VANET Secure Routing Protocol - Full Report

Introduction

This project aims to implement a secure routing protocol for Vehicular Ad Hoc Networks (VANET) using digital signatures and hash functions. It simulates vehicles transmitting messages and validates their integrity to protect against common VANET attacks such as impersonation, message modification, and data tampering.

Objective

To design and simulate a lightweight and efficient secure routing protocol that ensures data integrity and authentication in VANET communication using cryptographic hash functions and digital signatures.

Simulation Tools Used

- Python for simulation logic
- Matplotlib for plotting
- Pandas for data handling
- Hashlib for cryptographic hashing

Cryptographic Techniques

The simulation utilizes SHA256, MD5, SHA1, Blake2b, and SHA3_256 hash functions to ensure message integrity. A salted hash technique is used for enhanced security against replay attacks.

Functionality and Features

- Simulates multiple vehicles with speed and location
- Detects vehicle collisions
- Validates message integrity using hashes
- Displays graphs for hash generation time, vehicle speeds, and positions over time

Performance Metrics

VANET Secure Routing Protocol - Full Report

The implemented simulation evaluates:

- Hash function computation time
- Vehicle speed consistency
- Movement paths and potential collisions
- Integrity validation efficiency

Conclusion

The project demonstrates the effectiveness of integrating cryptographic techniques in VANET simulations. It provides a strong base for further development into full routing protocol integration with network simulators like NS-2, NS-3, or OMNeT++.

VANET Secure Routing Protocol - Full Report

Appendix: Full Source Code

```
# VANET Secure Routing Simulation with Digital Signatures and Hash Functions
# Based on the project overview and included sample

import random
import hashlib
import matplotlib.pyplot as plt
import time
import pandas as pd

class Vehicle:
    def __init__(self, vehicle_id, speed, position):
        self.id = vehicle_id
        self.speed = speed
        self.position = position
        self.salt = "vanet" + str(random.random())  # Add a salt

    def move(self, dt):
        self.position = (self.position[0] + self.speed * dt * random.uniform(-0.1, 1.1),
                         self.position[1] + self.speed * dt * random.uniform(-0.1, 1.1))

    def check_collision(self, other, threshold=1):
        distance = ((self.position[0] - other.position[0]) ** 2 +
                    (self.position[1] - other.position[1]) ** 2) ** 0.5
        return distance < threshold

    def generate_message(self):
        message = {
            "vehicle_id": self.id,
            "speed": self.speed,
            "position": self.position,
        }
        return message, self.hash_message(message)

    def hash_message(self, message):
        message_bytes = str(message).encode()
        hashes = {}
        start_time = time.time()
        hashes["sha256"] = hashlib.sha256(message_bytes + self.salt.encode()).hexdigest()
        hashes["sha256_time"] = time.time() - start_time
        start_time = time.time()
        hashes["md5"] = hashlib.md5(message_bytes + self.salt.encode()).hexdigest()
        hashes["md5_time"] = time.time() - start_time
        start_time = time.time()
        hashes["sha1"] = hashlib.sha1(message_bytes + self.salt.encode()).hexdigest()
        hashes["sha1_time"] = time.time() - start_time
        start_time = time.time()
```

```
        hashes["blake2b"] = hashlib.blake2b(message_bytes + self.salt.encode()).hexdigest()
        hashes["blake2b_time"] = time.time() - start_time
        start_time = time.time()
        hashes["sha3_256"] = hashlib.sha3_256(message_bytes + self.salt.encode()).hexdigest()
        hashes["sha3_256_time"] = time.time() - start_time
        return hashes

    def check_integrity(self, message, hashes):
        for hash_type, hash_value in hashes.items():
            if hash_type in ["sha256", "md5", "sha1", "blake2b", "sha3_256"]:
                if hash_value != self.hash_message(message)[hash_type]:
                    return False
        return True

    def receive_message(self, message, hashes):
        print(f"Vehicle {self.id} received message from {message['vehicle_id']}: {message}")
        print(f"Received hashes: {hashes}")
        if self.check_integrity(message, hashes):
            print("Message integrity is valid.")
        else:
            print("Message integrity is NOT valid!")

def simulate(vehicles, dt, num_steps):
    hash_times = {"sha256": [], "md5": [], "sha1": [], "blake2b": [], "sha3_256": []}
    for _ in range(num_steps):
        for vehicle in vehicles:
            vehicle.move(dt)
                        collisions = [other for other in vehicles if vehicle != other and
vehicle.check_collision(other)]
            if collisions:
                  print(f"Collision detected between vehicle {vehicle.id} and: {', '.join(other.id
for other in collisions)}")

            message, hashes = vehicle.generate_message()
            for other in vehicles:
                if hasattr(other, 'receive_message'):
                    other.receive_message(message.copy(), hashes.copy())

            for hash_type, hash_time in hashes.items():
                if hash_type.endswith("_time"):
                    hash_times[hash_type[:-5]].append(hash_time)

    hash_data = []
    for hash_type, times in hash_times.items():
        for time_val in times:
            hash_data.append({"hash_type": hash_type, "time": time_val})
    hash_df = pd.DataFrame(hash_data)
    hash_df.boxplot(column="time", by="hash_type", showmeans=True)
    plt.title("Hash Generation Times")
```

```
    plt.ylabel("Time (s)")
    plt.xlabel("Hash Function")
    plt.xticks(rotation=45)
    plt.show()

def plot_speeds(vehicles, num_steps, dt):
    times = list(range(num_steps))
    speeds = {vehicle.id: [] for vehicle in vehicles}
    for i in range(num_steps):
        for vehicle in vehicles:
            vehicle.move(dt)
            speeds[vehicle.id].append(vehicle.speed)
                  collisions = [other for other in vehicles if vehicle != other and
vehicle.check_collision(other)]
        if collisions:
             print(f"Collision detected between vehicle {vehicle.id} and: {', '.join(other.id for
other in collisions)}")

    speeds_df = pd.DataFrame(speeds, index=times)
    speeds_df.plot(figsize=(10, 5), title="Vehicle Speeds Over Time", legend=True)
    plt.xlabel("Time")
    plt.ylabel("Speed")
    plt.show()

def plot_positions(vehicles, num_steps, dt):
    positions = {vehicle.id: [] for vehicle in vehicles}
    for i in range(num_steps):
        for vehicle in vehicles:
            vehicle.move(dt)
            positions[vehicle.id].append(vehicle.position)
                  collisions = [other for other in vehicles if vehicle != other and
vehicle.check_collision(other)]
        if collisions:
             print(f"Collision detected between vehicle {vehicle.id} and: {', '.join(other.id for
other in collisions)}")

    positions_df = pd.DataFrame(positions)
     positions_df.apply(pd.Series.explode).plot.scatter(x=0, y=1, figsize=(10, 5), title="Vehicle
Positions Over Time", legend=True)
    plt.xlabel("X Position")
    plt.ylabel("Y Position")
    plt.show()

# Simulation setup
vehicle1 = Vehicle("V1", 65, (0, 0))
vehicle2 = Vehicle("V2", 50, (10, 20))
vehicle3 = Vehicle("V3", 35, (30, 50))
vehicle4 = Vehicle("V4", 20, (10, 50))
```

VANET Secure Routing Protocol - Full Report

```
# Example use case
message, hashes = vehicle1.generate_message()
vehicle2.receive_message(message, hashes.copy())

tampered_message = message.copy()
tampered_message["speed"] = 100
vehicle2.receive_message(tampered_message, hashes.copy())

simulate([vehicle1, vehicle2], 0.1, 100)
plot_speeds([vehicle1, vehicle2], 100, 1)
plot_positions([vehicle1, vehicle2], 100, 0.1)
```