Negotiable and Undefined	Large and Undefined (depends on the web server or the programming technology)
6. Semantics/ Methods	Connect, Disconnect, Publish, Subscribe, Unsubscribe, Close	Get, Post, Put, Delete	Consume, Deliver, Publish, Get, Select, Ack, Delete, Nack, Recover, Reject, Open, Close	Get, Post, Head, Put, Patch, Options, Connect, Delete
7. Cache and Proxy Support	Partial	Yes	Yes	Yes
8. Quality of Service (QoS)/ Reliability	QoS 0 - At most once (Fire-and-Forget), QoS 1 - At least once, QoS 2 - Exactly once	Confirmable Message (similar to At most once) or Non-confirmable Message (similar to At least once)	Settle Format (similar to At most once) or Unsettle Format (similar to At least once)	Limited (via Transport Protocol - TCP)
9. Standards	OASIS, Eclipse Foundations	IETF, Eclipse Foundation	OASIS, ISO/IEC	IETF and W3C
10. Transport Protocol	TCP (MQTT-SN can use UDP)	UDP, SCTP	TCP, SCTP	TCP
11. Security	TLS/SSL	DTLS, IPSec	TLS/SSL, IPSec, SASL	TLS/SSL
12. Default Port	1883/ 8883 (TLS/SSL)	5683 (UDP Port)/ 5684 (DLTS)	5671 (TLS/SSL), 5672	80/ 443 (TLS/SSL)
13. Encoding Format	Binary	Binary	Binary	Text
14. Licensing Model	Open Source	Open Source	Open Source	Free
15. Organisational Support	IBM, Facebook, Eurotech, Cisco, Red Hat, Software AG, Tibco, ITSO, M2Mi, Amazon Web Services (AWS), InduSoft, Fiorano	Large Web Community Support, Cisco, Contiki, Erika, IoTivity	Microsoft , JP Morgan, Bank of America, Barclays, Goldman Sachs, Credit Suisse	Global Web Protocol Standard

This table presents a comparative analysis of the four widely accepted and emerging messaging protocols for IoT systems MQTT, CoAP, AMQP and HTTP based on several criteria to introduce their characteristics comparatively [29].

2.4 Software-Defined Networking (SDN)

Software Defined Network is a network architecture paradigm characterized by the fundamental decoupling of the network's control logic (the control plane) from the underlying hardware that forwards traffic (the data plane).

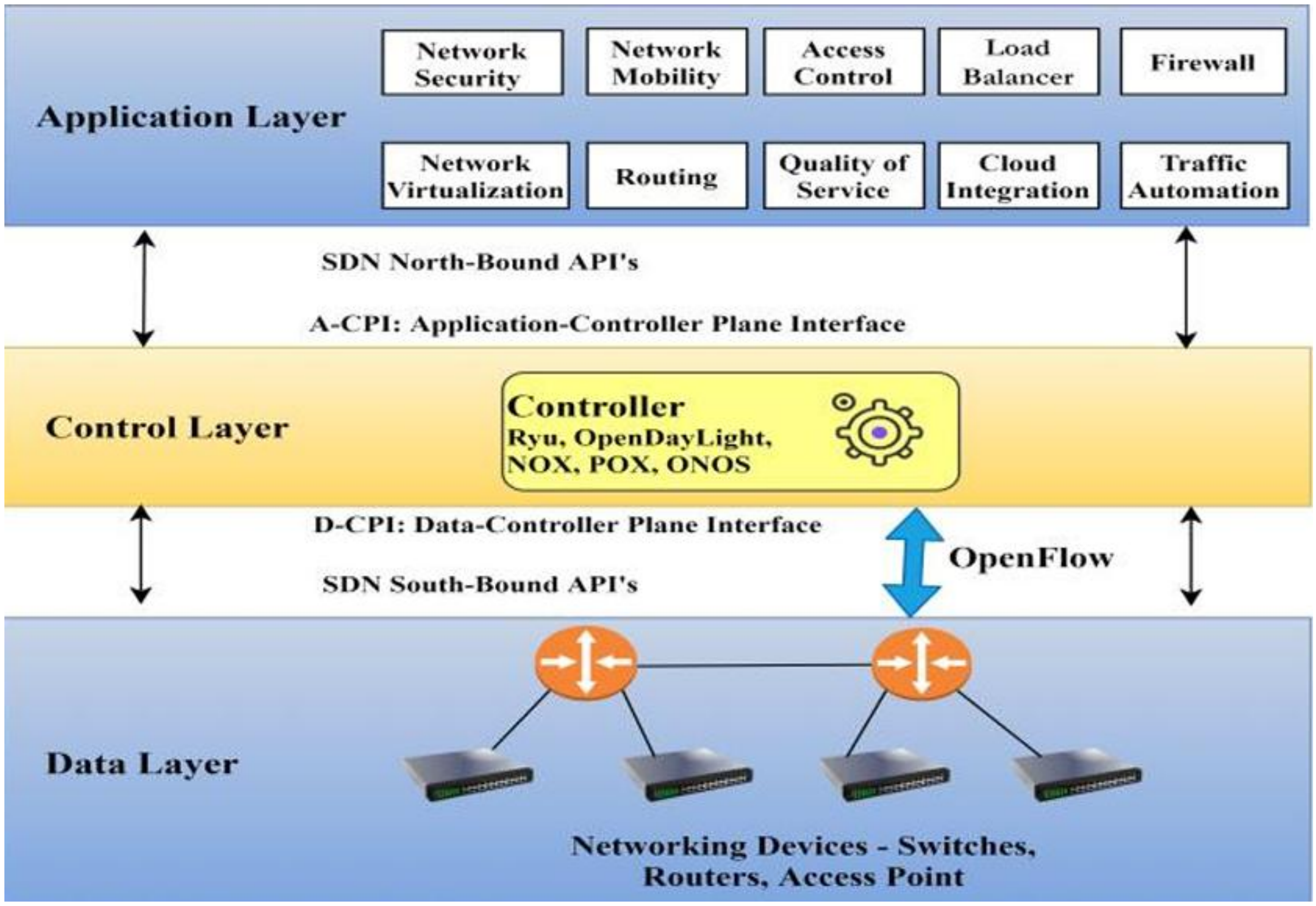


Figure 2.1– Block diagram of a three-layered SDN architecture [10].

As shown in the figure, the SDN architecture is structured into three distinct layers: The Application Layer, which hosts network services and applications; the Control Layer, represented by the SDN Controller; and the Data Layer, comprising physical and virtual networking devices. Communication between these layers is facilitated via standardized APIs: The Northbound API (A-CPI) for application-controller interaction and the Southbound API (D-CPI), typically implemented through OpenFlow, for controller-device communication. This separation of control and data planes enhances network flexibility, management, and automation.

2.4.1 SDN Architectural Principles

The proposed system leverages the inherent flexibility of the SDN architecture, where a centralized software-based orchestrator abstracts the underlying hardware infrastructure. Through the use of Northbound and Southbound APIs, the controller provides a programmable environment capable of reconfiguring data plane forwarding rules on the fly. This architectural separation is a critical enabler for network automation, allowing for reactive security controls and real-time IDS integration through direct manipulation of flow-based forwarding tables.

2.4.2 Flow Abstraction and Telemetry

OpenFlow and similar southbound protocols expose a flow abstraction. Flows are defined by header match rules (e.g., IP, port, VLAN, higher-layer fields) and associated actions (forward, drop, modify). Switches maintain flow tables and often provide counters and statistics for installed flows. Controllers can query switches for flow statistics or configure switches to export active flow summaries. This telemetry makes SDN controllers a natural aggregation point for IDS. Flow features (byte/packet counts, durations) and event streams (PACKET_IN events) enable both real-time anomaly detection and retrospective analysis.

2.5 Intrusion Detection System (IDS)

An intrusion detection system is a software or hardware-based security mechanism designed to monitor a network or computer system for abnormal activities, malicious behaviors, or policy violations. Unlike a firewall, which proactively filters traffic based on rules, an IDS primarily acts as a "listen-only" or monitoring device that alerts administrators when it detects a suspected intrusion.

2.5.1 Taxonomy

IDS can be categorized along several axes:

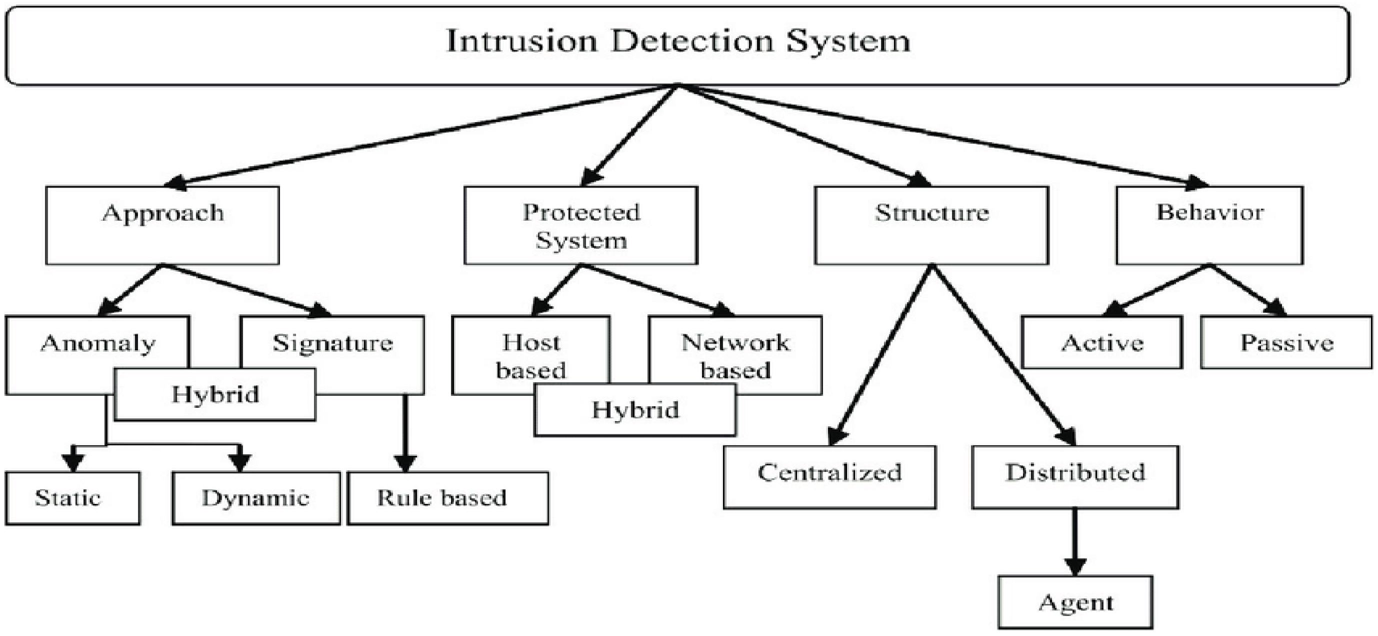


Figure 2.2- IDS Classification

2.5.2 Signature-Based IDS

Signature-based approaches detect known attacks by matching traffic against predefined patterns or rules. They provide precise detection for known threats but fail for zero-day or polymorphic attacks.

2.5.3 Anomaly-Based IDS

Anomaly-based approaches learn a model of normal behavior and flag deviations; they are better suited to novel attack detection but are susceptible to false positives and concept drift as legitimate behavior changes.

2.5.4 Hybrid IDS

Hybrid systems combine both signatures for known exploits and anomaly detectors to flag unknown behaviors.

2.5.5 Host-Based IDS (HIDS)

Host Intrusion Detection System is an architectural component deployed directly onto an endpoint to analyze system-level events. HIDS provides granular visibility into the host's internal state, monitoring data such as system calls, file integrity, audit logs, and process behavior to detect threats that bypass perimeter defenses.

2.5.6 Network-Based IDS (NIDS)

Network Intrusion Detection System is a specialized security solution deployed at strategic points within a network to monitor and analyze all network traffic.

2.5.7 Flow-Based NIDS

Flow-based NIDS analyze aggregated, summarized metadata (e.g., NetFlow, headers, byte counts) of network traffic connections to identify anomalies or attacks, making them suitable for high-speed networks.

2.5.8 Packet-Based NIDS

Packet-Based NIDS analyze network traffic at the individual packet level, inspecting both the header and the payload for malicious activities. Unlike flow-based systems that look at aggregated, summary data, packet-based NIDS perform Deep Packet Inspection (DPI) on raw, real-time data packets to identify signatures of known attacks or anomalies.

2.6 Mininet

Mininet is a widely used, lightweight network emulator that runs a complete (virtualized) network on a single machine. It creates virtual hosts, switches (often Open vSwitch), and links in Linux network namespaces, enabling researchers to prototype and validate SDN applications and topologies without physical hardware.

2.7 Ryu Controller

Ryu is an open-source SDN controller framework implemented in Python. It supports multiple OpenFlow versions and provides simple APIs for developing controller applications.

2.8 In-Depth Analysis: Signature-Based Intrusion Detection Systems

A Signature-Based Intrusion Detection System (SIDS) is a deterministic security mechanism that operates by comparing observed network activity against a predefined database of known attack patterns, known as signatures or rules. This paradigm, also referred to as misuse detection, forms the operational core of this project, implemented using the Suricata IDS engine. Its primary function is to identify malicious behavior that matches these documented indicators of compromise with high precision.

2.8.1 Core Principles and Architecture

The efficacy of a SIDS hinges on its rule set. Each signature is a formalized description of a specific attack sequence, exploit attempt, or malicious payload. These rules are typically written in a specialized language that can inspect various protocol layers. For example, a rule might detect a denial-of-service (DoS) attack by matching an excessive rate of CONNECT packets in MQTT traffic or identify a buffer overflow attempt by a specific byte sequence in a packet payload [31].

The detection process follows a linear workflow:

1. Packet Capture: Network traffic is captured, typically at a strategic point like a gateway, switch, or broker.
2. Protocol Decoding & Normalization: Packets are decoded according to their protocol stack (e.g., Ethernet, IP, TCP, MQTT) and normalized to ensure consistent inspection.
3. Rule Matching: The decoded content is compared against all active signatures in the rule set. This involves pattern matching on packet headers, payloads, and derived flow characteristics.
4. Alert Generation: Upon a successful match, the system generates an alert, logs the event, and can trigger a predefined response action.

2.8.2 Signature Design and Rule Engineering for IoT/MQTT

Crafting effective signatures for IoT environments, and specifically for the MQTT protocol, requires moving beyond generic network-level indicators to incorporate protocol-aware semantics. High-quality signatures must balance detection accuracy with performance to avoid overwhelming the system in high-traffic IoT networks [31]. Key considerations include:

- Protocol-Specific Patterns: Signatures must parse MQTT control packet headers to identify malicious patterns, such as:
    - Flooding Attacks: Rules triggering on abnormal rates of CONNECT, PUBLISH, or SUBSCRIBE packets from a single source.
    - Malformed Packets: Rules detecting violations of the MQTT protocol specification, such as invalid packet lengths, reserved flag usage, or illegal QoS combinations.
    - Content-Based Attacks: Signatures that inspect topic strings or payloads for known exploit code or unauthorized command injections.
- Stateful Detection: Advanced SIDS can maintain session state to detect multi-packet attack sequences that would be missed by inspecting individual packets in isolation.
- Performance Optimization: Rule sets must be optimized to minimize processing overhead. Techniques include organizing rules by protocol, using fast-pattern matching for common threats, and disabling irrelevant rules for the monitored environment.

2.8.3 Strengths

1. For known attacks, SIDS provides highly reliable detection with minimal false alarms, as alerts are generated only for exact pattern matches.
2. Every alert is directly traceable to a specific rule, making the system's behavior transparent and easy for security analysts to understand and verify.
3. Once deployed, a well-tuned SIDS requires minimal ongoing configuration and computational resources compared to adaptive learning systems, making it suitable for resource-constrained environments.

2.8.4 Limitations

1. The fundamental limitation of a SIDS is its inability to detect novel attacks or variants for which no signature exists. The security posture is only as current as the rule database.
2. The rule set requires continuous updates from security vendors or dedicated teams to protect against newly discovered threats, incurring operational costs.
3. Attackers can modify their techniques (e.g., through polymorphism or encryption) to avoid matching static signatures without changing the attack's underlying effect.

2.8.5 Integration within SDN for Automated Mitigation

The true power of a SIDS in a modern network is realized through integration with a responsive control plane. In an SDN architecture, this creates a closed-loop security system:

1. The SIDS performs detection at a chosen point, such as the network perimeter or the IoT broker.
2. Upon generating an alert, the system logs the event (e.g., to an eve.json file) with details including the source IP, timestamp, and matched signature.
3. The SDN controller, via a dedicated security application, polls these alerts, parses them, and maps the malicious source to the corresponding network flow.
4. The controller programmatically enforces mitigation by installing a dynamic OpenFlow rule on the relevant switch to drop all packets from the malicious source, thereby neutralizing the threat at the network layer.

This integration transforms passive detection into active, automated defense, leveraging SDN's programmability to provide immediate, network-wide protection. It exemplifies a practical and deployable security model for IoT networks, where rapid response is critical.

3. System Design and Planning

This section presents the overall design and planning of the proposed system, including the functional and non-functional requirements, the network architecture, and the interaction between system components. It outlines how the intrusion detection mechanism is integrated within the SDN-IoT environment to meet security, performance, and scalability objectives while ensuring compatibility with MQTT-based communication.

3.1 Functional and Non-Functional Requirements

We derive the system requirements from the problem definition and IoT/SDN context. Functional requirements (FR) specify what the system must do, while non-functional requirements (NFR) constrain how it should perform. Table 2 summarizes the key requirements. For example, the system must capture and inspect MQTT traffic in real time and flag anomalous or malicious flows (FR). It must automatically alert administrators and reconfigure the switch to isolate or drop detected attacks, while allowing normal MQTT publish/subscribe traffic to continue. Non-functional requirements include real-time performance (e.g. processing latency under ~1 s) and high detection accuracy (to minimize false positives). The system should scale to typical IoT deployments (supporting dozens of hosts) and run on commodity hardware (using Python and open-source tools) for ease of development and deployment. Compliance with standard security practices (e.g. secure MQTT and OpenFlow channels) is also ensured.

Table 3-3 –System Requirements

Requirement Type	Requirement
Functional	The system must monitor MQTT publish/subscribe traffic in real time.
	The system must detect malicious activity (e.g. spoofed or flood messages).
	The system must generate alerts/logs and instruct the SDN switch to block or isolate flows identified as attacks.
	The system must allow legitimate publisher–subscriber communication to proceed unhindered.
Non-Functional	Detection latency must be very low (on the order of seconds or less) to allow immediate mitigation.
	The system must operate on standard Linux-based hosts (using Python) and interoperate with MQTT and SDN standards.
	The design should be modular and maintainable (with clear controller applications and logs).

3.2 Technology Stack Selection

The selection of an appropriate technology stack is critical for creating a functional, reproducible, and well-integrated experimental platform. This project leverages a suite of mature, open-source tools that are standard in SDN research and IoT security testing. Each component was chosen for its proven capabilities, strong community support, and interoperability within the designed architecture.

- Python 3 (Programming Language): Python serves as the primary development language due to its extensive library ecosystem for network programming and its native integration with several core tools. The paho-mqtt library facilitates MQTT client scripting, while the Ryu SDN controller framework itself is implemented in Python. This synergy simplifies the development of custom controller applications (e.g., for log parsing and flow rule installation) and Mininet topology scripts, streamlining the entire development and integration workflow.

- Ryu SDN Controller: Ryu was selected as the SDN controller for this project. It is a robust, component-based, open-source framework written in Python that provides full support for the OpenFlow protocol. Its primary advantages are an accessible API for developing custom applications and an active development community. In this architecture, Ryu runs the core controller logic, manages the OpenFlow switch, and hosts the custom security application that interfaces with the IDS to execute mitigation actions.
- Mininet Network Emulator: Mininet is employed to create a realistic, software-defined network topology on a single host. It instantaneously creates virtual hosts (IoT publishers, subscribers, attackers) and connects them via Open vSwitch instances that are controlled by the Ryu controller. Its lightweight virtualization and precise control over network links make it the *de facto* standard for SDN prototyping and research, allowing for the rapid iteration and testing of network security policies.
- Mosquitto MQTT Broker: The Eclipse Mosquitto broker implements the MQTT protocol and acts as the central messaging hub for the emulated IoT network. It is lightweight, fully open-source, and widely adopted in both industry and research, ensuring protocol compliance and reliability. All legitimate MQTT communication (publishers to subscribers) and attack traffic directed at the broker flow through this component, making it the focal point for traffic monitoring
- Suricata: Suricata is a high-performance, open-source Network Intrusion Detection and Prevention System (IDS/IPS). It was chosen as the core detection engine for its robust signature-based detection capabilities, efficient multi-threaded architecture, and comprehensive rule language. Suricata is deployed to perform Deep Packet Inspection (DPI) on all traffic directed at the Mosquitto broker, enabling the precise identification of known MQTT-based attack patterns through a customized rule set
- Supporting Tools: We A set of auxiliary tools is used for development, testing, and validation:

  For network traffic capture and protocol analysis, Wireshark/Tshark & tcpdump are essential for debugging and verifying both normal operation and attack simulations.

  For traffic generation, hping3 is versatile command-line packet crafter used to generate synthetic attack traffic (e.g., TCP-SYN floods) for testing the IDS and mitigation response.

  For version control throughout the project's development lifecycle, Git.

This integrated stack provides a cohesive environment for developing, testing, and validating a signature-based IDS within an SDN-secured IoT network, ensuring both technical feasibility and alignment with established research methodologies.

4. Implementation and Evaluation

The theoretical foundation established in previous chapters—encompassing Software-Defined Networking (SDN) architecture, the security vulnerabilities of IoT protocols like MQTT, and the operational principles of signature-based intrusion detection—now culminates in a practical, deployable security system. This chapter details the concrete translation of these concepts into a functional, integrated framework that bridges the gap between passive traffic monitoring and active, automated network defense.

Building upon the configured environment described in Chapter 3, the implementation phase involved the creation and integration of several core components within a live SDN-IoT testbed. The process followed a structured, three-stage pipeline:

1. Network and Service Emulation: Constructing a representative IoT topology using Mininet, deploying the Mosquitto MQTT broker, and establishing legitimate publisher-subscriber communication flows.
2. Security Sensor Deployment and Tuning**:** Integrating the Suricata IDS as a dedicated security sensor, configuring it with a custom rule-set tailored to detect MQTT-specific attack patterns, and validating its detection capability.
3. Orchestration and Automated Response: Developing a custom Ryu controller application to parse Suricata alerts and dynamically enforce mitigation by installing OpenFlow rules to block malicious traffic.

This chapter systematically documents each stage of this pipeline. It justifies key design decisions, such as the placement of the IDS, the logic for mapping security alerts to network flows, and the mechanism for automated rule insertion. The goal is to present a coherent blueprint for a closed-loop security system that leverages SDN's programmability to create a responsive and practical defense layer for IoT networks.

4.1 System Architecture and Topology Design

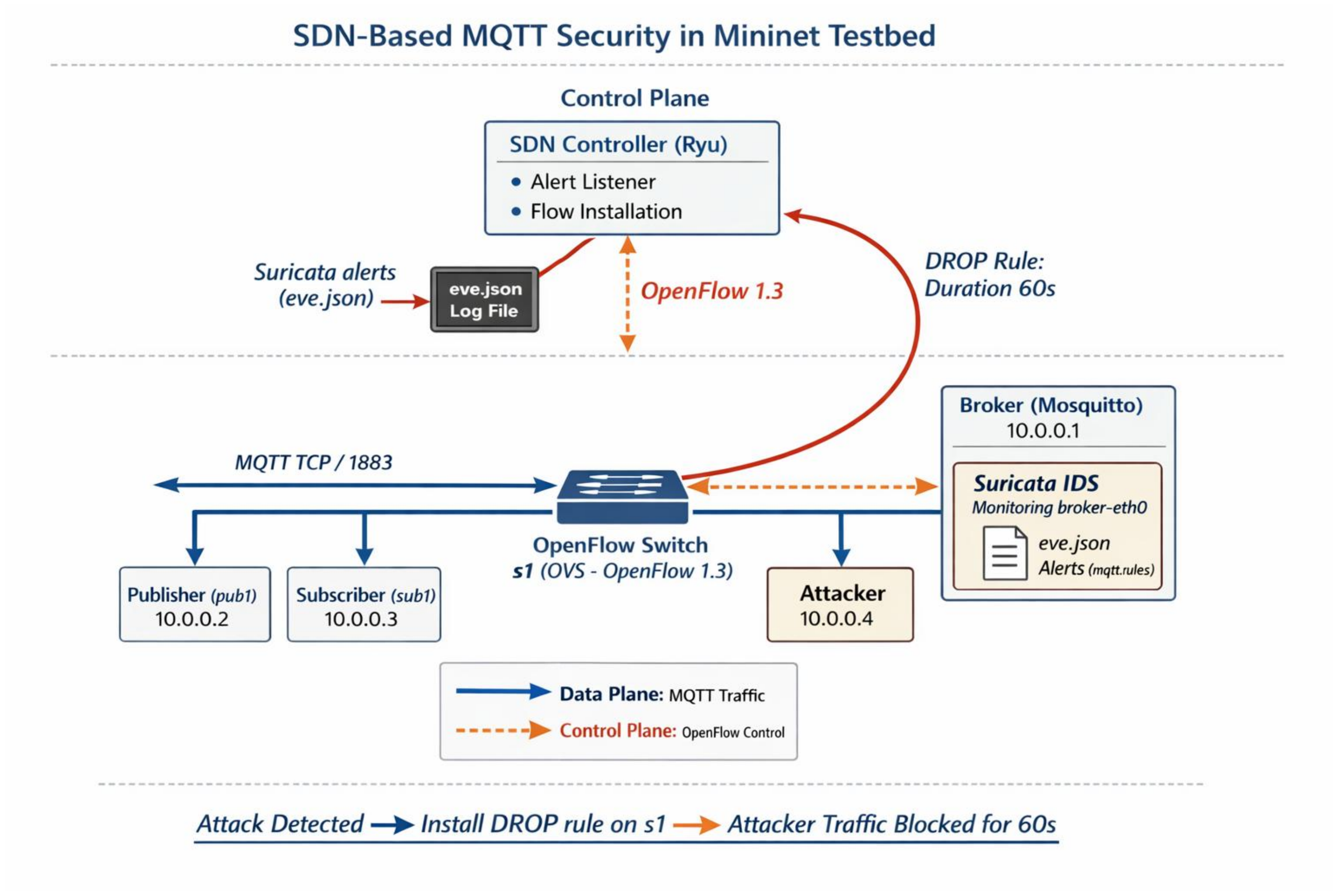
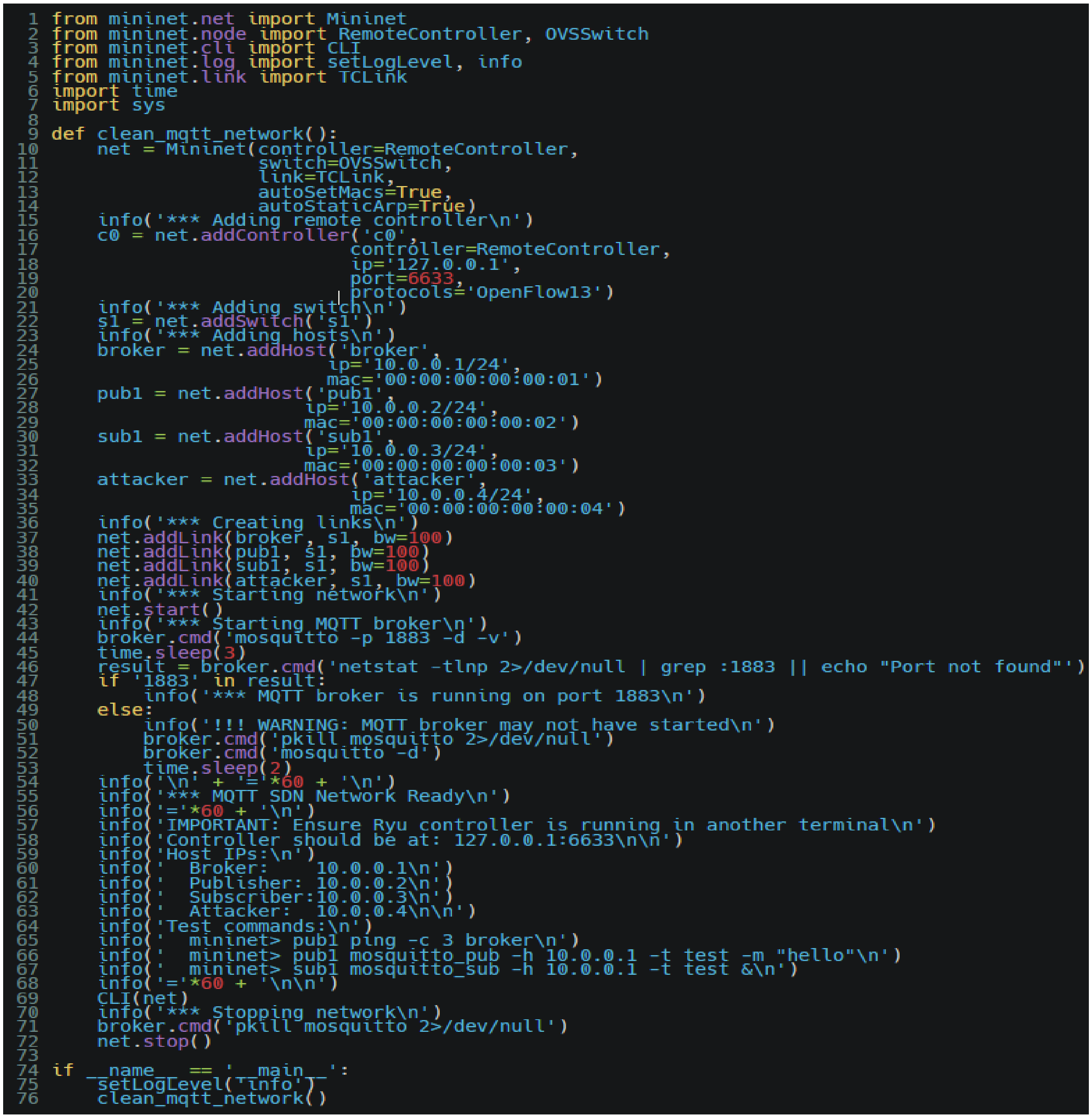


Figure 4.1 -System traffic flow

The architecture (Figure 4-1) comprises an MQTT publisher, subscriber, attacker host, and a central MQTT broker, all connected via an OpenFlow-enabled switch. The Ryu SDN controller (running our custom mqtt_ids.py app see Figure 4.3) sits logically above the switch. The publisher and subscriber simulate IoT sensors communicating via the broker; the attacker injects malicious MQTT traffic (e.g. false data or flooding messages). All network packets traverse the OpenFlow switch, which forwards flows based on the controller's rules. The Ryu controller periodically polls the switch for flow statistics and the ids for alert logs. As network traffic passes through, if the traffic matches known signature, the controller dynamically pushes a new flow rule (or drop rule) to the switch, thereby isolating the malicious traffic. Otherwise, the switch forwards packets normally (e.g. publisher → broker → subscriber). In this way, legitimate MQTT communications remain uninterrupted while attacks are detected and mitigated in real time. The centralized SDN control plane enables this global view and rapid response.



```python
from mininet.net import Mininet
from mininet.node import RemoteController, OVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink
import time
import sys

def clean_mqtt_network():
    net = Mininet(controller=RemoteController,
                  switch=OVSSwitch,
                  link=TCLink,
                  autoSetMacs=True,
                  autoStaticArp=True)
    info('*** Adding remote controller\n')
    c0 = net.addController('c0',
                          controller=RemoteController,
                          ip='127.0.0.1',
                          port=6633,
                          protocols='OpenFlow13')
    info('*** Adding switch\n')
    s1 = net.addSwitch('s1')
    info('*** Adding hosts\n')
    broker = net.addHost('broker',
                        ip='10.0.0.1/24',
                        mac='00:00:00:00:00:01')
    pub1 = net.addHost('pub1',
                      ip='10.0.0.2/24',
                      mac='00:00:00:00:00:02')
    sub1 = net.addHost('sub1',
                      ip='10.0.0.3/24',
                      mac='00:00:00:00:00:03')
    attacker = net.addHost('attacker',
                          ip='10.0.0.4/24',
                          mac='00:00:00:00:00:04')
    info('*** Creating links\n')
    net.addLink(broker, s1, bw=100)
    net.addLink(pub1, s1, bw=100)
    net.addLink(sub1, s1, bw=100)
    net.addLink(attacker, s1, bw=100)
    info('*** Starting network\n')
    net.start()
    info('*** Starting MQTT broker\n')
    broker.cmd('mosquitto -p 1883 -d -v')
    time.sleep(3)
    result = broker.cmd('netstat -tlnp 2>/dev/null | grep :1883 || echo "Port not found"')
    if '1883' in result:
        info('*** MQTT broker is running on port 1883\n')
    else:
        info('!!! WARNING: MQTT broker may not have started\n')
        broker.cmd('pkill mosquitto 2>/dev/null')
        broker.cmd('mosquitto -d')
        time.sleep(2)
    info('\n' + '='*60 + '\n')
    info('*** MQTT SDN Network Ready\n')
    info('='*60 + '\n')
    info('IMPORTANT: Ensure Ryu controller is running in another terminal\n')
    info('Controller should be at: 127.0.0.1:6633\n\n')
    info('Host IPs:\n')
    info('  Broker:     10.0.0.1\n')
    info('  Publisher:  10.0.0.2\n')
    info('  Subscriber:10.0.0.3\n')
    info('  Attacker:   10.0.0.4\n\n')
    info('Test commands:\n')
    info('  mininet> pub1 ping -c 3 broker\n')
    info('  mininet> pub1 mosquitto_pub -h 10.0.0.1 -t test -m "hello"\n')
    info('  mininet> sub1 mosquitto_sub -h 10.0.0.1 -t test &\n')
    info('='*60 + '\n\n')
    CLI(net)
    info('*** Stopping network\n')
    broker.cmd('pkill mosquitto 2>/dev/null')
    net.stop()

if __name__ == '__main__':
    setLogLevel('info')
    clean_mqtt_network()
```

Figure 4.2- Custom Mininet Topology

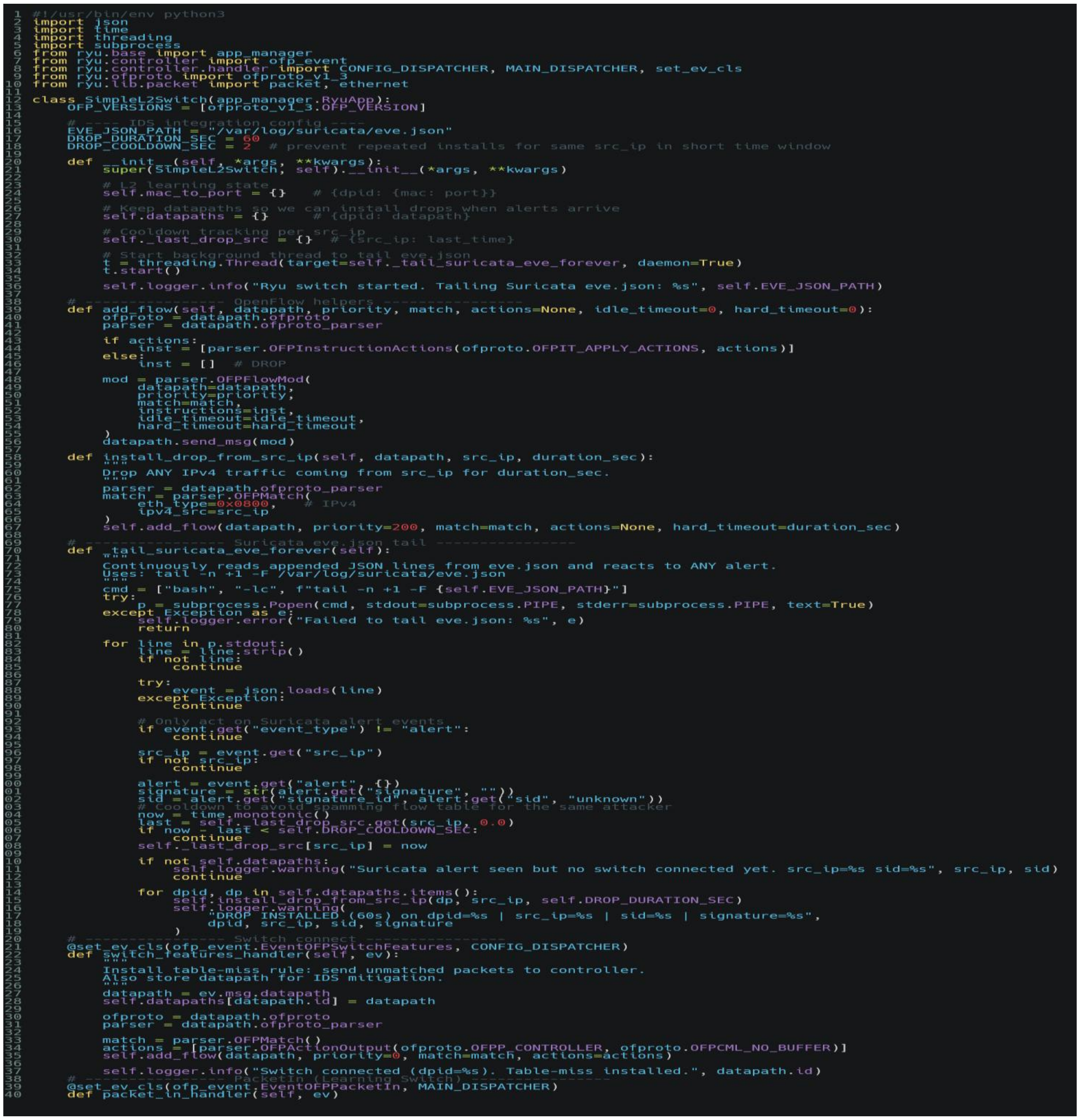
The proposed system architecture emulated in mininet using a custom python script (Figure 4.2) defining the network topology, including MQTT clients, the Mosquitto broker host, and the OpenFlow switch.

4.2 Security Module Implementation

The core intelligence of the automated security system resides in a custom application developed for the Ryu SDN controller. This application acts as the orchestrator, bridging the gap between the signature-based detection performed by Suricata and the programmable enforcement capability of the SDN data plane. Its primary function is to create a closed-loop control system: it consumes security alerts, makes a deterministic decision, and executes a network-level mitigation action.

The design of this application follows an event-driven polling model. Instead of performing computationally intensive packet inspection itself, it leverages the specialized detection engine of Suricata. The application periodically reads Suricata's structured alert log (eve.json), parses the relevant fields to identify the malicious source, and translates this information into an OpenFlow FlowMod message that instructs the switch to drop all future packets from that host. This design exemplifies the SDN principle of separating complex security logic from the high-speed forwarding plane.

The following code segment (Figure 4.3) illustrates the core loop and primary functions of this orchestrator application



```python
#!/usr/bin/env python3
import json
import time
import threading
import subprocess
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER, set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet, ethernet

class SimpleL2Switch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    # ---- IDS integration config ----
    EVE_JSON_PATH = "/var/log/suricata/eve.json"
    DROP_DURATION_SEC = 60
    DROP_COOLDOWN_SEC = 2  # prevent repeated installs for same src_ip in short time window

    def __init__(self, *args, **kwargs):
        super(SimpleL2Switch, self).__init__(*args, **kwargs)

        # L2 learning state
        self.mac_to_port = {}   # {dpid: {mac: port}}

        # Keep datapaths so we can install drops when alerts arrive
        self.datapaths = {}     # {dpid: datapath}

        # Cooldown tracking per src_ip
        self._last_drop_src = {}  # {src_ip: last_time}

        # Start background thread to tail eve.json
        t = threading.Thread(target=self._tail_suricata_eve_forever, daemon=True)
        t.start()

        self.logger.info("Ryu switch started. Tailing Suricata eve.json: %s", self.EVE_JSON_PATH)

    # ---------------- OpenFlow helpers ----------------
    def add_flow(self, datapath, priority, match, actions=None, idle_timeout=0, hard_timeout=0):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        if actions:
            inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
        else:
            inst = []  # DROP

        mod = parser.OFPFlowMod(
            datapath=datapath,
            priority=priority,
            match=match,
            instructions=inst,
            idle_timeout=idle_timeout,
            hard_timeout=hard_timeout
        )
        datapath.send_msg(mod)

    def install_drop_from_src_ip(self, datapath, src_ip, duration_sec):
        """
        Drop ANY IPv4 traffic coming from src_ip for duration_sec.
        """
        parser = datapath.ofproto_parser
        match = parser.OFPMatch(
            eth_type=0x0800,    # IPv4
            ipv4_src=src_ip
        )
        self.add_flow(datapath, priority=200, match=match, actions=None, hard_timeout=duration_sec)

    # ---------------- Suricata eve.json tail ----------------
    def _tail_suricata_eve_forever(self):
        """
        Continuously reads appended JSON lines from eve.json and reacts to ANY alert.
        Uses: tail -n +1 -F /var/log/suricata/eve.json
        """
        cmd = ["bash", "-lc", f"tail -n +1 -F {self.EVE_JSON_PATH}"]
        try:
            p = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
        except Exception as e:
            self.logger.error("Failed to tail eve.json: %s", e)
            return

        for line in p.stdout:
            line = line.strip()
            if not line:
                continue

            try:
                event = json.loads(line)
            except Exception:
                continue

            # Only act on Suricata alert events
            if event.get("event_type") != "alert":
                continue

            src_ip = event.get("src_ip")
            if not src_ip:
                continue

            alert = event.get("alert", {})
            signature = str(alert.get("signature", ""))
            sid = alert.get("signature_id", alert.get("sid", "unknown"))
            # Cooldown to avoid spamming flow table for the same attacker
            now = time.monotonic()
            last = self._last_drop_src.get(src_ip, 0.0)
            if now - last < self.DROP_COOLDOWN_SEC:
                continue
            self._last_drop_src[src_ip] = now

            if not self.datapaths:
                self.logger.warning("Suricata alert seen but no switch connected yet. src_ip=%s sid=%s", src_ip, sid)
                continue

            for dpid, dp in self.datapaths.items():
                self.install_drop_from_src_ip(dp, src_ip, self.DROP_DURATION_SEC)
                self.logger.warning(
                    "DROP INSTALLED (60s) on dpid=%s | src_ip=%s | sid=%s | signature=%s",
                    dpid, src_ip, sid, signature
                )
    # ---------------- Switch connect ----------------
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        """
        Install table-miss rule: send unmatched packets to controller.
        Also store datapath for IDS mitigation.
        """
        datapath = ev.msg.datapath
        self.datapaths[datapath.id] = datapath

        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, priority=0, match=match, actions=actions)

        self.logger.info("Switch connected (dpid=%s). Table-miss installed.", datapath.id)
    # ---------------- PacketIn (Learning Switch) ----------------
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev)
```

Figure 4.3- Ryu Orchestrator

The implemented orchestrator application successfully realizes the design objectives of a lightweight, responsive security control loop.

The current implementation focuses on IP-based blocking. In a more sophisticated deployment, rules could incorporate transport-layer fields (ports) for more surgical mitigation. Furthermore, the polling mechanism introduces an inherent latency (up to the polling interval plus processing time) between attack detection and mitigation. For ultra-low latency requirements, a direct IPC (Inter-Process Communication) mechanism between Suricata and the Ryu app, such as a Unix socket, could be explored.

4.3 Attack Simulation and Signature-Based Detection

Following the deployment of the network infrastructure and security orchestrator, the next critical phase involved validating the detection and mitigation pipeline against realistic threats. This section describes the attack simulations conducted to stress-test the system, focusing on common vulnerabilities in MQTT-based IoT communications. For each attack type, the conceptual basis, simulation technique, and the corresponding custom Suricata detection signature are explained. This process not only validates the functional integration of Suricata and the Ryu controller but also illustrates the practical workflow of threat modeling, signature development, and automated response that forms the cornerstone of the proposed security framework.

4.3.1 MQTT Publish Flood

A publish flood attack is a denial-of-service (DoS) technique targeting the MQTT broker. An attacker rapidly sends a high volume of PUBLISH messages, aiming to exhaust the broker's processing capacity, memory, or network bandwidth. This can degrade service for legitimate clients or cause a complete outage. The attack exploits the lightweight nature of the MQTT protocol, which lacks built-in rate limiting

Simulation:

The attack was simulated using a custom Python script (Figure 4.4) deployed on an attacker host within the Mininet topology. The script connects to the broker and publishes messages to a specified topic at a configurable, high frequency for a set duration (e.g., 10 seconds), emulating a malicious client.

Detection Signature:

To detect this volumetric attack, two rate-based Suricata rules were implemented with different sensitivity thresholds. The first rule (sid:1000001) serves as a high-volume alert, while the second (sid:1000002) provides a more aggressive, lower-threshold detection for rapid response.

Rule 1: High-Volume Alert

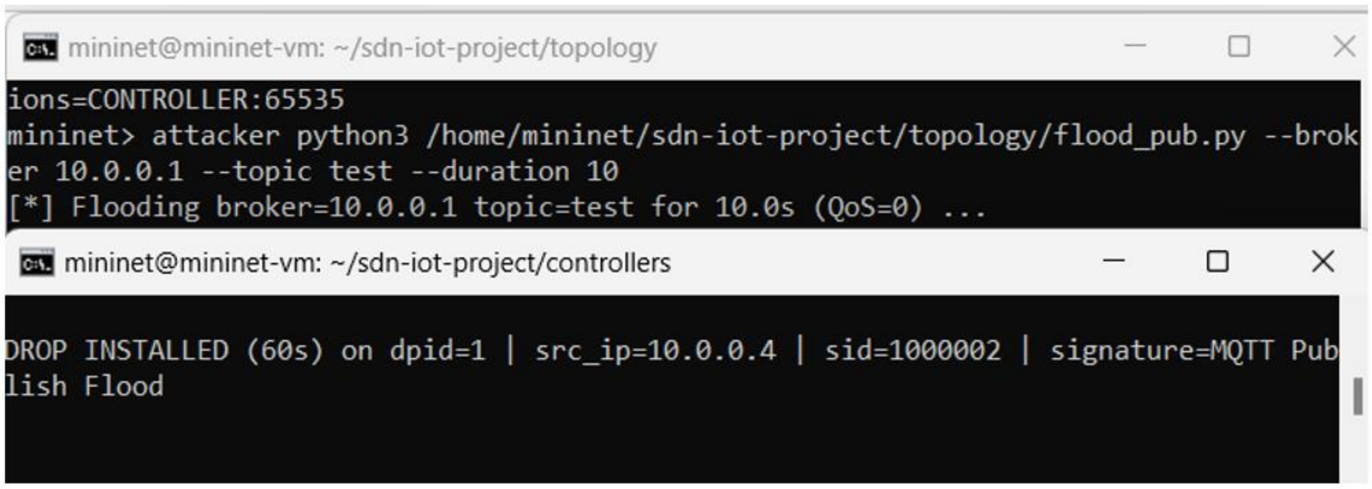
alert mqtt any any -> any any (msg:"MQTT Publish Flood - Level 1"; flow:to_server,established; mqtt.type:PUBLISH; threshold:type both, track by_src, count 40, seconds 5; sid:1000001; rev:1;)

Triggers if a single source sends more than 40 PUBLISH packets to any broker within a 5-second window.

Rule 2: Aggressive Detection

alert mqtt any any -> any any (msg:"MQTT Publish Flood - Level 2"; flow:to_server,established; mqtt.type:PUBLISH; threshold:type both, track by_src, count 10, seconds 5; sid:1000002; rev:1;)

Triggers if a single source sends more than 10 PUBLISH packets to any broker within a 5-second window.


mininet@mininet-vm: ~/sdn-iot-project/topology

```
ions=CONTROLLER:65535
mininet> attacker python3 /home/mininet/sdn-iot-project/topology/flood_pub.py --brok
er 10.0.0.1 --topic test --duration 10
[*] Flooding broker=10.0.0.1 topic=test for 10.0s (QoS=0) ...
```

mininet@mininet-vm: ~/sdn-iot-project/controllers

```
DROP INSTALLED (60s) on dpid=1 | src_ip=10.0.0.4 | sid=1000002 | signature=MQTT Pub
lish Flood
```

Figure 4.4 System Response

As illustrated in Figure 4.4, when the flood_pub.py script is executed, the attacker host (10.0.0.4) begins flooding the broker (10.0.0.1). Suricata, monitoring this traffic, detects the violation of rule sid:1000002 (10 publishes in 5 seconds) and generates an alert. The custom Ryu controller application, polling the eve.json log, parses this alert, extracts the attacker's source IP (10.0.0.4), and immediately installs a drop rule on the OpenFlow switch (dpid=1). The mitigation is enforced for a 60-second period, effectively neutralizing the attack and protecting the broker. This sequence validates the complete, closed-loop operation of the detection and mitigation system.

4.3.2 Malformed and Logical Error Attacks (RFC & LEC)

Malformed and logical error attacks target MQTT protocol implementations by sending packets that violate the protocol's structural or semantic rules. These attacks exploit insufficient validation in broker software, often leading to crashes, resource exhaustion, or unintended behavior. Based on the taxonomy presented in [31], these attacks are categorized as Lack of Required Fields Checks (RFC) and Missing Logical Error Checks (LEC).

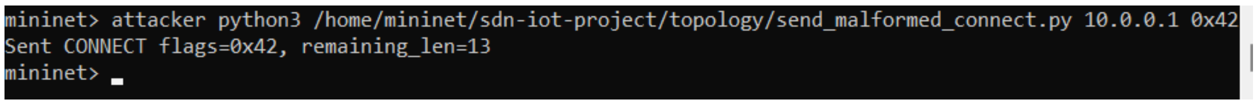
4.3.2.1 Required Field Consistency Attacks (RFC)

The lack of required fields check is a major cause that is found in the recently reported vulnerabilities. These vulnerabilities occur due to an ignorance of the required fields validation during the protocol implementation. So, there should be explicit implementation of the required fields check with respect to packet type. For instance, if a packet contains a user name field, then the relative password field must also be present, as an absence of such a password section will put the implementation at a stake.CVE-2016-9877, where the MQTT broker authenticates a connection request with a valid username but with an omitted password section, or CVE-2017-2893, where the MQTT broker crashes while processing a subscribed packet with no subscription argument.

Detection rule for CVE-2016-9877: *alert tcp any any -> any 1883 (msg:"MQTT RFC: CONNECT password=1 username=0";flow:to_server;content:"|10|"; offset:0; depth:1;byte_test:1,&,0x40,9;byte_test:1,!&,0x80,9;sid:1000005; rev:1;)*

This rule detects malformed MQTT CONNECT packets that violate the required field consistency defined by the MQTT protocol. According to the MQTT specification (version 3.1.1), the Password Flag and Username Flag are part of the Connect Flags field within the CONNECT packet. The specification explicitly states that if the Password Flag is set to 1, the Username Flag must also be set to 1. In other words, a password field is not allowed to exist without a corresponding username field.

Therefore, this rule aims to identify and block CONNECT packets that contain logically inconsistent required fields, which falls under the category of Lack of Required Fields Check (RFC) vulnerabilities.



```
mininet> attacker python3 /home/mininet/sdn-iot-project/topology/send_malformed_connect.py 10.0.0.1 0x42
Sent CONNECT flags=0x42, remaining_len=13
mininet> 
```

Figure 4.5- Malformed Packet Script

Send_malformed_connect.py as shown in figure 4.5 is designed to generate and transmit a deliberately malformed MQTT CONNECT packet directly over a TCP socket. Instead of using a standard MQTT client library (which enforces protocol correctness), the script manually constructs the raw MQTT packet bytes.

The script sets the fixed header to indicate a CONNECT packet (0x10) and assigns a minimal remaining length. Within the variable header, it inserts a crafted Connect Flags value (0x42) that enables the password flag while disabling the username flag. This violates the MQTT protocol's required field consistency rules.

In the crafted attack packet used to trigger this rule, the value 0x42 is used intentionally to manipulate the Connect Flags field.

The Connect Flags byte in an MQTT CONNECT packet is structured as follows (from most significant bit to least significant bit):

Figure 4.1 MQTT CONNECT Flag Byte

Bit	Meaning
Bit 7	Username Flag
Bit 6	Password Flag Bit 5
Bit 5	Will Retain
Bit 4–3	Will QoS
Bit 2	Will Flag
Bit 1–0	Reserved

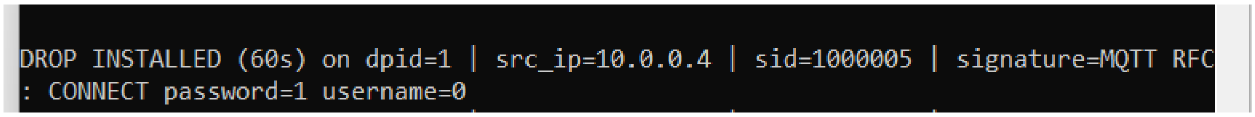
The hexadecimal value 0x42 corresponds to the binary value:

0x42 = 0100 0010

From this binary representation:

- Bit 6 (Password Flag) = 1
- Bit 7 (Username Flag) = 0

This combination directly violates the MQTT specification, which requires the username flag to be set whenever the password flag is set. All other flags remain unset, ensuring that the packet is malformed only due to this specific logical inconsistency.



```
DROP INSTALLED (60s) on dpid=1 | src_ip=10.0.0.4 | sid=1000005 | signature=MQTT RFC
: CONNECT password=1 username=0
```

Figure 4.6- Ryu's Response

As shown in Figure 4.5 Ryu successfully adds a flow rule in response to the triggered rule.

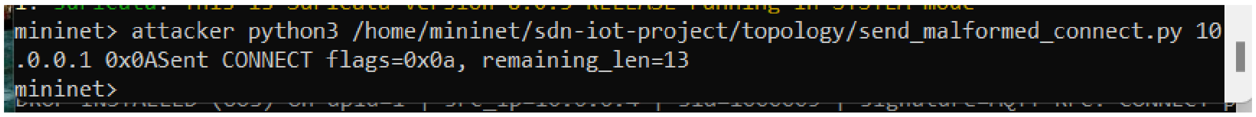
This approach allows controlled testing of the IDS rule by ensuring that the malformed condition originates solely from the crafted flags, thereby validating the effectiveness of the detection rule and the controller's mitigation mechanism

for checking required fields:

alert tcp any any -> any 1883 (msg:"MQTT RFC: CONNECT WillQoS>0 while WillFlag=0";flow:to_server;content:"|10|"; offset:0; depth:1;byte_test:1,!&,0x04,9;byte_test:1,&,0x18,9;sid:1000006; rev:1;)

This rule detects malformed MQTT CONNECT packets in which Will-related fields are set despite the Will Flag being disabled. According to the MQTT 3.1.1 specification, Will QoS and Will Retain fields must be zero if the Will Flag is not enabled. Violating this constraint indicates a malformed packet and reflects missing logical validation in protocol implementations. Such malformed packets have been associated with denial-of-service vulnerabilities in several MQTT implementations.

To test this rule, a malformed MQTT CONNECT packet was manually crafted using a raw TCP socket. the value 0x0A was deliberately chosen to manipulate the Will QoS field while keeping the Will Flag disabled. According to the MQTT 3.1.1 specification, the Will QoS field occupies bits 3 and 4 of the Connect Flags byte, whereas the Will Flag corresponds to bit 2. The hexadecimal value 0x0A (binary 00001010) sets the Will QoS bits to a non-zero value while leaving the Will Flag cleared.

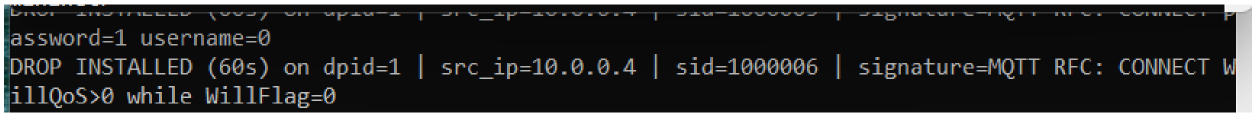


```
mininet> attacker python3 /home/mininet/sdn-iot-project/topology/send_malformed_connect.py 10
.0.0.1 0x0ASent CONNECT flags=0x0a, remaining_len=13
mininet>
```

Figure 4.7- Will QoS Malformed Packet

This specific bit combination violates the mandatory constraint defined in the MQTT specification, which states that if the Will Flag is set to zero, both the Will QoS and Will Retain fields must also be zero. By using 0x0A, the test packet introduces a precise logical inconsistency without modifying unrelated flags, ensuring that the generated alert is solely due to the missing logical validation of Will-related fields.

Upon transmission, Suricata successfully detected the protocol violation and generated an alert, which was then consumed by the SDN controller to install a temporary drop rule against the source IP :

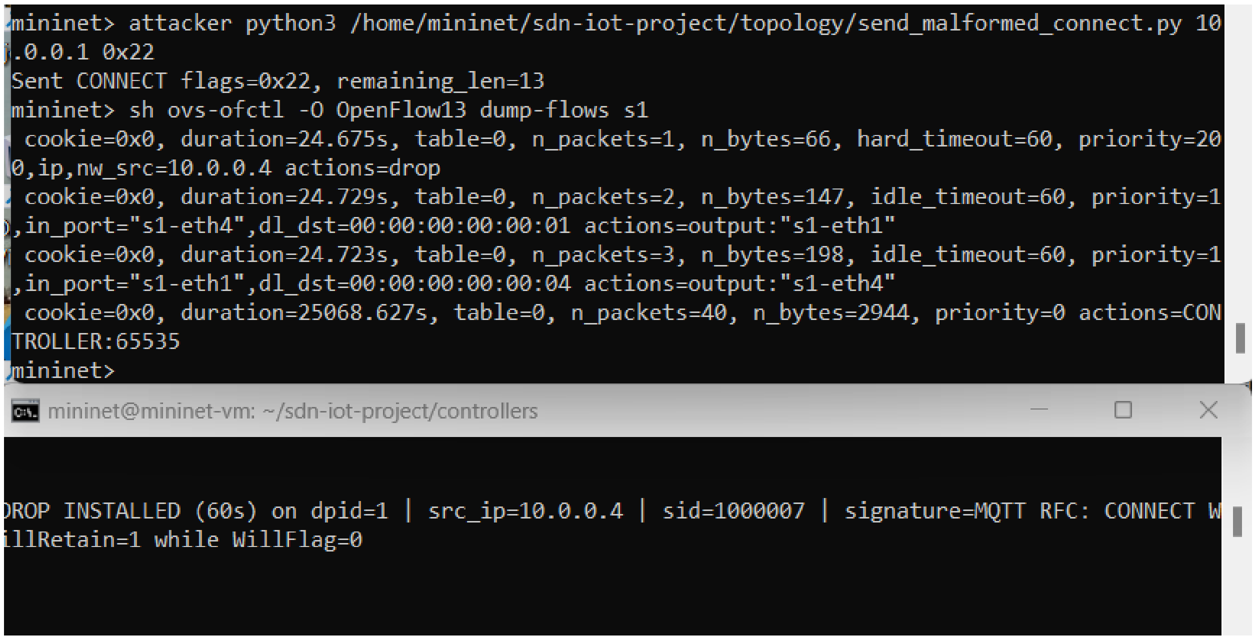


```
assword=1 username=0
DROP INSTALLED (60s) on dpid=1 | src_ip=10.0.0.4 | sid=1000006 | signature=MQTT RFC: CONNECT W
illQoS>0 while WillFlag=0
```

Figure 4.8- Triggered Rule

alert tcp any any -> any 1883 (msg:"MQTT RFC: CONNECT WillRetain=1 while WillFlag=0";flow:to_server;content:"|10|"; offset:0; depth:1;byte_test:1,!&,0x04,9;byte_test:1,&,0x20,9;sid:1000007; rev:1;)

This rule detects malformed MQTT CONNECT packets in which the Will Retain flag is enabled while the Will Flag is disabled. According to the MQTT 3.1.1 specification, the Will Retain field is only meaningful when a Will message is present; therefore, it must be set to zero if the Will Flag is cleared.



```
mininet> attacker python3 /home/mininet/sdn-iot-project/topology/send_malformed_connect.py 10
.0.0.1 0x22
Sent CONNECT flags=0x22, remaining_len=13
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
 cookie=0x0, duration=24.675s, table=0, n_packets=1, n_bytes=66, hard_timeout=60, priority=20
0,ip,nw_src=10.0.0.4 actions=drop
 cookie=0x0, duration=24.729s, table=0, n_packets=2, n_bytes=147, idle_timeout=60, priority=1
,in_port="s1-eth4",dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=24.723s, table=0, n_packets=3, n_bytes=198, idle_timeout=60, priority=1
,in_port="s1-eth1",dl_dst=00:00:00:00:00:04 actions=output:"s1-eth4"
 cookie=0x0, duration=25068.627s, table=0, n_packets=40, n_bytes=2944, priority=0 actions=CON
TROLLER:65535
mininet>
```

mininet@mininet-vm: ~/sdn-iot-project/controllers

```
ROP INSTALLED (60s) on dpid=1 | src_ip=10.0.0.4 | sid=1000007 | signature=MQTT RFC: CONNECT W
illRetain=1 while WillFlag=0
```

Figure 4.9- Attack and Response

As seen in Figure 4.9 this rule was tested by generating a crafted MQTT CONNECT control packet using a raw TCP socket, since standard MQTT client libraries do not allow constructing non-compliant CONNECT flags. The script send_malformed_connect.py builds a minimal CONNECT packet with a fixed header (0x10), a valid protocol name (“MQTT”), protocol level 4 (MQTT 3.1.1), a Keep-Alive value of 10 seconds (0x000A), and a minimal ClientID payload (“a”). To trigger the violation, the CONNECT Flags field was set to 0x22, which intentionally enables the Will Retain bit while keeping the Will Flag cleared, creating a logical inconsistency that violates the MQTT 3.1.1 CONNECT-flags constraints. The crafted packet was sent to the broker over TCP port 1883, causing Suricata to raise the corresponding alert; the SDN controller then reacted by installing a temporary drop flow for the attacker’s source IP for 60 seconds.

The command in the previous figure (sh ovs-ofctl -O OpenFlow13 dump-flows s1) is used to inspect the forwarding rules currently installed on an Open vSwitch (OVS) switch. Specifically, it instructs the system to:

- Use the ovs-ofctl utility to communicate with the switch.
- Specify OpenFlow version 1.3 (-O OpenFlow13) to match the protocol used by the SDN controller.
- Retrieve and display (dump) all flow entries installed in the flow table of switch s1.

This command is typically used for debugging and verification purposes, allowing the operator to confirm that the SDN controller has successfully installed forwarding or drop rules (e.g., temporary mitigation rules triggered by IDS alerts).

Malformed SUBSCRIBE

In MQTT 3.1.1, the SUBSCRIBE control packet has a strictly defined fixed header format. The low-order four bits of the fixed header are reserved and must be set to a specific value (0x2). Any deviation from this required value renders the packet non-compliant with the protocol specification. Such malformed SUBSCRIBE packets may bypass insufficient validation logic in MQTT implementations, potentially leading to parsing errors or denial-of-service conditions. Consequently, enforcing this constraint at the IDS level provides an effective mechanism for identifying malformed MQTT traffic.

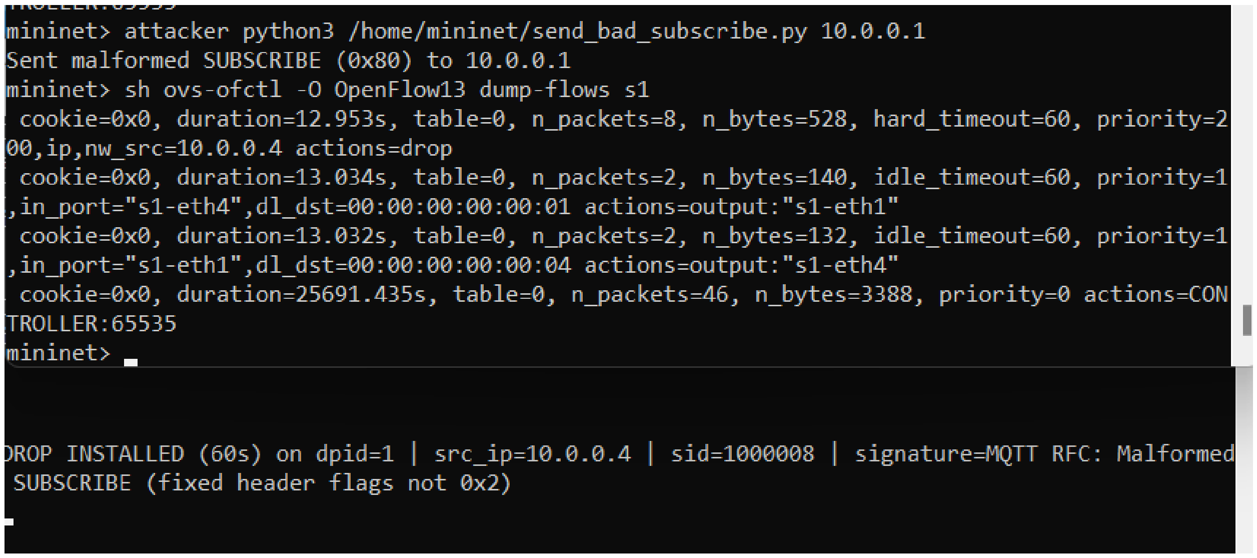
alert tcp any any -> any 1883 (msg:"MQTT RFC: Malformed SUBSCRIBE (fixed header flags not 0x2)";flow:to_server;content:"|80|"; offset:0; depth:1;content:!"|82|"; offset:0; depth:1;sid:1000008; rev:1;)

The IDS rule detects this condition by:

• Identifying packets whose packet type corresponds to SUBSCRIBE.

• Verifying whether the fixed header byte differs from the required value 0x82.

• Raising an alert whenever a SUBSCRIBE packet is observed with invalid fixed header flags.



```
mininet> attacker python3 /home/mininet/send_bad_subscribe.py 10.0.0.1
Sent malformed SUBSCRIBE (0x80) to 10.0.0.1
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
 cookie=0x0, duration=12.953s, table=0, n_packets=8, n_bytes=528, hard_timeout=60, priority=2
00,ip,nw_src=10.0.0.4 actions=drop
 cookie=0x0, duration=13.034s, table=0, n_packets=2, n_bytes=140, idle_timeout=60, priority=1
,in_port="s1-eth4",dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=13.032s, table=0, n_packets=2, n_bytes=132, idle_timeout=60, priority=1
,in_port="s1-eth1",dl_dst=00:00:00:00:00:04 actions=output:"s1-eth4"
 cookie=0x0, duration=25691.435s, table=0, n_packets=46, n_bytes=3388, priority=0 actions=CON
TROLLER:65535
mininet>
```

```
DROP INSTALLED (60s) on dpid=1 | src_ip=10.0.0.4 | sid=1000008 | signature=MQTT RFC: Malformed
 SUBSCRIBE (fixed header flags not 0x2)
```

Figure 4.10- Malformed Subscribe Attack and Response

This approach does not attempt to enumerate all malformed combinations but instead enforces the mandatory constraint defined by the MQTT specification.

The analysis presented in [31] demonstrates that a significant class of MQTT vulnerabilities originates from the absence of strict validation of mandatory fields and their logical relationships during packet processing. Although the MQTT specification precisely defines which fields must be present and how control flags must be combined for each message type, many real-world implementations fail to consistently enforce these requirements. As a result, attackers can craft syntactically valid but semantically incorrect packets that exploit parsing logic and trigger abnormal behavior, including denial-of-service conditions.

The proposed RFC-based detection rules do not attempt to enumerate all possible malformed combinations. Instead, they focus on representative violations of mandatory field dependencies defined by the MQTT specification. This approach aligns with the findings of the analyzed paper, which emphasizes that missing required field checks constitute a structural weakness at the implementation level rather than an issue of protocol design. Consequently, enforcing these checks at the IDS level provides an effective and lightweight defensive layer against a broad class of malformed MQTT attacks.

4.3.2.2 Missing Logical Errors Checks (LEC)

This issue arises due to the lack of logical errors check in the packet data and lack of their identification in implementations. For instance, in CVE-2020-13849, the MQTT server implementation is vulnerable to a denial of service, and it is unable to establish new connections due to a lack of logical check on the Keep-Alive value sent by the client. The MQTT server extends a client connection timeout by 1.5 times the time asked by the client. An attacker exploited this functionality by sending a larger timeout request, keeping the server resources busy, which ultimately caused a DoS attack [31].

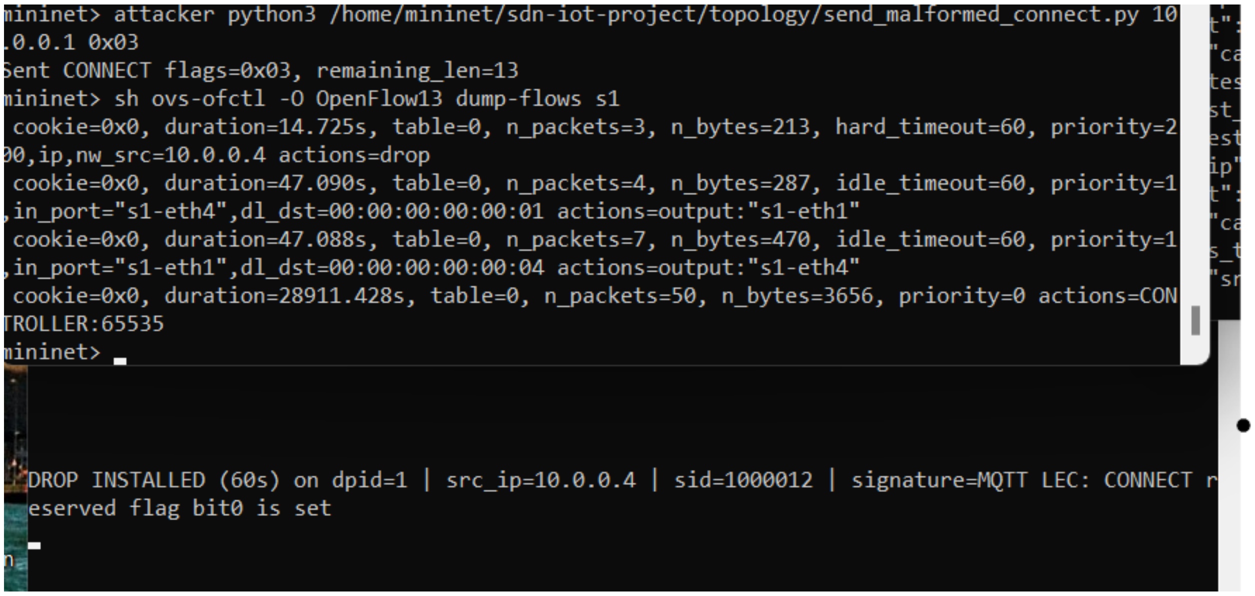
Similarly, in CVE-2019-11778, the MQTT server crashes while processing a packet having the value of “will delay interval” greater than the value of “session expiry interval”. According to the MQTT protocol specifications, the “will delay interval” should be less than or equal to the “session expiry interval”. Nevertheless, due to the missing logical errors check in the MQTT implementation; an attacker could crash the MQTT server by sending a crafted packet having the value of “will delay interval” greater than the value of “session expiry interval”.

Detection Rule for CVE-2019-11778:

alert tcp any any -> any 1883 (msg:"MQTT LEC: CONNECT reserved flag bit0 is set";flow:to_server;content:"|10|"; offset:0; depth:1;byte_test:1,&,0x01,9;sid:1000012; rev:1;)

This rule alerts when an MQTT CONNECT control packet contains a non-zero value in the reserved flag bit of the CONNECT flags field. According to the MQTT 3.1.1 specification, the least significant bit of the CONNECT flags byte is reserved and MUST be set to zero. Any deviation from this requirement indicates a malformed packet that violates the logical constraints of the protocol. Failure to validate this condition may cause unexpected behavior in MQTT implementations. This rule therefore targets logical inconsistencies where protocol fields are present but contain invalid values that are explicitly disallowed by the specification.

To validate this rule, a crafted MQTT CONNECT packet was generated with the reserved flag bit set to one. The packet was transmitted from an attacking host toward the MQTT broker, successfully triggering the rule and resulting in the installation of a temporary drop rule by the SDN controller.



```
ininet> attacker python3 /home/mininet/sdn-iot-project/topology/send_malformed_connect.py 10
.0.0.1 0x03
Sent CONNECT flags=0x03, remaining_len=13
ininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
 cookie=0x0, duration=14.725s, table=0, n_packets=3, n_bytes=213, hard_timeout=60, priority=2
00,ip,nw_src=10.0.0.4 actions=drop
 cookie=0x0, duration=47.090s, table=0, n_packets=4, n_bytes=287, idle_timeout=60, priority=1
,in_port="s1-eth4",dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=47.088s, table=0, n_packets=7, n_bytes=470, idle_timeout=60, priority=1
,in_port="s1-eth1",dl_dst=00:00:00:00:00:04 actions=output:"s1-eth4"
 cookie=0x0, duration=28911.428s, table=0, n_packets=50, n_bytes=3656, priority=0 actions=CON
TROLLER:65535
ininet>
```

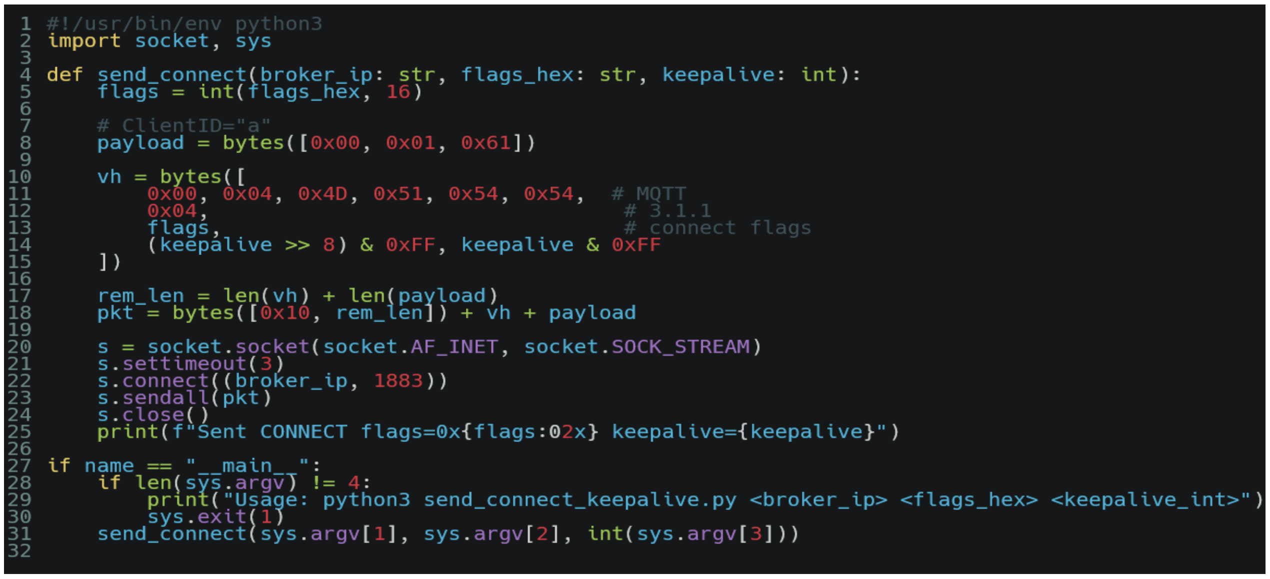
```
DROP INSTALLED (60s) on dpid=1 | src_ip=10.0.0.4 | sid=1000012 | signature=MQTT LEC: CONNECT r
eserved flag bit0 is set
```

Figure 4.11- Malformed Will Delay Interval Attack and Response

Detection Rule for CVE-2020-13849:

alert tcp any any -> any 1883 (msg:"MQTT LEC: CONNECT KeepAlive unusually high (possible resource abuse)";flow:to_server;content:"|10|"; offset:0; depth:1;byte_test:2,>,1200,10;sid:1000013; rev:1;)

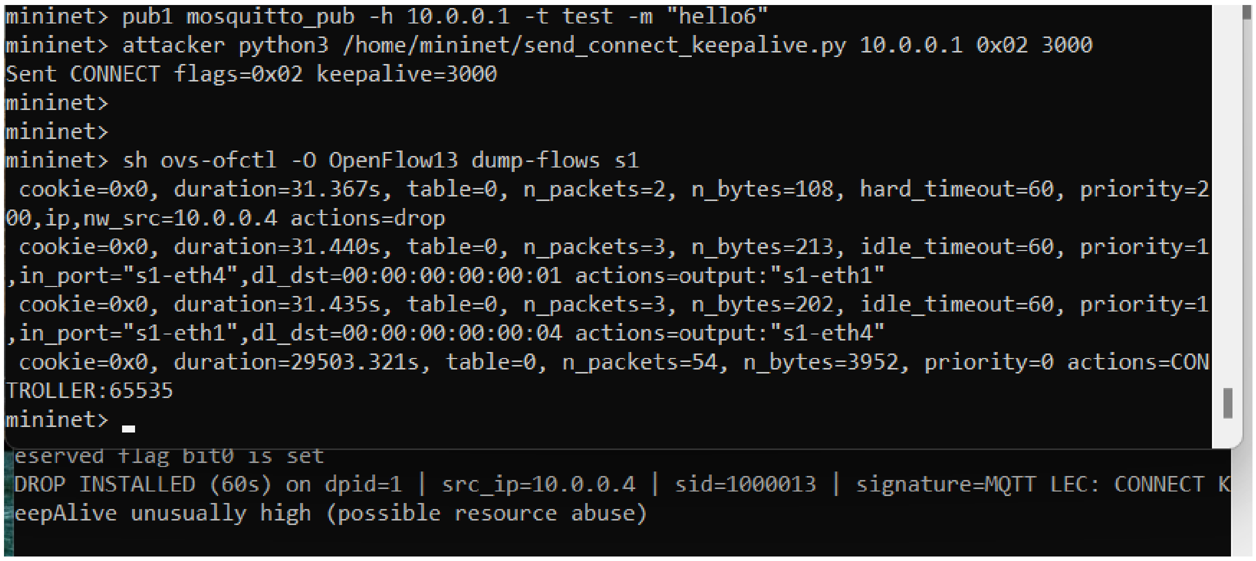
This rule alerts when an MQTT CONNECT packet carries an unusually large Keep-Alive value that exceeds a predefined threshold. The Keep-Alive parameter is used by MQTT clients to indicate the maximum allowed interval between control packets; however, excessively large values may be abused to keep server-side resources allocated for prolonged periods. Such behavior represents a logical misuse of a valid protocol field rather than a structural violation. Several reported vulnerabilities have shown that insufficient validation of Keep-Alive values can lead to resource exhaustion and denial-of-service conditions in MQTT brokers.



```python
#!/usr/bin/env python3
import socket, sys

def send_connect(broker_ip: str, flags_hex: str, keepalive: int):
    flags = int(flags_hex, 16)

    # ClientID="a"
    payload = bytes([0x00, 0x01, 0x61])

    vh = bytes([
        0x00, 0x04, 0x4D, 0x51, 0x54, 0x54,  # MQTT
        0x04,                                 # 3.1.1
        flags,                                # connect flags
        (keepalive >> 8) & 0xFF, keepalive & 0xFF
    ])

    rem_len = len(vh) + len(payload)
    pkt = bytes([0x10, rem_len]) + vh + payload

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(3)
    s.connect((broker_ip, 1883))
    s.sendall(pkt)
    s.close()
    print(f"Sent CONNECT flags=0x{flags:02x} keepalive={keepalive}")

if name == "__main__":
    if len(sys.argv) != 4:
        print("Usage: python3 send_connect_keepalive.py <broker_ip> <flags_hex> <keepalive_int>")
        sys.exit(1)
    send_connect(sys.argv[1], sys.argv[2], int(sys.argv[3]))
```

Figure 4.12- Malformed Keep Alive Script

This rule was tested by transmitting a CONNECT packet with an intentionally inflated Keep-Alive value from an attacking host (Figure 4.12). Upon detection, the intrusion detection system raised an alert, which triggered the SDN controller to dynamically block traffic originating from the attacker for a limited duration (Figure 4.13).



```
mininet> pub1 mosquitto_pub -h 10.0.0.1 -t test -m "hello6"
mininet> attacker python3 /home/mininet/send_connect_keepalive.py 10.0.0.1 0x02 3000
Sent CONNECT flags=0x02 keepalive=3000
mininet>
mininet>
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
 cookie=0x0, duration=31.367s, table=0, n_packets=2, n_bytes=108, hard_timeout=60, priority=2
00,ip,nw_src=10.0.0.4 actions=drop
 cookie=0x0, duration=31.440s, table=0, n_packets=3, n_bytes=213, idle_timeout=60, priority=1
,in_port="s1-eth4",dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=31.435s, table=0, n_packets=3, n_bytes=202, idle_timeout=60, priority=1
,in_port="s1-eth1",dl_dst=00:00:00:00:00:04 actions=output:"s1-eth4"
 cookie=0x0, duration=29503.321s, table=0, n_packets=54, n_bytes=3952, priority=0 actions=CON
TROLLER:65535
mininet> 
eserved flag bit0 is set
DROP INSTALLED (60s) on dpid=1 | src_ip=10.0.0.4 | sid=1000013 | signature=MQTT LEC: CONNECT K
eepAlive unusually high (possible resource abuse)
```

Figure 4.13- Keep Alive Attack and Response

4.3.2.3 Brute-Force

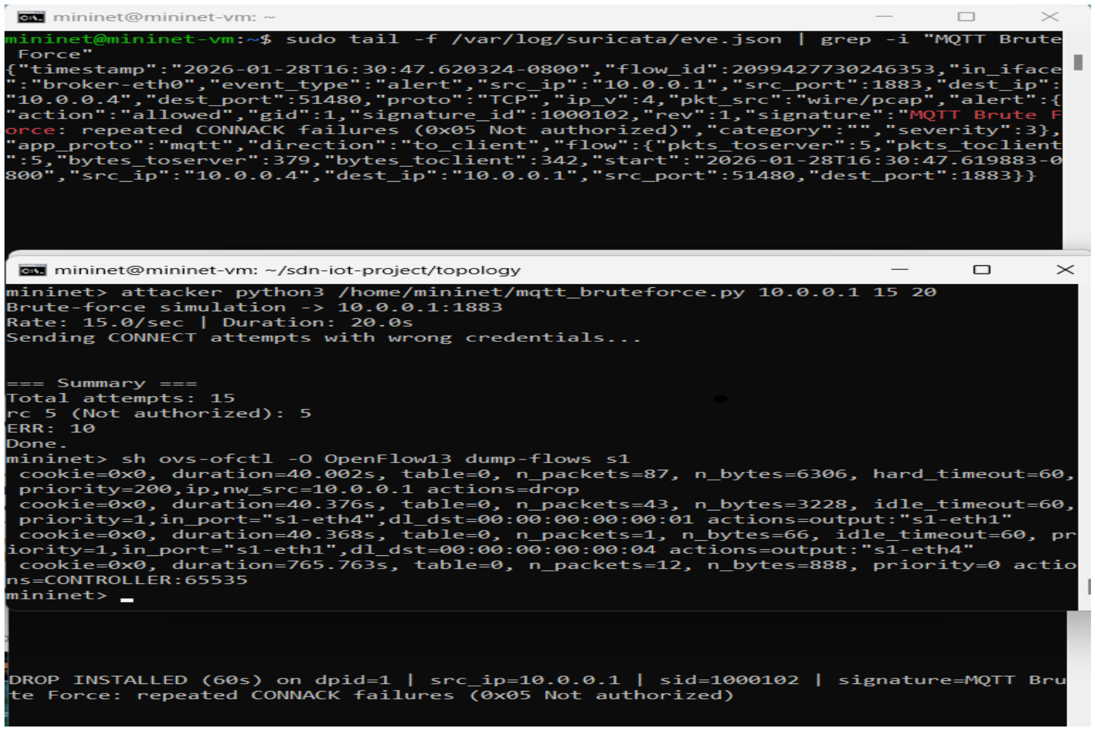
Brute force attacks against MQTT brokers attempt to gain unauthorized access by systematically trying username/password combinations. The MQTT protocol provides a clear failure signal: when authentication fails, the broker responds with a CONNACK packet containing a Return Code of 0x05 (Not authorized). An attacker performing rapid, automated login attempts will generate a high frequency of these 0x05 error responses from the broker to a single client or IP address. Monitoring the rate of these authentication failure packets enables the detection of credential stuffing and brute force attempts without inspecting login payloads.

Therefore, MQTT brokers that do not enforce strong authentication limits or lockout policies are highly susceptible.

alert tcp any 1883 -> any any (msg:"MQTT Brute Force: repeated CONNACK failures (0x05 Not authorized)";flow:from_server,established;content:"|20 02 00 05|"; offset:0; depth:4;threshold:type threshold, track by_dst, count 10, seconds 30;sid:1000102; rev:1;)

This rule detects brute force MQTT login attempts by alerting when a broker sends more than 10 "Not Authorized" CONNACK responses to a single client within 30 seconds.

As seen in Figure 4.14 brute-force script was executed and the alert was triggered, therefore, the controller successfully responded with adding a new flow rule to the switch.



```
mininet@mininet-vm: ~
mininet@mininet-vm:~$ sudo tail -f /var/log/suricata/eve.json | grep -i "MQTT Brute
 Force"
{"timestamp":"2026-01-28T16:30:47.620324-0800","flow_id":2099427730246353,"in_iface
":"broker-eth0","event_type":"alert","src_ip":"10.0.0.1","src_port":1883,"dest_ip":
"10.0.0.4","dest_port":51480,"proto":"TCP","ip_v":4,"pkt_src":"wire/pcap","alert":{
"action":"allowed","gid":1,"signature_id":1000102,"rev":1,"signature":"MQTT Brute F
orce: repeated CONNACK failures (0x05 Not authorized)","category":"","severity":3},
"app_proto":"mqtt","direction":"to_client","flow":{"pkts_toserver":5,"pkts_toclient
":5,"bytes_toserver":379,"bytes_toclient":342,"start":"2026-01-28T16:30:47.619883-0
800","src_ip":"10.0.0.4","dest_ip":"10.0.0.1","src_port":51480,"dest_port":1883}}
```

```
mininet@mininet-vm: ~/sdn-iot-project/topology
mininet> attacker python3 /home/mininet/mqtt_bruteforce.py 10.0.0.1 15 20
Brute-force simulation -> 10.0.0.1:1883
Rate: 15.0/sec | Duration: 20.0s
Sending CONNECT attempts with wrong credentials...


=== Summary ===
Total attempts: 15
rc 5 (Not authorized): 5
ERR: 10
Done.
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
 cookie=0x0, duration=40.002s, table=0, n_packets=87, n_bytes=6306, hard_timeout=60,
 priority=200,ip,nw_src=10.0.0.1 actions=drop
 cookie=0x0, duration=40.376s, table=0, n_packets=43, n_bytes=3228, idle_timeout=60,
 priority=1,in_port="s1-eth4",dl_dst=00:00:00:00:00:01 actions=output:"s1-eth1"
 cookie=0x0, duration=40.368s, table=0, n_packets=1, n_bytes=66, idle_timeout=60, pr
iority=1,in_port="s1-eth1",dl_dst=00:00:00:00:00:04 actions=output:"s1-eth4"
 cookie=0x0, duration=765.763s, table=0, n_packets=12, n_bytes=888, priority=0 actio
ns=CONTROLLER:65535
mininet> _
```

```
DROP INSTALLED (60s) on dpid=1 | src_ip=10.0.0.1 | sid=1000102 | signature=MQTT Bru
te Force: repeated CONNACK failures (0x05 Not authorized)
```

Figure 4.14- Brute-Force Attack and Response

4.4 Challenges

Previous research and current developments in intrusion detection heavily focus on machine learning (ML) and deep learning (DL)-based IDS solutions for IoT protocols such as MQTT. Following this trend, our initial approach was to implement an ML-based IDS trained on the MQTTset dataset, evaluating performance using established metrics such as accuracy, recall, precision, and F1 score. Indeed, when trained and evaluated offline under controlled conditions, the model achieved high detection performance, with excellent accuracy and F1 values on the test set, suggesting strong classification capabilities.

However, when deploying the trained model in a real-time operational environment, its performance deteriorated significantly, with a high false positive rate that renders the model unsuitable for practical use. This behavior reflects a common challenge in ML-based IDS research: strong offline accuracy does not necessarily translate to reliable real-time detection. One key factor underlying this issue is the nature of MQTTset itself — it is fundamentally packet-based, with features extracted from individual packet attributes without capturing the broader temporal, flow or session context that often distinguishes benign from malicious behavior in network traffic. This limitation aligns with recent empirical findings showing that models trained solely on individual packet features can produce misleadingly high detection rates but fail to generalize effectively due to lack of interaction information and temporal context [33].

In contrast, many real-world attacks — especially those most relevant in MQTT environments such as DoS, publish flooding, and SlowITe — are characterized by patterns over time (e.g., rate, frequency, duration) rather than isolated packet characteristics. The absence of flow statistics or session-based descriptors in the dataset means that ML models may lack essential information needed for robust discrimination between normal and attack traffic in real time. Furthermore, the performance drop in deployment reflects broader ML IDS challenges, such as model generalization and concept drift, where the statistical properties of real traffic differ from the training data distribution.

Therefore, although MQTTset is a valuable resource for initial model development, our experience and emerging research suggest that extensive feature engineering, inclusion of flow and session features, and hybrid modeling approaches are necessary to achieve reliable real-time performance for ML-based MQTT intrusion detection.

4.5 Technical Recommendations

1. More approximation to real world deployments:

In the experimental testbed, the SDN controller consumes Suricata alerts from local log files for simplicity and ease of integration, as both components run on the same host environment. In real-world deployments, however, Suricata alerts are typically exported through dedicated interfaces such as Unix sockets, syslog streams, or event pipelines, allowing the controller or security orchestration systems to receive alerts in real time and enforce mitigation policies without relying on file-based communication.

2. Controller Performance and Resource Optimization

Since the SDN controller plays a critical role in traffic mitigation, excessive alert handling or frequent rule installation may affect controller responsiveness. It is recommended to introduce rate-limiting mechanisms for alert processing and to batch mitigation actions when possible. Additionally, monitoring controller CPU and memory usage during attack scenarios can help optimize performance and prevent controller overload.

3. Rule Management and Prioritization

The coexistence of built-in IDS rules and custom MQTT-specific rules can lead to overlapping alerts. To address this, a rule prioritization strategy should be applied, ensuring that protocol-specific rules are evaluated in the intended order. Periodic validation of rule effectiveness and false-positive behavior is also recommended to maintain reliable detection.

4. Secure Communication Between Components

In a real deployment, communication between IDS sensors, controllers, and network devices should be protected against interception and tampering. Using encrypted channels (e.g., TLS) and authentication mechanisms between components is recommended to prevent attackers from injecting fake alerts or disabling mitigation actions

4.6 Future Work

Although the current implementation demonstrates the feasibility of combining signature-based intrusion detection with SDN-enabled mitigation to protect MQTT traffic, several avenues exist to enhance robustness, adaptability, and real-world applicability.

1. Custom Dataset Generation and ML/DL Model Training

One major limitation encountered in this work was the inability of machine learning models trained on the MQTTset dataset to perform reliably in real-time deployment, primarily due to the dataset's packet-based nature and lack of temporal or flow context. A significant future direction is to develop a custom dataset that reflects the specific network topology and traffic patterns of the SDN–MQTT environment, capturing both individual packet attributes and session/flow-level statistics (e.g., inter-arrival times, throughput, burstiness). This custom dataset would enable training and evaluation of both traditional ML algorithms and modern deep learning models (e.g., recurrent neural networks or graph neural networks) that can learn temporal and spatial dependencies inherent in attack behaviors such as flooding, publish storming, and protocol misuse. By tailoring the dataset to the deployment scenario and enriching it with contextual features, future work can significantly enhance model generalization and reduce false positives, thereby bridging the gap between offline performance and live deployment.

2. Hybrid Detection Frameworks

Another valuable extension is to investigate hybrid IDS architectures that combine signature-based detection with anomaly-based learning mechanisms. Signature-based rules, such as those implemented in this project, excel at identifying known attack patterns; anomaly detection models, on the other hand, can flag previously unseen or evolving threats. Integrating both approaches within the SDN controller's decision pipeline can enable more comprehensive coverage, reduce dependence on rule maintenance, and adapt to zero-day behaviors while maintaining low false positive rates.

3. Protecting the SDN Control Plane

While this project focuses on protecting MQTT traffic in the data plane, future research should consider the security of the SDN control plane itself. The SDN controller and the OpenFlow channel represent critical assets that, if compromised, could undermine the entire detection and mitigation framework.

Future work can explore secure communication protocols (e.g., TLS/SSL for controller–switch channels), controller hardening techniques, and intrusion detection models specifically tailored to detect anomalies in control-plane messages.

4. Real-World Deployment and Scalability Evaluation

The current system was evaluated within a Mininet emulation environment. Future work should extend evaluation to real hardware testbeds or edge computing environments typical of IoT deployments. This would allow evaluation of performance under realistic network conditions, latency constraints, and resource limitations, as well as investigation of scalability under larger topologies and higher traffic loads.

5. Automated Rule Generation and Updating

As both MQTT protocol usage and attack techniques evolve, maintaining effective signature rule sets becomes labor-intensive. Future research could explore automated rule generation using machine learning techniques that derive discriminative patterns from labeled traffic or unsupervised anomaly clusters. Such an approach could help keep the detection knowledge base up-to-date with minimal human intervention.

6. Context-Aware Mitigation Policies

Finally, future systems can incorporate contextual information — such as client identity, device type, or traffic class — into mitigation decisions rather than applying uniform drop actions.

For instance, mitigation policies might throttle or re-route suspicious flows, notify network administrators, or quarantine devices based on severity and context, improving the balance between security and availability.

5.7 Conclusion

This project presents a concrete, reproducible blueprint for enhancing IoT network security through SDN. It proves that a well-integrated signature-based IDPS can provide immediate, reliable protection against known threats while establishing the infrastructural groundwork for next-generation, intelligent security systems capable of adapting to the evolving IoT threat landscape.

References

1. Mishra, S. R., Shanmugam, B., Yeo, K. C., & Thennadil, S. (2025). SDN-Enabled IoT Security Frameworks—A Review of Existing Challenges. *Technologies*, *13*(3), 121.
2. Rahman, M. M., Shakil, S. A., & Mustakim, M. R. (2025). A survey on intrusion detection system in IoT networks. *Cyber Security and Applications, 3,* 100082.
3. Vaccari, I., Chiola, G., Aiello, M., Mongelli, M., & Cambiaso, E. (2020). MQTTset, a New Dataset for Machine Learning Techniques on MQTT. *Sensors*, *20*(22), 6578.
4. Sebestyen, H., Popescu, D. E., & Zmaranda, R. D. (2025). *A literature review on security in the Internet of Things: Identifying and analysing critical categories*. Computers, 14(2), 61.
5. Al Hayajneh, A., Bhuiyan, M. Z. A., & McAndrew, I. (2020). *Improving Internet of Things (IoT) security with software-defined networking (SDN)*.
6. Berhili, M., Chaieb, O., & Benabdellah, M. (2024). *Intrusion detection systems in IoT based on machine learning: A state of the art*. Procedia Computer Science, 251, 99–107.
7. Ataa, M. S., Sanad, E. E., & El-khoribi, R. A. (2024). *Intrusion detection in software defined network using deep learning approaches*. Scientific Reports, 14, 29159.
8. Karmakar, K. K., Varadharajan, V., Nepal, S., & Tupakula, U. (2021). *SDN-enabled secure IoT architecture*. IEEE Internet of Things Journal, 8(8).
9. Dhirar, H., & Hamad, A. (2025). *Comparative evaluation of a novel IDS dataset for SDN-IoT using deep learning models against InSDN, BoT-IoT, and ToN-IoT*. *Measurement: Digitalization, 4*, 100015.
10. Ahmed, Md. Rayhan; Islam, salekul; Shatabda, Swakkhar; Islam, A. K. M. Muzahidul; Robin, Md. Towhidul Islam (2021): *Intrusion Detection System in Software-Defined Networks Using Machine Learning and Deep Learning Techniques –A Comprehensive Survey.* TechRxiv. Preprint.
11. Ferrão, T., Manene, F., & Ajibesin, A. A. (2023). *Multi-Attack Intrusion Detection System for Software-Defined Internet of Things Network*. Computers, Materials & Continua, 75(3), 4985–5007.
12. Owusu, A. I., & Nayak, A. (2020). *An intelligent traffic classification in SDN-IoT: A machine learning approach*. In 2020 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom). IEEE.
13. Bedhief, I., Kassar, M., & Aguili, T. (2023). *Empowering SDN-Docker based architecture for Internet of Things heterogeneity*. Journal of Network and Systems Management, 31, 14.
14. Ali, M. A., & Al-Sharafi, S. A. H. (2025). *Intrusion detection in IoT networks using machine learning and deep learning approaches for MitM attack mitigation*. Discover Internet of Things, 5, 48.
15. Khan, M. A., Khan, M. A., Jan, S. U., Ahmad, J., Jamal, S. S., Shah, A. A., Pitropakis, N., & Buchanan, W. J. (2021). *A deep learning-based intrusion detection system for MQTT enabled IoT*. Sensors, 21(21), 7016.
16. Logeswari G., Bose S., Anitha T. (2023). *An Intrusion Detection System for SDN Using Machine Learning*. *Intelligent Automation & Soft Computing*, 35(1), 867–880.
17. Chaganti R., Suliman W., Ravi V., Dua A. (2023). *Deep Learning Approach for SDN-Enabled Intrusion Detection System in IoT Networks*. *Information*, 14(1):41.

18. Cherian M.M., Varma S.L. (2022). *Mitigation of DDOS and MiTM Attacks using Belief Based Secure Correlation in SDN-IoT Networks*. *IJCNIS*, 14(1), 52–68.
19. Nandyala, C. S., & Kim, H.-K. (2016). *Green IoT agriculture and healthcare application (GAHA). International Journal of Smart Home,* 10(4), 289–300.
20. Maurya, S. K., Pal, O. P., & Sarvakar, K. (2024). Layered Architecture of IoT. In S. Singh, S. Tanwar, R. Jadeja, S. Singh, & Z. Polkowski (Eds.), *Secure and Intelligent IoT-Enabled Smart Cities* (pp. 164-194). IGI Global Scientific Publishing.
21. Gardašević, G., Veletić, M., Maletić, N., Vasiljević, D., Radusinović, I., Tomović, S., & Radonjić, M. (2017). *The IoT architectural framework, design issues and application domains. Wireless Personal Communications,* 92(1), 127–148.
22. M. S. Virat, S. M. Bindu, B. Aishwarya, B. N.Dhanush, and M. R. Kounte, "Security and Privacy Challenges in Internet of Things," Proc. 2nd Int.Conf. Trends Electron. Informatics, ICOEI 2018, pp.454–460, 2018.
23. Junejo AK, Breza M, McCann JA. Threat Modeling for Communication Security of IoT-Enabled Digital Logistics. Sensors (Basel). 2023 Nov 29;23(23):9500.
24. Albulayhi K, Smadi AA, Sheldon FT, Abercrombie RK. IoT Intrusion Detection Taxonomy, Reference Architecture, and Analyses. *Sensors*. 2021; 21(19):6432
25. Vaccari I, Aiello M, Cambiaso E. SlowITe, a Novel Denial of Service Attack Affecting MQTT. Sensors (Basel). 2020 May 21;20(10):2932.
26. De La Cadena, L., Loachamin, J., Gamboa, D., Guerrero, G., Quishpe, S., Nacimba, E. (2026). Man-in-the-Middle Attacks on IoT Devices: Message Manipulation and Vulnerabilities in the MQTT Protocol An Experimental Case Study. In: Valencia-Garcia, R., *et al.* Technologies and Innovation. CITI 2025. Communications in Computer and Information Science, vol 2776. Springer, Cham.
27. Yakupov, D. R. (2022). Overview and comparison of protocols Internet of Things: MQTT and AMQP. *International Journal of Open Information Technologies*, 10(5), 1–9.
28. Radwan, Nael & Alves-Foss, Jim. (2024). MQTT in Focus: Understanding the Protocol and Its Recent Advancements. 18. 1-14.
29. Naik, N. (2017). *Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP*. 1–7.
30. Song W, Beshley M, Przystupa K, Beshley H, Kochan O, Pryslupskyi A, Pieniak D, Su J. A Software Deep Packet Inspection System for Network Traffic Analysis and Anomaly Detection. Sensors (Basel). 2020 Mar 14;20(6):1637.
31. Husnain, M., Hayat, K., Cambiaso, E., Fayyaz, U. U., Mongelli, M., Akram, H., Ghazanfar Abbas, S., & Shah, G. A. (2022). Preventing MQTT Vulnerabilities Using IoT-Enabled Intrusion Detection System. *Sensors*, *22*(2), 567.
32. Hanan Hindy, Christos Tachtatzis, Robert Atkinson, Ethan Bayne, Xavier Bellekens (2020). MQTT-IoT-IDS2020: MQTT Internet of Things Intrusion Detection Dataset. IEEE Dataport.
33. Kostas, K., Just, M., & Lones, M. A. (2024). Individual Packet Features are a Risk to Model Generalisation in ML-Based Intrusion Detection. arXiv.