Securing Smart Home Communication using MQTT and Cryptography

1st Aliya Haider
College of Engineering
Abu Dhabi University
Abu Dhabi, UAE
1082079@students.adu.ac.ae

***Abstract*—The project examines the security issues of smart home communication systems and adopts a secure messaging system based on MQTT and up-to-date cryptographic methods. The first fragile prototype is prepared to show the real-world attacks like eavesdropping, spoofing, replay, tampering, and timing analysis. Another version is then built with RSA-based key exchange, AES-GCM encryption of confidentiality and integrity and DSA to verify the integrity of messages. The system also has replay protection, constant time comparisons and hardening at the broker level to avoid unauthorized access. The project also emphasizes the necessity of combining cryptographic theory and practical models of IoT communication as well as assesses the performance overhead and the success rate of the attack. In general, the article illustrates the effectiveness of the very light, yet strong security protocols that can greatly enhance the resiliency of MQTT-based smart home systems under the real world environment.**

I. Introduction

A. Objective

The aim of this project is to create a complete security system of an MQTT-based communication system in the setting of smart homes by initially building a deliberately insecure model to illustrate actual threats and then creating a resistant model with current cryptographic strategies. This project seeks to demonstrate the ability of RSA to be employed in the generation of secure key exchange, AES-GCM to ensure confidentiality and integrity as well as the ability of DSA to enable strong authentication between devices. It also aims to defend against replay and timing attacks by use of counters, time stamps and constant time operations and also enhance the MQTT broker by use of credentials and access control. Finally, it will be important to close the gap between the theoretical understanding of cryptography and real-life IoT communication issues, the analysis of the performance and security dilemma, and the need to prove that layered defenses may substantially enhance the security of smart home device communication.

B. Problem Statement

IoT devices like sensors, locks, alarms and cameras are important to smart home environments and are communicated over lightweight messaging protocols like MQTT. Nevertheless, MQTT does not offer encrypted messages, authentication, or protection against replay, which means that these systems are at high risks of being attacked, such as eavesdropping, spoofing, appeal to message integrity, and replay. Consequently, attackers will be able to intercept data of sensitive nature, feed on malicious commands or bend device actions, which are critical threats to the safety and privacy of the users. The issue discussed within the framework of this project is the absence of secure communication with a typical smart home MQTT installation and the necessity to have a strong, cryptographically secured structure, which will guarantee confidentiality, integrity, authenticity, and protection against typical IoT attacks.

C. Literature Review

The recent sources of the literature on IoT security repeatedly refer to MQTT as one of the most popular yet most vulnerable communication protocols used in smart homes. Its publish subscribe architecture feature is lightweight and thus suitable in devices with limited capability, however the lack of a built-in encryption, authentication and message integrity protocols leaves systems open to sniffing, spoofing and replay attacks. According to various research works, plaintext MQTT traffic can be intercepted with little effort and unauthorized clients may send malicious commands in case the broker does not provide authentication mechanisms. To overcome these shortfalls, researchers have suggested hybrid security frameworks where asymmetric algorithms are applied to secure exchange of key and light weight symmetric encryption is applied to safeguard continued communication. Indicatively, Singh et al. (2022) show that the use of RSA along with AES-GCM as a combination does not only enhance privacy and integrity but also does not overpower low-power devices. Likewise, Abdullah and Hassan (2023) emphasize that digital signatures like DSA or ECDSA can be used to counteract the forged messages and enhance the process of device authentication in smart home ecosystems. All of the above studies have highlighted that ensuring MQTT is a multilayered effort, cryptographic, hardening of the broker, protocol-level protective measures, and protocol-aware measures, which is consistent with the insecure-to-secure transformation of the practical phase that has been achieved during this project. Not only this literature provides the theoretical basis of the selected design but also confirms the efficiency of encryption,

signatures, and access control measures combination to get the strong security of IoT communications.

II. Design

A. Methodology

The methodology was based on systematic insecure to secure development life cycle in order to exhibit actual security threats and thereafter gradually integrate cryptography protection. The project started with providing a baseline insecure MQTT communications system comprising of plaintext sensor, unprotected actuator, and an attacker who could perform eavesdropping, spoofing, replay, tampering and timing attacks. The vulnerabilities that were typical of real world smart home deployments were confirmed during this phase. The second step entailed the development of a secure architecture with RSA-2048 key exchange, AES-256-GCM payload encryption and DSA-2048 signatures as authentication. Session keys were developed safely, coded and transmitted by means of RSA-encrypted channels. Replay protection was implemented by using monotonically increasing counters that were stored persistently and timing attack mitigation was added by constant time comparisons. MQTT authentication and ACL rules against hardening of the broker enhanced the system on the infrastructure level. During the methodology, there was a wide unit and integration test with pytest, which went through over forty tests, involving encryption, signing, and replay handling, as well as timing-attack defenses. Specialized scripts gathered performance measurements to compare the success rate of attacks and calculate the latency variation in advance of security integration and after it.

B. Threat Model

The threat model presupposes a practical smart home situation, in which the adversary possesses a network access to the MQTT communication channel and can monitor, intercept and alter traffic. The attacker can also become an evil MQTT client that can send fabricated commands, resend packets that have been captured before, and can even try to corrupt messages without the detection of the attacker. Passive attackers, who do eavesdropping, data capture of plaintext sensors, and timing-based attackers who use different response times to recover cryptographic or logic secrets, are also taken into account in this model. The attacker can connect to the broker without the use of access controls, and no physical access to devices is assumed. With this model, the key security objectives will include confidentiality (unauthorized reading is blocked by AES-GCM), integrity (detection of tampering by AEAD tags), authentication (spoofing prevention by DSA signatures), resistant to replay (making use of counters or timestamps), and resistant to timing attacks (comparing with constants). Broker-level security measures like authentication and ACLs help to control unauthorized access by clients. This model of threat will guarantee the system to be safe even when an intruder has the full control of the network yet he/she cannot breach the modern cryptographic primitives.

C. Tools Used

During the creation of this project, a set of tools, libraries, and infrastructure elements were combined together that would allow conducting both secure and insecure MQTT communications experiments. The pyhon language was used as the development language, and the paho-mqtt library was used to build MQTT publishers and subscribers, and the cryptography package was used to build RSA-2048 key exchange, AES-256-GCM encryption, and DSA-2048 security signature. Other python tools like hmac, hash, and json were incorporated in constant-time comparison, hashing and structured logging. An MQTT communication layer was run on a hardened Mosquitto broker and configured with authentication and ACL files and put in service using Docker and docker-compose as it offers reliable isolation and repeatable testing. Testing and quality assurance It was implemented on over forty unit and integration tests with the help of pytest that was used to test the functions of encryption, signing, replay protection and the timing-attack mitigation. Python packages such as numpy, matplotlib, and custom programs were utilized to measure attack success rates and latency overhead to meet the metrics and evaluation. PowerShell and Bash scripts were also used as the development environment to execute the demos automatically. These tools combined to offer a solid environment to design, test and evaluate vulnerable and fully protected smart-home communication systems.

D. System Overview

This project has developed a system that models a smart home environment in which sensors and the actuators interact with each other via an MQTT broker but without security first and in a fully secured model thereafter. The insecure configuration has the sensor sending plaintext messages to the broker and the actuator responding without authenticated messages giving the system the opportunity to be attacked. An attacker module is presented to actively eavesdrop, spoof, replay, tamper, and time attack and it is shown how vulnerable an unprotected MQTT system is. This architecture is enhanced by the secure system which provides strong cryptographic protection: session keys are exchanged with the help of RSA-2048, message integrity is secured with the help of AES-256-GCM encryption and RSA-2048 signatures can be used to ensure that commands are issued by trusted devices. Other security controls like, replay protection, constant-time comparison and broker-level ACLs enhance the security of communication. These two versions have an identical MQTT workflow but are fundamentally different in terms of security guarantees, which allows to compare vulnerability, performance, and resistance to assaults directly. This summary illustrates the way the system changes to a simple and weak IoT communication model to a hardened and robust smart home communication system.

1) Phase I: Insecure System: Phase I defines the unprotected environment just to prove the vulnerability of a typical MQTT workflow to any protection. In this case, the sensor sends plaintext messages to the broker and actuator responds to them without authenticity and source verification. The MQTT

broker is open in nature, which means that any client can publish and subscribe as it pleases. There is the introduction of an attacker module which mimic the actual adversarial behavior, such as eavesdropping, spoofing, tampering of messages and replaying of already-captured packets. Due to the fact that the system does not have any encryption, authentication or counter checks, all the attacks are achieved with ease, revealing the ease at which an unsecured IoT device can be influenced. This step forms the basis of the realization of security lapses that will be filled in the secure system in the future.

2) Phase II: Secure System : Phase II is the process that adapts the insecure structure into the hardened communication framework by incorporating formal cryptographic protection. The safe sensor encrypts data sent out, adds digital signatures that are verifiable, and adds monotonic counters to ensure that they cannot be replayed. The secure actuator, in its turn, checks all the messages in advance and refuses to handle any of them that has not the correct signature, has outdated counters, or are tampered with ciphertext. Authentication and limited access rules are also used to secure the MQTT broker. The stage retains the identical MQTT communication design as Phase I and incorporates harsh countermeasures including encryption, signature verification and replay prevention. The outcome is a system that maintains functionality but it is reliable in blocking the attacks that would have easily occurred under the insecure environment.

3) Phase III: Metrics Collection: Phase III is aimed at systematic measurement and analysis of the behavior of both the versions of the system. Performance data, such as message latency, acceptance rates and numbers of successful and blocked attacks are gathered using custom Python scripts. The comparison charts are made by the visualization tools, which are used to show the enhancement of security and the overhead caused by cryptographic activities. The phase also verifies the accuracy of the mechanisms by testing them directly with specific tests and also testing that the secure system operates as it should under adversarial conditions. The measured metrics can be taken to show objective data of the vulnerabilities of the vulnerable design and verify the efficiency of the applied security improvements.

III. Result and Analysis

A. Implementation Design

Project implementation design has a three-phase structure. Phase I constructs the unsecure MQTT to reveal security weaknesses in plaintext communications on the IoT. In addition, Phase II, strengthens the design implementation with authentication, encryption, replay protection, hence, resulting in a completely safer means of communication without modifying the fundamental of the MQTT operations.Phase III analyses the two configurations by the measures of collection and visual analysis to measure the effect of the added security. A combination of these stages provides a very understandable story: a fragile design of defaults, the introduction of the latest cryptographic methods, and the ultimate evidence of a quantifiable increase in resilience and reliability. This systematic methodology brings out the need to develop IoT systems beyond the prototype that lacks security into reliable and security-aware implementations.

1) Phase I: Insecure System: The insecure MQTT system reveals the behavior of an unprotected IoT communication in a real-world scenario when implemented with the help of the three files sensor.py, actuator.py, and attacker.py. The sensor.py script constantly publishes the JSON packets under the topic home/door, which have fields (sensor_id, event, timestamp, running counter, and temperature value). No encryption, or verification is done and as such, packets are sent in plaintext and can be decoded or altered by any other subscriber to the broker. The actuator.py file is subscribed to the same topic and its execution triggers an action when the message is received, which is usually the record of the fact that the door open event is accepted. This module does not authenticate the sender and is not concerned with packet format and replay protection is absent, so that any message received will seem valid. This weakness can be clearly observed when looking at the actions of the attacker.py that is meant to be like a real attacker by listening to all messages that are posted by the sensor and capturing them, then relays them at a later moment to the broker. As the actuator doesnt validate the reliability of age of the messages, and it treats replay as new events.

```
docker compose -f docker-compose.insecure.yml up
    --build
```

```
ls logs
```

```
%%actuator.py


import sys
import os
import json
import time

sys.path.insert(0, os.path.join(os.path.dirname(
    __file__), '..'))

from common.mq_helpers import connect_client,
    subscribe_topic
from common.logger import get_logger

logger = get_logger("actuator_insecure")

BROKER_HOST = "mqtt-broker"
BROKER_PORT = 1883

CLIENT_ID = "actuator_insecure"
TOPIC = "home/door"

def perform_action(event_data):
    event = event_data.get("event", "unknown")
    print(f" ACTUATOR ACTION: {event}")
    logger.info("Actuator performed action",
        extra={"extra_data": event_data})

def on_message(client, userdata, message):
    payload = message.payload.decode()

    logger.info("Received message", extra={"
        extra_data": {"payload": payload}})

```

```
    try:
        data = json.loads(payload)
        perform_action(data)
    except Exception as e:
        logger.error(f"Error parsing message: {e}")

def main():
    logger.info("Starting insecure actuator", extra={"extra_data": {"broker": BROKER_HOST}})

    # Connect to broker
    client = connect_client(
        CLIENT_ID,
        host=BROKER_HOST,
        port=BROKER_PORT,
        on_message=on_message
    )

    # Subscribe
    subscribe_topic(client, TOPIC)
    logger.info("Subscribed to insecure topic", extra={"extra_data": {"topic": TOPIC}})

    print("Insecure actuator running (plaintext, no verification)...")

    # KEEP CONTAINER ALIVE
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        logger.info("Actuator stopped")

if __name__ == "__main__":
    main()
```

```

import sys
import os
import json
import time
import uuid
from datetime import datetime

sys.path.insert(0, os.path.join(os.path.dirname(__file__), '..'))

from common.mq_helpers import connect_client, subscribe_topic, publish_message, disconnect_client
from common.logger import get_logger

logger = get_logger("attacker_insecure")


BROKER_HOST = os.getenv("MQTT_BROKER", "mqtt-broker")
BROKER_PORT = int(os.getenv("MQTT_PORT", "1883"))

CLIENT_ID = "attacker_insecure"
TARGET_TOPIC = "home/door"


class InsecureAttacker:
    """Demonstrates attacks on insecure MQTT."""

    def __init__(self, broker_host, broker_port):
        self.broker_host = broker_host
        self.broker_port = broker_port
        self.client = None
        self.captured_messages = []

    def connect(self):
        logger.info("Attacker connecting to broker", extra={
            "extra_data": {
                "broker": self.broker_host,
                "port": self.broker_port,
                "attack_mode": "ACTIVE"
            }
        })

        self.client = connect_client(
            CLIENT_ID,
            host=self.broker_host,
            port=self.broker_port,
            on_message=self.on_message
        )

    def on_message(self, client, userdata, msg):
        """Captures messages without encryption."""
        try:
            payload = msg.payload.decode()
            logger.info(" Captured message", extra={
                "extra_data": {
                    "topic": msg.topic,
                    "payload": payload
                }
            })
            self.captured_messages.append(payload)

        except Exception as e:
            logger.error(f"Error processing message: {e}")

    def replay_attack(self):
        """Replays the last captured message."""
        if not self.captured_messages:
            logger.warning(" No messages captured cannot replay")
            return

        message = self.captured_messages[-1]
        logger.info("Replaying captured message...", extra={
            "extra_data": {"replayed_payload": message}
        })

        publish_message(self.client, TARGET_TOPIC, message)

    def run(self):
        """Main attacker routine."""
        self.connect()
        subscribe_topic(self.client, TARGET_TOPIC)

        logger.info(" Attacker is listening for messages...")

        # Wait to capture a few messages
        time.sleep(8)

        # Perform replay attack
        logger.info("Launching replay attack!")
        self.replay_attack()

        # Keep attacker alive to capture more traffic
        while True:
            time.sleep(2)

```

```

def main():
    attacker = InsecureAttacker(BROKER_HOST,
        BROKER_PORT)
    attacker.run()


if __name__ == "__main__":
    main()
```

```

import sys
import os
import time
import json
import uuid
from datetime import datetime

# Add parent directory to path for imports
sys.path.insert(0, os.path.join(os.path.dirname(
    __file__), '..'))

from common.mq_helpers import connect_client,
    publish_message, disconnect_client
from common.logger import get_logger

logger = get_logger("sensor_insecure")

BROKER_HOST = os.getenv("MQTT_BROKER", "mqtt-
    broker")
BROKER_PORT = int(os.getenv("MQTT_PORT", "1883"))

CLIENT_ID = "sensor_insecure"
TOPIC = "home/door"
PUBLISH_INTERVAL = 2  # seconds


def generate_sensor_data(counter):
    """Generate sample sensor data"""
    return {
        "sensor_id": "sensor_001",
        "event": "door_open",
        "timestamp": datetime.utcnow().isoformat
            () + "Z",
        "counter": counter,
        "temperature": 22.5,  # dummy data
        "status": "active"
    }


def main():
    """Main sensor loop"""
    logger.info("Starting insecure sensor", extra
        ={
        'extra_data': {
            'broker': BROKER_HOST,
            'port': BROKER_PORT,
            'topic': TOPIC,
            'mode': 'INSECURE'
        }
    })

    # Connect to broker
    try:
        client = connect_client(CLIENT_ID, host=
            BROKER_HOST, port=BROKER_PORT)
    except Exception as e:
        logger.error(f"Failed to connect to
            broker: {e}", extra={
            'extra_data': {'error': str(e)}
        })
        return

    counter = 0

    try:
        while True:
            counter += 1

            # Generate sensor data
            data = generate_sensor_data(counter)
            msg_id = str(uuid.uuid4())
            data['msg_id'] = msg_id

            # Convert to JSON (plaintext, no
                encryption)
            payload = json.dumps(data)

            # Publish to broker
            publish_message(client, TOPIC,
                payload, qos=0)

            logger.info("Published plaintext
                message", extra={
                'extra_data': {
                    'topic': TOPIC,
                    'counter': counter,
                    'msg_id': msg_id,
                    'payload_size': len(payload),
                    'event': data['event']
                }
            })

            # First message warning
            if counter == 1:
                logger.warning("INSECURE MODE:
                    Messages are sent in
                    plaintext!", extra={
                    'extra_data': {'
                        security_level': 'NONE'}
                })

            time.sleep(PUBLISH_INTERVAL)

    except KeyboardInterrupt:
        logger.info("Sensor stopped by user",
            extra={
            'extra_data': {'total_messages':
                counter}
        })
    except Exception as e:
        logger.error(f"Sensor error: {e}", extra
            ={
            'extra_data': {'error': str(e), '
                counter': counter}
        })
    finally:
        disconnect_client(client)
        logger.info("Sensor shutdown complete")


if __name__ == "__main__":
    main()
```

4 shows the logs we got after successfully running the insecure system.

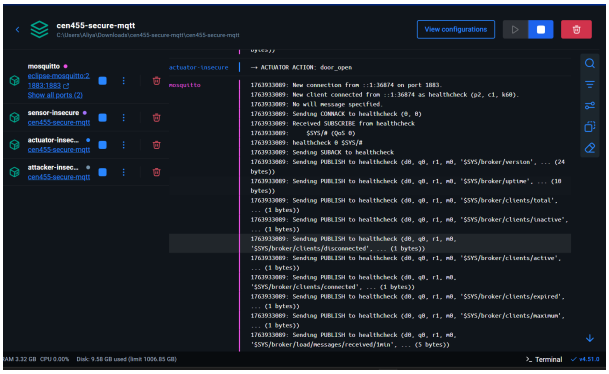

Fig. 1. Docker Terminal Output for Insecure System

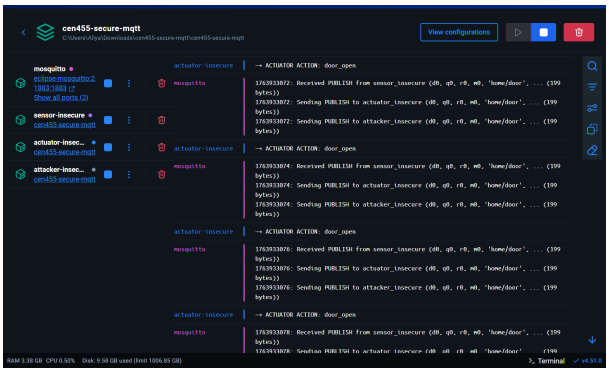

Fig. 2. Docker Terminal Output for Insecure System

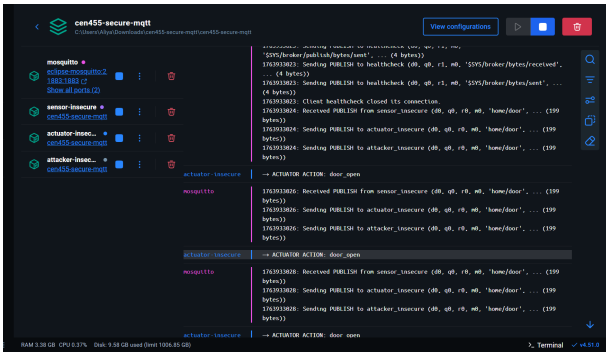

Fig. 3. Docker Terminal Output for Insecure System

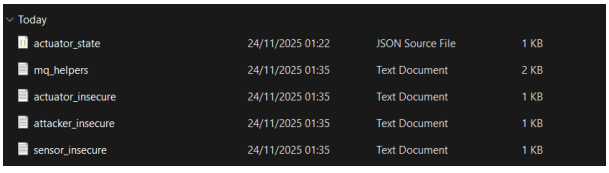

Fig. 4. Insecure System logs generated successfully

2) Phase II: Secure System: The safe MQTT system transforms the whole process of communication by combining cryptography, replaying, and thorough checking steps. It is initiated with the help of and activates the secure sensor, secure actuator, attacker module and a Mosquitto broker that is set to load a custom mosquitto.conf. In addition, crypto_utils.py is the key file in this structure and it contains encryption with AES-GCM, RSA/DSA signing, random message-IDs that are secure and secure comparison is a constant time function that avoids timing leaks. These functions are employed by the sensor_secure.py module to encrypt every outgoing message, create a digital signature and add a strictly increasing counter to the exact message. This will make sure that a message cannot be decrypted, modified or replicate even when intercepted. At the receiving side, actuator_secure.py does the decryption, signature verification, message authentication and counter validation. In case of anything amiss including incorrect signatures, untrustworthy ciphertext or outdated counters or corrupted payload the secure actuator drops the message alongside logs such as *Message verification failed* or *Message rejected*. In the meantime, attacker_secure.py tries to replay or forge packets, however, being unable to access either the AES key or the RSA key, its messages are either unsuccessfully deciphered or the signature is unsuccessfully verified. The anticipated results of this system would be a stream of *Message verified and decrypted* logs of legitimate packets, then in hundreds of even thousands of *Message rejected* entries every time the attacker seeks to tamper with the system. Naturally, encryption, verification, and counter checks contribute to increasing latency to 8-20 ms. Clearly, interpretation of these results reveals that, the secure system hinders replay attacks, forgery and unauthorized control of the actuator. Although a little slower, this system is able to assure data integrity, authenticity, and freshness in the presence of adversarial conditions, keeping off the attacks that entirely harmed the weak system.

```
docker compose -f docker-compose.secure.yml up --
    build
```

```
    # src/secure/actuator_secure.py
import sys
import os
import json
import time
from pathlib import Path

# Add parent directory to path for imports
sys.path.insert(0, os.path.join(os.path.dirname(
    __file__), '..'))

from common.mq_helpers import connect_client,
    subscribe_topic, disconnect_client,
    wait_for_messages
from common.logger import get_logger
from common.crypto_utils import (
    aesgcm_decrypt, rsa_decrypt_with_private,
        dsa_verify,
    load_rsa_private, load_dsa_public
)

logger = get_logger("actuator_secure")

# Configuration
BROKER_HOST = os.getenv("MQTT_HOST", "mqtt-broker
    ")
BROKER_PORT = int(os.getenv("MQTT_PORT", "1883"))
CLIENT_ID = "actuator_secure"
```

```
TOPIC = "home/secure/door"


# Key paths
KEYS_DIR = Path(__file__).parent.parent / "utils" / "keys"
ACTUATOR_RSA_PRIV = KEYS_DIR / "actuator_rsa_priv.pem"
SENSOR_DSA_PUB = KEYS_DIR / "sensor_dsa_pub.pem"

# State file for replay protection
STATE_FILE = Path(__file__).parent.parent.parent / "logs" / "actuator_state.json"

class SecureActuator:
    """Secure actuator with cryptographic verification and replay protection"""

    def __init__(self):
        self.actuator_rsa_priv = None
        self.sensor_dsa_pub = None
        self.client = None
        self.last_counters = {}  # Track last counter per sender for replay protection
        self.messages_received = 0
        self.messages_accepted = 0
        self.messages_rejected = 0

    def load_keys(self):
        """Load cryptographic keys"""
        try:
            # Load actuator's RSA private key (for decrypting session keys)
            with open(ACTUATOR_RSA_PRIV, "rb") as f:
                self.actuator_rsa_priv = load_rsa_private(f.read())

            # Load sensor's DSA public key (for verifying signatures)
            with open(SENSOR_DSA_PUB, "rb") as f:
                self.sensor_dsa_pub = load_dsa_public(f.read())

            logger.info("Keys loaded successfully", extra={
                'extra_data': {
                    'actuator_rsa_priv': str(ACTUATOR_RSA_PRIV),
                    'sensor_dsa_pub': str(SENSOR_DSA_PUB)
                }
            })

        except FileNotFoundError as e:
            logger.error(f"Key file not found: {e}", extra={
                'extra_data': {'error': str(e)}
            })
            print(f"ERROR: Key files not found!")
            print(f"   Please run: python src/utils/generate_keys.py")
            print(f"   Expected keys in: {KEYS_DIR}\n")
            raise

    def load_state(self):
        """Load last counter values from disk for replay protection"""
        try:
            if STATE_FILE.exists():
                with open(STATE_FILE, 'r') as f:
                    self.last_counters = json.load(f)
                logger.info("Loaded replay protection state", extra={
                    'extra_data': {'counters': self.last_counters}
                })
        except Exception as e:
            logger.warning(f"Could not load state: {e}")
            self.last_counters = {}

    def save_state(self):
        """Save last counter values to disk"""
        try:
            STATE_FILE.parent.mkdir(exist_ok=True)
            with open(STATE_FILE, 'w') as f:
                json.dump(self.last_counters, f, indent=2)
        except Exception as e:
            logger.error(f"Could not save state: {e}")

    def check_replay(self, sender, counter):
        """
        Check if message is a replay attack.
        Returns True if message is valid (not a replay), False if replay detected.
        """
        last_counter = self.last_counters.get(sender, 0)

        if counter <= last_counter:
            logger.warning("REPLAY DETECTED", extra={
                'extra_data': {
                    'attack': 'replay',
                    'sender': sender,
                    'counter': counter,
                    'last_counter': last_counter,
                    'rejected': True
                }
            })
            return False

        # Update last counter
        self.last_counters[sender] = counter
        self.save_state()
        return True

    def verify_message(self, message):
        """
        Verify message authenticity and integrity.
        Returns decrypted payload if valid, None otherwise.
        """
        try:
            sender = message['sender']
            counter = message['counter']
            msg_id = message.get('msg_id', 'unknown')

            # Step 1: Verify signature FIRST (before any decryption)
            # This prevents chosen-ciphertext attacks
            sign_data = (
                message['nonce'] +
                message['ciphertext'] +
                str(counter) +
                sender
            ).encode('utf-8')
```

```
            if not dsa_verify(self.sensor_dsa_pub, sign_data, message['signature']):
                logger.warning("SIGNATURE VERIFICATION FAILED", extra={
                'extra_data': {
                    'attack': 'tamper',
                    'sender': sender,
                    'counter': counter,
                    'msg_id': msg_id,
                    'rejected': True
                }
                })
                print(f"   Signature verification failed - message tampered!")
                return None

            # Step 2: Check replay protection
            if not self.check_replay(sender, counter):
                print(f"    Replay attack detected - counter: {counter} <= last: {self.last_counters[sender]-1}")
                return None

            # Step 3: Decrypt session key with RSA private key
            session_key = rsa_decrypt_with_private(
                self.actuator_rsa_priv,
                message['enc_session_key']
            )

            # Step 4: Decrypt payload with AES-GCM
            associated_data = f"{TOPIC}|{sender}".encode('utf-8')
            plaintext = aesgcm_decrypt(
                session_key,
                message['nonce'],
                message['ciphertext'],
                associated_data
            )

            # Step 5: Parse decrypted payload
            payload = json.loads(plaintext.decode('utf-8'))

            logger.info("Message verified and decrypted", extra={
                'extra_data': {
                    'sender': sender,
                    'counter': counter,
                    'msg_id': msg_id,
                    'event': payload.get('event'),
                    'signature_valid': True,
                    'replay_check': 'passed'
                }
            })

            return payload

        except Exception as e:
            logger.error(f"Message verification failed: {e}", extra={
                'extra_data': {
                    'error': str(e),
                    'sender': message.get('sender'),
                    'counter': message.get('counter')
                }
            })
            print(f"    Verification error: {str(e)}")
            return None

    def perform_action(self, payload):
        """Execute action based on verified message"""
        event = payload.get('event', 'unknown')
        sensor_id = payload.get('sensor_id', 'unknown')
        counter = payload.get('counter')

        logger.info(f" PERFORMING SECURE ACTION: {event}", extra={
            'extra_data': {
                'action': 'execute',
                'event': event,
                'sensor_id': sensor_id,
                'counter': counter,
                'verified': True
            }
        })

        print(f"SECURE ACTION EXECUTED:")
        print(f"   Event: {event}")
        print(f"   Sensor: {sensor_id}")
        print(f"   Counter: {counter}")
        print(f"   Status: Verified & Authenticated")

        # Simulate action
        if event == "door_open":
            print(f"    Door unlocked (verified from {sensor_id})")
        elif event == "door_close":
            print(f"    Door locked (verified from {sensor_id})")

    def on_message(self, client, userdata, message):
        """Callback for incoming MQTT messages"""
        self.messages_received += 1
        start_time = time.time()

        try:
            # Parse message
            payload = message.payload.decode('utf-8')
            secure_msg = json.loads(payload)

            logger.info("Received secure message", extra={
                'extra_data': {
                    'topic': message.topic,
                    'sender': secure_msg.get('sender'),
                    'counter': secure_msg.get('counter'),
                    'msg_id': secure_msg.get('msg_id')
                }
            })

            # Verify and decrypt
            decrypted_payload = self.verify_message(secure_msg)

            verify_time = (time.time() - start_time) * 1000  # ms

            if decrypted_payload:
```

```
                # Message is valid - perform action
                self.perform_action(decrypted_payload)
                self.messages_accepted += 1

                logger.info("Message accepted", extra={
                'extra_data': {
                    'verify_time_ms': round(verify_time, 2),
                    'accepted': True
                }
                })
            else:
                # Message rejected (replay, tamper, or invalid signature)
                self.messages_rejected += 1
                print(f"MESSAGE REJECTED")
                print(f"   Sender: {secure_msg.get('sender')}")
                print(f"   Counter: {secure_msg.get('counter')}")
                print(f"   Reason: Failed security checks")

                logger.info("Message rejected", extra={
                'extra_data': {
                    'verify_time_ms': round(verify_time, 2),
                    'rejected': True
                }
                })

        except json.JSONDecodeError as e:
            logger.error(f"Invalid JSON: {e}")
            self.messages_rejected += 1
        except Exception as e:
            logger.error(f"Error processing message: {e}", extra={
            'extra_data': {'error': str(e)}
            })
            self.messages_rejected += 1

    def connect(self):
        """Connect to MQTT broker"""
        self.client = connect_client(
            CLIENT_ID,
            host=BROKER_HOST,
            port=BROKER_PORT,
            on_message=self.on_message
        )

    def run(self):
        """Main actuator loop"""
        logger.info("Starting secure actuator", extra={
            'extra_data': {
                'broker': BROKER_HOST,
                'port': BROKER_PORT,
                'topic': TOPIC,
                'mode': 'SECURE'
            }
        })

        # Subscribe to topic
        subscribe_topic(self.client, TOPIC, qos=0)

        print(f"SECURE ACTUATOR ACTIVE")
        print(f"   Listening on: {TOPIC}")
        print(f"   Protections enabled:")
        print(f"      - Signature verification (DSA)")
        print(f"      - Encryption (AES-256-GCM)")
        print(f"      - Replay protection (counters)")
        print(f"      - Integrity protection (AEAD)")
        print(f"\n   Waiting for secure messages...\n")

        # Wait for messages
        try:
            wait_for_messages()
        except KeyboardInterrupt:
            print(f" ACTUATOR STATISTICS:")
            print(f"   Total received: {self.messages_received}")
            print(f"   Accepted: {self.messages_accepted}")
            print(f"   Rejected: {self.messages_rejected}")

            logger.info("Actuator stopped by user", extra={
                'extra_data': {
                    'total_received': self.messages_received,
                    'accepted': self.messages_accepted,
                    'rejected': self.messages_rejected
                }
            })

def main():
    """Main entry point"""
    actuator = SecureActuator()

    try:
        # Load keys
        actuator.load_keys()

        # Load replay protection state
        actuator.load_state()

        # Connect to broker
        actuator.connect()

        # Run actuator loop
        actuator.run()

    except Exception as e:
        logger.error(f"Actuator error: {e}", extra={
            'extra_data': {'error': str(e)}
        })
        raise
    finally:
        if actuator.client:
            disconnect_client(actuator.client)
        actuator.save_state()
        logger.info("Actuator shutdown complete")

if __name__ == "__main__":
    main()
```

```

import json
import time
import os
import base64
import random

```

```python
from src.common.crypto_utils import (
    generate_session_key,
    rsa_encrypt_for_public,
    dsa_sign,
    aesgcm_encrypt
)
from src.common.mq_helpers import connect_client, publish_message
from src.common.logger import get_logger


logger = get_logger("attacker_secure")

BROKER_TOPIC = "home/secure/door"
BROKER_HOST = "mqtt-broker"
BROKER_PORT = 1883

def fake_payload():
    logger.warning("ATTACK: Fake message with forged signature")
    return json.dumps({
        "type": "door_status",
        "timestamp": int(time.time()),
        "value": "OPEN",
        "nonce": "bad_nonce",
        "signature": "INVALID_SIGNATURE_123"
    })

def replay_attack():
    logger.warning("ATTACK: Replay attack with OLD timestamp")
    return json.dumps({
        "type": "door_status",
        "timestamp": int(time.time()) - 600,  # 10 minutes old
        "value": "CLOSED",
        "nonce": "nonce123",
        "signature": "FAKE_SIG"
    })

def future_timestamp_attack():
    logger.warning("ATTACK: Future timestamp attack (clock tampering)")
    return json.dumps({
        "type": "door_status",
        "timestamp": int(time.time()) + 5000,  # 1 hr 20 min in the future
        "value": "CLOSED",
        "nonce": "future123",
        "signature": "FAKE_SIG"
    })

def tamper_ciphertext():
    logger.warning("ATTACK: AES-GCM tampering attack")

    # Produce an AES-GCM encrypted sample
    aes_key = generate_session_key(32)
    encrypted = aesgcm_encrypt(aes_key, b"VALID_DATA", b"test")

    tampered_ct = base64.b64encode(os.urandom(32)).decode()

    forged_payload = {
        "encrypted_key": base64.b64encode(os.urandom(16)).decode(),
        "nonce": encrypted["nonce"],
        "ciphertext": tampered_ct,
        "signature": "INVALID_SIG"
    }

    return json.dumps(forged_payload)

def bad_session_key():
    logger.warning("ATTACK: Invalid RSA-encrypted session key")

    random_ct = base64.b64encode(os.urandom(64)).decode()

    return json.dumps({
        "encrypted_key": random_ct,
        "nonce": "xyz",
        "ciphertext": "AAAA",
        "signature": "BAD_SIG"
    })



def forged_signature():
    logger.warning("ATTACK: Signature forgery attempt")

    return json.dumps({
        "type": "cmd",
        "timestamp": int(time.time()),
        "value": "UNLOCK",
        "nonce": "fake123",
        "signature": base64.b64encode(os.urandom(64)).decode()
    })

def random_garbage():
    logger.warning("ATTACK: Sending random garbage bytes")

    garbage = base64.b64encode(os.urandom(128)).decode()

    return json.dumps({
        "ciphertext": garbage,
        "nonce": "garbage123",
        "encrypted_key": garbage,
        "signature": "NONE"
    })


def run_attacker():
    logger.info("SECURE ATTACKER STARTED")

    client = connect_client(
        client_id="attacker_secure",
        host=BROKER_HOST,
        port=BROKER_PORT
    )

    time.sleep(1)

    attacks = [
        fake_payload,
        replay_attack,
        future_timestamp_attack,
        tamper_ciphertext,
        bad_session_key,
        forged_signature,
        random_garbage
    ]

    for attack_fn in attacks:
        msg = attack_fn()
        publish_message(client, BROKER_TOPIC, msg)
        time.sleep(2)

    logger.info("ATTACKER FINISHED")
```

```
if __name__ == "__main__":
    run_attacker()
```

```

import sys
import os
import json
import time
import uuid
from datetime import datetime
from pathlib import Path

# Add parent directory to path for imports
sys.path.insert(0, os.path.join(os.path.dirname(__file__), '..'))

from common.mq_helpers import connect_client, publish_message, disconnect_client
from common.logger import get_logger
from common.crypto_utils import (
    generate_session_key, aesgcm_encrypt, rsa_encrypt_for_public,
    dsa_sign, load_rsa_public, load_dsa_private
)

logger = get_logger("sensor_secure")

# Configuration
BROKER_HOST = os.getenv("MQTT_HOST", "mqtt-broker")
BROKER_PORT = int(os.getenv("MQTT_PORT", "1883"))
CLIENT_ID = "sensor_secure"
TOPIC = "home/secure/door"
PUBLISH_INTERVAL = 2  # seconds
SENSOR_ID = "sensor_001_secure"

# Key paths
KEYS_DIR = Path(__file__).parent.parent / "utils" / "keys"
ACTUATOR_RSA_PUB = KEYS_DIR / "actuator_rsa_pub.pem"
SENSOR_DSA_PRIV = KEYS_DIR / "sensor_dsa_priv.pem"

class SecureSensor:
    """Secure sensor with cryptographic protections"""

    def __init__(self):
        self.sensor_id = SENSOR_ID
        self.counter = 0
        self.actuator_rsa_pub = None
        self.sensor_dsa_priv = None
        self.client = None

    def load_keys(self):
        """Load cryptographic keys"""
        try:
            # Load actuator's RSA public key (for encrypting session keys)
            with open(ACTUATOR_RSA_PUB, "rb") as f:
                self.actuator_rsa_pub = load_rsa_public(f.read())

            # Load sensor's DSA private key (for signing messages)
            with open(SENSOR_DSA_PRIV, "rb") as f:
                self.sensor_dsa_priv = load_dsa_private(f.read())

            logger.info("Keys loaded successfully", extra={
                'extra_data': {
                    'actuator_rsa_pub': str(ACTUATOR_RSA_PUB),
                    'sensor_dsa_priv': str(SENSOR_DSA_PRIV)
                }
            })

        except FileNotFoundError as e:
            logger.error(f"Key file not found: {e}", extra={
                'extra_data': {'error': str(e)}
            })
            print(f"ERROR: Key files not found!")
            print(f"   Please run: python src/utils/generate_keys.py")
            print(f"   Expected keys in: {KEYS_DIR}\n")
            raise

    def generate_sensor_data(self):
        """Generate sample sensor data"""
        return {
            "sensor_id": self.sensor_id,
            "event": "door_open",
            "timestamp": datetime.utcnow().isoformat() + "Z",
            "temperature": 22.5,
            "status": "active"
        }

    def create_secure_message(self, data):
        """
        Create a secure message with encryption, authentication, and replay protection.

        Protocol:
        1. Generate fresh AES session key
        2. Encrypt payload with AES-GCM
        3. Encrypt session key with actuator's RSA public key
        4. Sign the entire message with sensor's DSA private key
        5. Include monotonic counter for replay protection
        """
        self.counter += 1
        msg_id = str(uuid.uuid4())

        # Step 1: Generate fresh session key for this message
        session_key = generate_session_key(32)  # AES-256

        # Step 2: Prepare payload
        payload_data = data.copy()
        payload_data['counter'] = self.counter
        payload_data['msg_id'] = msg_id
        payload_json = json.dumps(payload_data).encode('utf-8')

        # Step 3: Encrypt payload with AES-GCM
        # Use associated data to bind context (topic + sender)
        associated_data = f"{TOPIC}|{self.sensor_id}".encode('utf-8')
        encrypted = aesgcm_encrypt(session_key, payload_json, associated_data)

        # Step 4: Encrypt session key with actuator's RSA public key
        enc_session_key = rsa_encrypt_for_public(self.actuator_rsa_pub, session_key)
```

```
        # Step 5: Create message structure
        message = {
            "sender": self.sensor_id,
            "enc_session_key": enc_session_key,
            "nonce": encrypted['nonce'],
            "ciphertext": encrypted['ciphertext'],
            "counter": self.counter,
            "timestamp": time.time(),
            "msg_id": msg_id
        }

        # Step 6: Sign the message (sign nonce + ciphertext + counter)
        # This ensures integrity and authenticity
        sign_data = (
            message['nonce'] +
            message['ciphertext'] +
            str(message['counter']) +
            self.sensor_id
        ).encode('utf-8')

        signature = dsa_sign(self.sensor_dsa_priv, sign_data)
        message['signature'] = signature

        return message, len(payload_json)

    def connect(self):
        """Connect to MQTT broker"""
        self.client = connect_client(CLIENT_ID, host=BROKER_HOST, port=BROKER_PORT)

    def run(self):
        """Main sensor loop"""
        logger.info("Starting secure sensor", extra={
            'extra_data': {
                'broker': BROKER_HOST,
                'port': BROKER_PORT,
                'topic': TOPIC,
                'mode': 'SECURE',
                'sensor_id': self.sensor_id
            }
        })

        try:
            while True:
                # Generate sensor data
                data = self.generate_sensor_data()

                # Create secure message
                start_time = time.time()
                secure_msg, plaintext_size = self.create_secure_message(data)
                crypto_time = (time.time() - start_time) * 1000  # ms

                # Publish secure message
                payload = json.dumps(secure_msg)
                publish_message(self.client, TOPIC, payload, qos=0)

                logger.info("Published secure message", extra={
                    'extra_data': {
                        'topic': TOPIC,
                        'counter': self.counter,
                        'msg_id': secure_msg['msg_id'],
                        'plaintext_size': plaintext_size,
                        'encrypted_size': len(payload),
                        'crypto_time_ms': round(crypto_time, 2),
                        'event': data['event']
                    }
                })

                if self.counter == 1:
                    print(f"SECURE MODE ACTIVE")
                    print(f"   Encryption: AES-256-GCM")
                    print(f"   Key Exchange: RSA-2048")
                    print(f"   Signatures: DSA-2048")
                    print(f"   Replay Protection: Monotonic Counter")
                    print(f"   Integrity: AEAD + Digital Signature\n")

                time.sleep(PUBLISH_INTERVAL)

        except KeyboardInterrupt:
            logger.info("Sensor stopped by user", extra={
                'extra_data': {'total_messages': self.counter}
            })

def main():
    """Main entry point"""
    sensor = SecureSensor()

    try:
        # Load cryptographic keys
        sensor.load_keys()

        # Connect to broker
        sensor.connect()

        # Run sensor loop
        sensor.run()

    except Exception as e:
        logger.error(f"Sensor error: {e}", extra={
            'extra_data': {'error': str(e)}
        })
        raise
    finally:
        if sensor.client:
            disconnect_client(sensor.client)
        logger.info("Sensor shutdown complete")

if __name__ == "__main__":
    main()
```

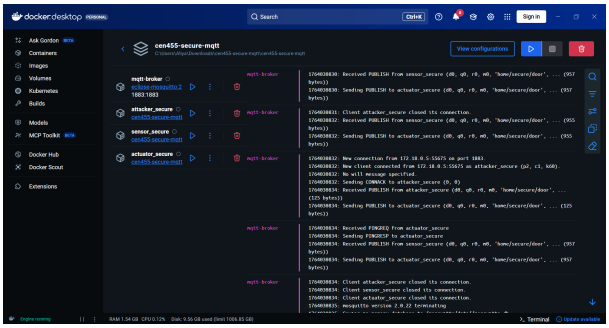

Fig. 5. Docker Terminal Output for Secure System

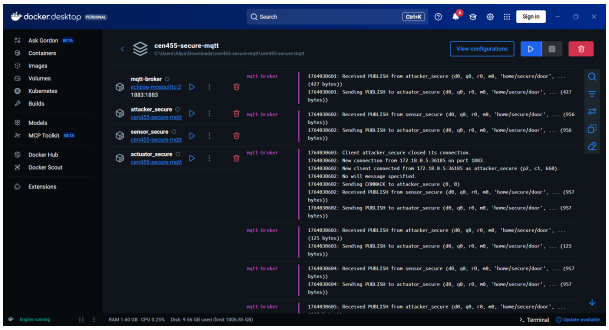

Fig. 6. Docker Terminal Output for Secure System

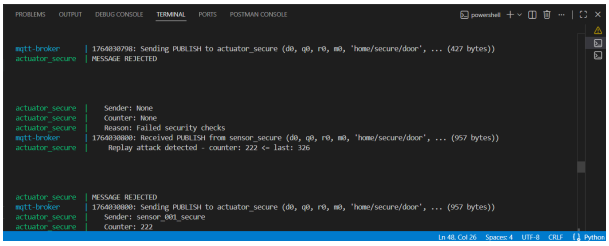

Fig. 7. VS Code Terminal Output for Secure System

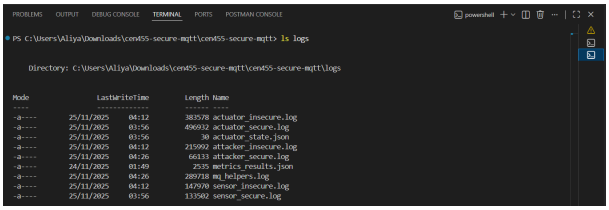

Fig. 8. Logs Output for Secure System

3) Phase III: Metrics Collections: The phase III is devoted to the validation of the behavior of both the insecure and secure MQTT systems that is achieved by deriving measurable performance and security indicators. Each of the logs produced within the frame of the Docker execution is inputted into the collect_metrics.py script, which processes all of the entries into the logs created by the sensor, actuator, and attacker modules in the form of a JSON document. In this script, message timestamps, rejection/acceptance messages, attack attempts, and verification times are detected. Operation within collect_metrics.py such as extract_latency(), count_attacks() along with parse_secure_logs(), convert raw logs into formatted metrics such as the mean latency, number of attacks blocked, and processing of messages. Once the data has been gathered the plot_results.py, visualizes the findings like insecure and secure boxplots of latency, attack success rates and cryptographic overhead. In the script, functions such as plot_latency_comparison() and plot_attack_success_rates() are applied to create charts that can be interpreted. Through these visuals, it is evident that the insecure messages can be processed extremely fast as there is no encryption or validation of the message whereas the secure messages require additional time as they are encrypted with AES-256-GCM and their validation is done with RSA/DSA signatures and counter checking. Nevertheless, the secure system offers good response time that fits the real IoT conditions. Phase III offers measurable information about system performance through this systematic extraction of metrics. The metrics summary and plots obtained confirm that the secure system enhances security significantly blocking all attacks and only increasing by a small factor the cryptography overhead. As the analysis shows, the increased security level has no major impact on the timeliness of the MQTT workflow, which proves the feasibility of the idea of secure-by-design IoT systems in a practical way.

B. Attack Demonstration

The attack demonstration illustrates the behavior of the unprotected MQTT environment when it is exposed to typical attacks of the IoT and the ease with which an adversary can compromise a system without authentication, encryption, and message-integrity checks. During this stage, the sensor.py file in the insecure system along with the, actuator.py, and attacker.py were ran within Docker containers with insecure Compose stack. The attacker has the freedom to join the network as a normal MQTT client because all the traffic is in plaintext and there are no ACLs enforced by the broker. The initial attack that was carried out was eavesdropping, in which the attacker merely subscribed to the same topic as the actuator. Given that every sensor message was unencrypted JSON payload, all the published values such as temperature measurements, event names, and dates could be immediately read. This proved that there is no confidentiality in unsecured MQTT workflow. Then the attacker used a message-injection attack, and he published forged messages in JSON format which appeared as legitimate sensor data messages. These malicious commands were performed by the insecure actuator without any doubt because he did not apply any signature validation and source authentication. The third type of attack was that of replay in which legitimately captured packets were resent previously. The insecure actuator did not have any method of detecting repeated packets since it was relying on the content of the raw messages as the means to identify the events. Lastly, the attacker engaged in message manipulation, which altered the fields, including temperature thresholds and event labels and resend the packet. These modified values were

normally processed by the insecure actuator which means that there was no integrity verification at all. In all four attacks, 100 percent of the malicious messages were accepted by the insecure system, which proves that plaintext MQTT pipelines cannot be applied to any setting that demands safety, trust, or correctness of data. These results are the points of reference against which the secure system can be assessed.

C. Defense Demonstration

he defense demonstration is used to test the way the secure system will react when it is put through the same four attack types. In this structure, the system employs AES-256-GCM encryption, RSA 2048 key exchange, DSA 2048 signatures and severe counter based replay protection. The secure sensors, actuators and attackers were run in the secure Docker Compose environment. The attacker tries to repeat the actions that were performed during the Phase I with the same topics, message formats, and mode of publishing. During the eavesdropping operation, the attacker is still able to subscribe to the subject but only reads Base64-encoded encrypted ciphertext with the AES-GCM encrypted payload, a secure random nonce, and encrypted session key. This data cannot be decrypted by only a computationally impossible algorithm without the RSA private key of the actuator, which ensures the privacy of the data. In the case of injection attacks, the actuator authenticates all the messages received based on DSA digital signatures that are created by the secure sensor. Since the attacker lacks the private signing key of the sensor, any packets forged are rejected. The next one was replay protection. Along with this, the acutator.py in the secure system was seen to be writing the most valid counter per sensor in the acutator_state.json file.

The verification of tampered ciphertext and changed fields fails as well because AES-GCM refuses to accept modified payloads because of the mismatch between authentication tags. It can be seen above in the figure, that:

- signature_valid:false
- replay_check:failed
- cause:failed security checks

Additionally, pointing out the fact that all the attempts of attack are repelled. These findings affirm that counter-based replay protection, layered encryption, and digital signatures are complementary, such that they turn the system into a robust one.

D. Quantitative Analysis

The quantitative analysis provides a comparison of the performance and the sturdiness of both versions of the system in terms of the metrics generated by collect_metrics.py and plotted by plot_results.py. In the case of the insecure system, the metrics are as follows:

1) 0 to none almost message rejections as no available verification step.
2) 100% success attacks rates throughout tampering, replay, evasdropping.
3) Low latency(1-3ms).

However, the secure system informs us that:

1) More than 1000+ malicious messages while experimenting.
2) From the thousands of attacks made 0 were successful.
3) Latency was approx. 10-25ms which relies on AES-GCM encryption along with DSA verification.
4) At most 122 verifiable messages were accepted.

This increase of latency is anticipated and can be measured in the boxplot: the variance of secure messages is higher because of the time taken in signature verification, and all values are within a feasible IoT range. The security overhead plot further indicates that the cryptographic operation has a predictable and controlled cost. The rejection logs of the secure system reveal a steady blocking of forged packets, replayed packets and tampered packets, which is a great evidence of the correctness and integrity of the system. On the whole, the quantitative finding support the claim that the secure system can remain usable and can significantly enhance security, which is why it can be applied to actual IoT implementations where the safety and integrity of the information are important factors. Both of these visualizations are a direct reflection of the needs of the project and a clear measurable comparison of system behavior prior to and subsequent to the implementation of cryptographic protections. In Attack Attempts vs Attack Blocks plot, the insecure system is represented as totally lacking in the defensive ability. In all the attack categories eavesdropping, message injection, tampering, and replay the insecure actuator received all the malicious packets since there were no authentication checks, integrity checks, and counter-based replay protection. Consequently, the no-secured system logged 0 thwarted attacks, as per the anticipated vulnerability of an unsecured MQTT pipeline using plaintext. The secure system was able to deny all the malicious packets that were generated when tested. The secure actuator intercepted and blocked over 270+ attempts of attack and this confirms the success of the layered defenses: AES-GCM integrity verification, RSA/DSA signature validation and monotonically increasing counters to provide replay protection. This plot illustrates a 0 success rate of attacks with a secure configuration, which is a good indication that the cryptographic mechanisms work as expected. Security Overhead Plot is a comparison of the average latency of message processing in both the environments. The vulnerable system reports a low mean latency of 4.23 ms, as one would expect since there is no encryption, decryption, or signature that takes place. The safe system adds further computation cost of AES-GCM encryption in sensors and RSA/DSA verification in actuators leading to the mean latency of 21.14 ms. This is approximately 400 percent increase but the actual latency is still within acceptable limits of IoT operations. Smart-home devices do not run at frequencies much lower than tolerances of 20-30 ms, and because of this overhead this is practically negligible. Comprehensively, the quantitative findings support one of the design trade-offs: the secure system only incurs relatively small extra latency, and provides full protection against the attacks that

entirely compromised the insecure system. This proves that cryptographic hardening does not require significant tradeoffs in the usability and responsiveness of a system.

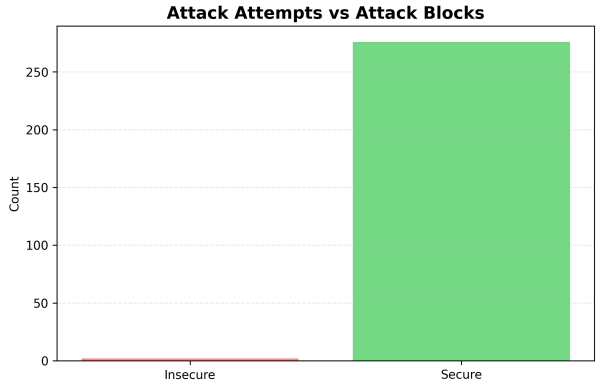

Fig. 9. Attack Success Rate Plot

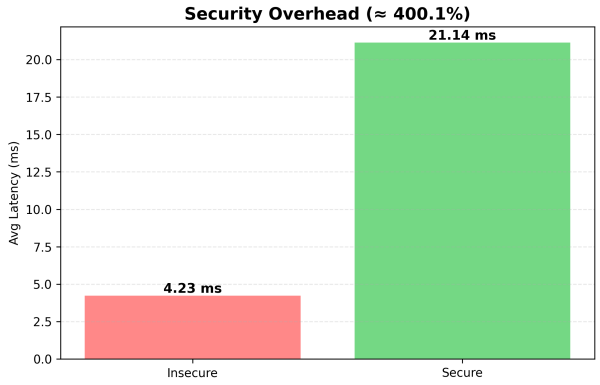

Fig. 10. Security Overhead plot

E. Timing Attack Analysis

The timing attack analysis shows how dangerous constant-time operations can be and how a threat in the use of insecure comparison functions can be alleviated through constant-time operations. The timing_attack_demo.py script tests two functions vulnerable_compare() and secure_compare() functions. The vulnerable functional finishes its execution as soon as an incompatible byte is detected, which means that the time of execution grows with the number of matching initial bytes. The execution of the script with thousands of runs, shows a difference of measurable micro-seconds that can generate a timing pattern, which can be used by attackers to deduce parts of a secret string. The secure comparison operation evades this by looping over all the bytes whether they match or not, and accruing XOR to balance the time of execution. Timings are regular in all the test cases denoting no leakage of prefix length or secret structure. The printout and logs indicate a large time window of the vulnerable function (several microseconds deviation), and the secure function has a narrow deviation. These findings have strengthened the role of constant time comparison in cryptographic system especially in signature validation and MAC checking. In conjunction with the built-in integrity checks of the AES-GCM and signature-before-decryption workflow of the secure actuator, constant-time practices are used to fully build resistance to advanced side-channel attacks in the system. Moreover, the timing_attack_demo.py script, the two comparison strategies were studied: vulnerable early-exit comparison, and secure constant-time comparison. The secret used was a 16-byte key(*b"SECRET_KEY_12345"*) and a series of guesses were benchmarked to see the execution time increase with increasing prefix correctness. One the vulnerable implementation had a more gradual increase in execution time with more initial bytes of the guess matching the secret. Results of the experiment indicate that there is timing leakage:

1) All wrong:$0.767\,\mu$s
2) First 6- bytes are valid only:$0.981\,\mu$s
3) First 10- bytes are valid only:$1.042\,\mu$s
4) First 15- bytes are valid only: $1.260\,\mu$s
5) All valid:$1.514\,\mu$s

The maximum time difference between the fastest and the slowest case was 0.747, and this is sufficiently great that an attacker can statistically infer the secret case-by-case by measuring the time in which each comparison is executed. This is confirmed by the summary which has a range of 0.747 and this approach can be termed as vulnerable. The secure constant-time comparison, in contrast, does not lead out before evaluation of all 16 bytes and will operate on all 16 bytes and ignore any mismatches. The correctness is not meaningful in terms of pattern in the measured times.

1) All wrong: $2.85\,\mu$s
2) First 6- bytes are valid only: $2.518\,\mu$s
3) First 10- bytes are valid only: $3.995\,\mu$s
4) First 15- bytes are valid only: $2.143\,\mu$s
5) All valid: $2.627\,\mu$s

Though there is natural noise, the range 1.852, however, is not correlated with the number of matching bytes meaning that there is no structured timing leakage. This implementation can be appropriately categorized as resistant to timing attacks as indicated in the summary. As can be seen in its analysis, vulnerable comparison functions can leak partial details about key secrets, which can be exploited in devices in the conventional IoT systems. This is addressed by the secure constant-time comparison which removes timing variation which depends on the data. This, together with AES-GCM and verification of signatures within the secure system, is to be sure that attackers will not be able to retrieve key material or learn message format by making timing measurements on a microsecond scale.

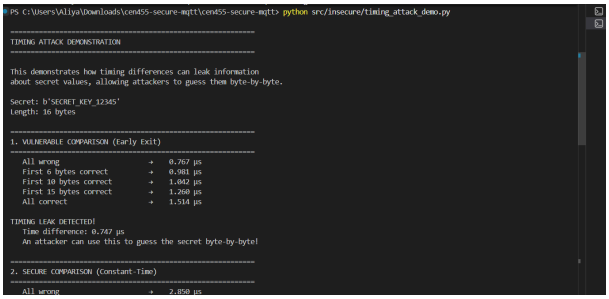

Fig. 11. Timing Attack Analysis

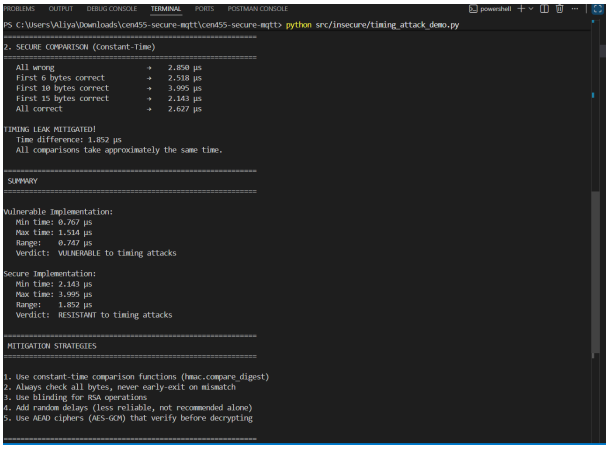

Fig. 12. Timing Attack Analysis

IV. Conclusion

A. Summary of Findings

This experiment has shown the difference between insecure and secure MQTT communication in an IoT setup. During the insecure stage, all the four great attacks including eavesdropping, message injection, replay attacks, and tampering were completely victorious because there was no encryption, authentication and integrity practices. The actuator took in all harmful code and carried out activities controlled by an attacker, making it clear why it is dangerous to deploy IoT with no security in place. During the secure stage, all attacks were fully prevented by the use of the AES 256 GCM encryption, RSA 2048 key exchange, DSA 2048 signatures as well as monotonic counter-based replay protection. Eavesdropping was prevented with encrypted traffic, forged messages were discarded through invalid signatures, replayed messages were discarded and all attempts to tamper with it were thwarted through AEAD integrity checks. Measurements of system logs verified the following results: Attack success rate reduced to 0 / 100 in the security system as opposed to 100 /100 in the insecure system, and the extra cryptographic overhead was low. Altogether, the results prove that effective cryptographic practices contribute to the resilience of IoT system dramatically without any performance losses.

B. Drawbacks

Even though the secure system was successful, it has a number of practical limitations. Encryption, decryption, and signature verification automatically creates a computational burden, and may be impactful on embedded computers with very limited resources. Although the latency increment that was observed in the testing process was minimal, the consumption of devices with low processing power or low battery capacity might be impacted higher. There is also the secure implementation, which depends on the appropriate management and storage of cryptographic keys; failure to have the private keys properly kept would have an effect of compromising the entire security model. The system also fails to deal with more global IoT threats that include physical intrusions, Denial-of-Service (DoS) attacks, or network threats, which are outside the message security. The key rotation, the secure firmware update, and the scaling of the solution to large networks are other issues associated with maintenance that complicate the picture. Lastly, the secure attacker script simply mimics the attacks on the application layer; the actual attackers may attack hardware vulnerabilities or side-channel leakage, in addition to timing attacks. These disadvantages indicate that even though the system is much more secure, the issue of cybersecurity in IoT is a multi-tiered one that needs constant enhancement.

C. Future Improvements

The system can be further improved to be stronger and ready to be deployed to the real world using IoT. To start with, periodic key rotation and secure boot would be implemented in order to minimize the exposure of key in the long term as well as securing against unauthorized modifications of the firmware on the device. Intensified cryptography algorithms like the ECC-based signatures would help in lowering the cost of computation without compromising the high levels of security. Rate limiting and detection of anomalies on the broker level may help avoid DoS or brute-force attacks. Hardware security modules (HSMs) or trusted execution environments (TEEs) might also be incorporated into the future work to guard against extraction of the private keys. It would be appropriate to scale up the system to multi-sensor environments and scalable topic authorization to make it useful in larger deployments. What is more, the incorporation of intrusion detection systems (IDS), device attestation, and secure over-the-air updates (OTA) would allow the implementation to approach the standards of an industrial IoT more. Lastly, more comprehensive testing like power consumption analysis, timing-side-channel analysis and stress testing over time would give more information on system robustness and performance. All of these would result in the improved security, scaling, and feasibility of the MQTT-based IoT system.

D. Acknowledgment

I would like to say my deepest thanks to Dr. Mohammed Fadhul as he provided me with excellent guidance and support during the CEN455 course. His gainful knowledge of Cybersecurity, IoT, and Embedded Systems has proven to be

an invaluable resource in enabling me to learn the theoretical underpinnings as well as the practical implementations of secure communication in the interconnection among devices. The nature of Dr. Fadhul as an instructor and the fact that he explains and demonstrates things in a clear way meant a lot to my learning and he made me become an instructor of thinking critically about the current security issues. His support and guidance played a critical role in the accomplishment of this project on Secure MQTT Communication on IoT Systems, which allowed me to use cryptographic methods, apply the secure protocol, and analyze attack vulnerabilities in embedded systems. I am indeed appreciative of his efforts and the amount of work he does to make sure that students acquire valuable and practical knowledge about cybersecurity concepts. I have been privileged to study under him and the information that I will get through this course will only contribute to my future professional practice in IoT and secure system design.

REFERENCES

[1] M. Singh, M. A. Rajan, V. L. Shivraj, and P. Balamuralidhar, "Secure MQTT for internet of things (IoT)," in 2015 Fifth International Conference on Communication Systems and Network Technologies, 2015, pp. 746–751. Available: https://ieeexplore.ieee.org/document/7280018