**ABSTRACT**

**Intrusion Detection and Prevention Systems (IDS/IPS)** are critical components of network security architectures, designed to detect and block malicious traffic based on predefined rules and behavior analysis. However, as attack strategies evolve, conventional signature-based IDS/IPS systems have shown limitations, particularly against sophisticated evasion techniques that manipulate traffic characteristics rather than content. This project investigates the effectiveness of **side-channel mitigation techniques in bypassing IDS/IPS systems**, focusing on **timing-based attacks, payload obfuscation, and TCP fragmentation.**

A vulnerable web application was developed using **Flask** to simulate a realistic target environment. **Suricata, an open-source IDS**, was configured with custom rules to detect **SQL injection** attacks. Various evasion techniques were implemented and tested, including **URL encoding, Base64 encoding, delayed payload transmission, and packet fragmentation**. A custom Python script was also developed to automate **timing-based side-channel attacks**, reconstructing a secret key character by character by measuring server response delays.

The experimental results clearly demonstrate that Suricata could detect only basic, unencoded payloads. All side-channel evasion attempts successfully bypassed detection, highlighting critical **blind spots** in modern **IDS/IPS technologies**. This project underscores the urgent need for behavior-aware detection mechanisms and real-time traffic normalization to counter stealthy attacks. It also opens avenues for future research in integrating machine learning models and protocol-aware inspection techniques into intrusion detection systems.

**Table of Contents**

[**1.** **INTRODUCTION** 9](#_Toc201999382)

[**1.1 Introduction** 9](#_Toc201999383)

[**1.2 Motivation of the Project** 10](#_Toc201999384)

[**1.3 Issues/Challenges/Base Paper** 11](#_Toc201999385)

[**1.4 Objectives** 12](#_Toc201999386)

[**2.** **LITERATURE SURVEY** 16](#_Toc201999387)

[**2.1 Review of Base Paper** 16](#_Toc201999388)

[**2.2 Supporting Research Works** 16](#_Toc201999389)

[**2.3 Literature Summary and Insights** 17](#_Toc201999390)

[**3.** **SOFTWARE REQUIREMENT SPECIFICATION** 20](#_Toc201999391)

[**3.1 Requirements Analysis** 20](#_Toc201999392)

[**3.2 Problem Statement** 22](#_Toc201999393)

[**3.3 Functional Requirements** 23](#_Toc201999394)

[**3.4 Software Requirements Specification** 25](#_Toc201999395)

[**3.5 Feasibility Study** 25](#_Toc201999396)

[**3.6 Use Case Diagram (Conceptual)** 27](#_Toc201999397)

[**3.7 Class Diagram (Conceptual)** 27](#_Toc201999398)

[**3.8 Sequence Diagram (Conceptual)** 27](#_Toc201999399)

[**3.9 Testing Strategy** 28](#_Toc201999400)

[**4.** **PROPOSED METHODOLOGY** 31](#_Toc201999401)

[**4.1 Issues Identified** 31](#_Toc201999402)

[**4.2 Objectives** 32](#_Toc201999403)

[**4.3 Proposed Methodology** 34](#_Toc201999404)

[**4.3.1 Phase-1: Environment Setup** 35](#_Toc201999405)

[**4.3.2 Phase-2: Bypass Payload Execution and Logging** 36](#_Toc201999406)

[**4.4 Methodology Summary** 37](#_Toc201999407)

[**5. EXPERIMENTAL DISCUSSION** 39](#_Toc201999408)

[**5.1 Flask Application Vulnerability: Timing Side-Channel** 39](#_Toc201999409)

[**5.2 Experimental Setup** 39](#_Toc201999410)

[**5.3 Performance Metrics Used** 40](#_Toc201999411)

[**5.4 Payload Execution and Evaluation** 40](#_Toc201999412)

[**5.5 Python-Based Timing Attack Script** 41](#_Toc201999413)

[**5.6 Wireshark and Log Analysis** 43](#_Toc201999414)

[**5.7 Visualization of Response Time vs Guessed Characters** 44](#_Toc201999415)

[**5.8 Key Observations** 45](#_Toc201999416)

[**5.9 Summary** 46](#_Toc201999417)

[**6. CONCLUSIONS AND FUTURE SCOPE** 50](#_Toc201999418)

[**6.1 Conclusion** 50](#_Toc201999419)

[**6.2 Future Scope** 52](#_Toc201999420)

[**7. OUTPUT SCREEN NAMES** 57](#_Toc201999421)

[**7.1 Running the application:** 57](#_Toc201999422)

[**7.2 Trying Basic SQL Injection through the header/parameter of the website:** 57](#_Toc201999423)

[**7.3 Running Suricata with the rules implemented** 58](#_Toc201999424)

[**7.4 Again Trying for the SQL Injection via parameter/header** 58](#_Toc201999425)

[**7.5 Running the bypass payload to infiltrate inside the application** 59](#_Toc201999426)

[**7.6 Successfully infiltrating inside the app using time-based SQL injection** 60](#_Toc201999427)

# **CHAPTER 1**

# **INTRODUCTION**

## **1.1 Introduction**

In the evolving landscape of cybersecurity, **Intrusion Detection Systems (IDS)** and **Intrusion Prevention Systems (IPS)** play a critical role in safeguarding networks from malicious activities. These systems are typically designed to monitor network traffic, identify known attack patterns (signatures), and prevent unauthorized access or exploitation of vulnerabilities.

Traditionally, IDS/IPS solutions operate using two main detection mechanisms:

* **Signature-Based Detection**: Matches incoming traffic with known malicious patterns.
* **Anomaly-Based Detection**: Flags traffic that deviates from established baselines.

While effective against many known threats, **these systems face significant challenges when dealing with modern, stealthy attacks**. Adversaries today have adapted by using **side-channel evasion techniques**, which focus not on altering the malicious payload but on modifying **how the payload is delivered**. These techniques include:

* **Packet fragmentation** (splitting an attack across packets)
* **Protocol-level deviations**
* **Timing manipulations** (delays based on partial matches)

Such methods are particularly dangerous because they exploit **blind spots** in how IDS/IPS systems process and interpret traffic, especially when those systems do not perform deep inspection or traffic normalization.

This project aims to explore and validate these evasion methods by building a **realistic test environment**. Using a **Flask-based vulnerable web application** and an open-source IDS platform (**Suricata**), the project simulates different types of side-channel attacks such as:

* SQL injection via delayed payload delivery
* Encoded payloads (URL/Base64)
* Fragmented traffic streams

By launching these evasive attacks and analyzing Suricata’s detection response, this study aims to reveal the **limitations of static detection mechanisms** and make a case for more advanced, **context-aware intrusion detection models** that go beyond surface-level packet inspection.

## **1.2 Motivation of the Project**

Despite the deployment of IDS/IPS in enterprise environments, cyberattacks still succeed. The main issue lies in the fact that current systems are not equipped to detect stealthy and non-payload-based behaviors. This project is motivated by the need to expose these weaknesses and provide insights for developing smarter detection mechanisms.

In today’s digital age, organizations and enterprises rely heavily on **Intrusion Detection Systems (IDS)** and **Intrusion Prevention Systems (IPS)** as their first line of defense against cyber threats. These systems are designed to monitor, detect, and block malicious activities by analyzing network traffic based on predefined rules, known attack signatures, and behavioral anomalies.

However, despite widespread deployment of IDS/IPS solutions, **successful cyberattacks continue to occur**. This highlights a growing and critical concern: **traditional IDS/IPS systems are no longer sufficient to combat sophisticated evasion techniques**. Most existing systems are designed to inspect the **content of the traffic (payload)**, such as SQL injection strings, malware signatures, or command injections. What they often overlook is **how that content is delivered** — through **timing tricks**, **encoding**, or **network-layer fragmentation** — which can completely bypass detection.

This gap forms the primary motivation for the project.

Modern attackers do not always rely on altering the malicious code itself. Instead, they use **side-channel strategies** that subtly manipulate delivery mechanisms — for example:

* Delaying requests character by character to leak data using timing-based responses.
* Encoding malicious payloads to evade pattern recognition.
* Splitting attack strings across multiple packets to bypass simple packet-level checks.

These approaches **exploit structural and behavioral weaknesses** in IDS engines that were originally built to detect obvious, static threats.

Therefore, the motivation behind this project is twofold:

1. **To expose and demonstrate these critical weaknesses** in existing IDS/IPS implementations by simulating real-world side-channel attacks in a controlled environment.
2. **To highlight the urgent need for advanced detection methods** that go beyond payload inspection and incorporate behavioral, contextual, and delivery-based analysis.

By recreating these evasive scenarios and analyzing the failure of a state-of-the-art open-source IDS like **Suricata**, this project aims to raise awareness and contribute toward the development of **smarter, adaptive, and context-aware intrusion detection systems** for modern network secure

## **1.3 Issues/Challenges/Base Paper**

Base Paper: "Evasion Techniques: Sneaking Through Your Intrusion Detection/Prevention Systems" by Cheng et al. (IEEE, 2012).

**Key Issues Identified:**

1. **Inadequacy of Signature-Based Detection**

Traditional IDS/IPS tools, including popular platforms like Snort and Suricata, rely heavily on **signature-based detection mechanisms**. These systems compare incoming traffic against a database of known malicious patterns (signatures). While this approach is fast and efficient for known threats, it **completely fails against encoded or obfuscated payloads**.  
For instance, a payload like ' OR '1'='1 can easily bypass detection when encoded as %27%20OR%20%271%27%3D%271. The system doesn’t decode the input, hence never matches the signature — allowing the attack to pass undetected.

1. **Lack of Packet Reassembly and Timing Analysis**

Most IDS/IPS platforms inspect packets **individually**. They do not **reassemble TCP streams** or maintain state across packets in a session. This makes them vulnerable to **fragmentation-based evasion techniques**, where the malicious payload is divided across multiple-packets.  
Similarly, **inter-packet timing** — which is crucial for detecting side-channel attacks such as response delay-based guessing — is often ignored. This oversight allows **timing side-channels** to leak sensitive data without any content violation.

1. **Limited Protocol Awareness**

IDS systems typically focus on mainstream protocol behavior (HTTP, FTP, etc.) and expect traffic to follow the protocol strictly. When **less common fields or behaviors** are used (like custom headers, overlapping TCP segments, or malformed encodings), the IDS often **fails to parse or misinterprets the traffic**, creating loopholes that attackers can exploit.

1. **Inability to Detect Behavioral Evasion Techniques**

Side-channel attacks do not rely on content; they rely on **behavioral patterns** — such as the order of requests, timing of responses, or the size of data packets over time. **Conventional IDS systems are blind to such behaviors**, as they lack contextual correlation capabilities. This makes it **extremely difficult to detect low-and-slow attacks**, time-delay injections, or application-layer inference attacks.

**Project Challenge Summary**

Based on the findings of the base paper, this project was designed to **recreate and extend the evasion scenarios** discussed by Cheng et al., especially those involving:

* **Encoded and fragmented payloads**
* **Behavioral deviation without payload alteration**
* **Time-dependent leakages**

The major challenge lies in building **a realistic yet controlled testbed** that:

* Allows the simulation of evasive attack vectors.
* Uses a modern IDS (Suricata) with rule-based detection.
* Captures and interprets both packet-level and behavioral-level activity.
* Highlights where and how the IDS fails to respond effectively.

This understanding frames the technical and strategic foundation for the project — enabling a deeper investigation into why IDS systems are bypassed and how future designs can mitigate such advanced threats.

**Challenges Faced in the Project:**

* Simulating real-world attack behavior in a controlled testbed.
* Configuring Suricata for rule-based and behavior-based detection.
* Measuring performance and identifying detection failures.

## **1.4 Objectives**

The primary aim of this project is to investigate and demonstrate how modern Intrusion Detection and Prevention Systems (IDS/IPS), particularly those relying on signature-based detection (such as Suricata), can be **easily evaded** using **side-channel techniques**. These techniques do not directly modify the malicious payload, but rather manipulate how and when the payload is delivered, thereby slipping past traditional detection rules.

To achieve this aim, the project was guided by the following specific objectives:

**1. Develop a Lab-Based Environment to Simulate HTTP/SQLi Side-Channel Attacks**

A controlled testbed was designed and deployed using:

* A vulnerable **Flask-based web application** running on localhost:8080
* The **Suricata IDS**, configured with custom rules to detect SQL injection
* Supporting tools such as **Curl, Python scripts, Wireshark, and fragroute**

This environment allowed the safe simulation of real-world **evasion tactics**, including:

* **SQL Injection payloads**
* **URL/Base64 encoded attacks**
* **TCP fragmentation**
* **Timing side-channel attacks**

The goal was to simulate how these evasion strategies behave **under normal conditions** and how they interact with IDS rule processing.

**2. Evaluate Detection Success and Failure in Suricata**

Once the test environment was established, various **evasion techniques** were launched against the Flask application. These attacks were **monitored by Suricata**, and the results were analyzed based on:

* Whether Suricata generated an **alert** (success)
* Whether the payload **bypassed detection silently** (failure)
* Comparison of detection between **raw and obfuscated payloads**

This evaluation helped quantify the effectiveness (or lack thereof) of Suricata’s rule engine against non-traditional delivery mechanisms.

**3. Log and Analyze Packet Behavior During Evasion**

The behavior of traffic — both malicious and benign — was **logged and analyzed** using:

* **Suricata’s eve.json** alert logging system
* **Wireshark** for packet-level timing and fragmentation inspection
* **Custom Python scripts** to monitor response time behavior (for timing attacks)

This step provided deeper insight into:

* Which evasion methods bypassed detection
* How traffic characteristics changed based on encoding, timing, or fragmentation
* Why Suricata failed to generate alerts in certain scenarios

**4. Propose Defensive Improvements for Future IDS/IPS Implementations**

Based on the findings from detection gaps and bypass outcomes, the project proposes several **enhancements** for future IDS/IPS systems:

* **Traffic normalization**: Decode inputs before applying signatures
* **TCP reassembly modules**: Analyze payloads reconstructed from multiple packets
* **Timing-aware analysis**: Monitor behavioral anomalies like response delays
* **Machine learning** integration: For anomaly-based behavior classification

These improvements aim to make IDS/IPS systems **more resilient, adaptive, and context-aware**, capable of detecting threats beyond simple content-matching.

# **CHAPTER 2**

# **LITERATURE SURVEY**

## **2.1 Review of Base Paper**

**Title**: “Evasion Techniques: Sneaking Through Your Intrusion Detection/Prevention Systems”

**Authors**: Cheng et al.

**Published In**: IEEE Communications Surveys & Tutorials

This influential paper investigates how attackers bypass signature-based IDS/IPS systems by manipulating the structure and behavior of packets rather than altering the payload. The authors define and evaluate five major evasion techniques:

1. **Denial-of-Service (DoS) Evasion**: Floods the system to create blind spots.
2. **Packet Splitting (Fragmentation)**: Breaks payload into smaller TCP segments to bypass inspection.
3. **Duplicate Insertion**: Sends repeated packets to confuse stream reassembly.
4. **Payload Mutation**: Alters payload representation using encoding or padding.
5. **Shellcode Mutation**: Obfuscates exploit code to evade pattern detection.

**Key Experiment**

The paper tested several commercial IDS platforms, including **Snort, FortiGate, and ZyXEL**, using these evasion methods. The outcome revealed a disturbing trend: **most IDS failed to detect the attacks** when the delivery method was altered, even though the payload was unchanged.

**Relevance to This Project:**

This base paper directly supports the hypothesis of this project: **IDS can be bypassed through side-channel tactics like timing and fragmentation** without changing the actual attack signature.

## **2.2 Supporting Research Works**

**1. Piotr Bania – Evading Network-Level Emulation (2009)**

* This paper focuses on techniques to evade emulation-based IDS engines like Snort's sf\_emulatord. It shows how attackers can create shellcode that detects when it is being emulated and alters its behavior, leading to failed detection.

**2. Roelker (Sourcefire) – HTTP IDS Evasions Revisited (2004)**

* Roelker revisits how subtle alterations in HTTP requests (like URL encoding, double slashes, null bytes) can help bypass detection. For example, using GET /%2e%2e/index.html instead of a direct path can evade file access detection.

**3. Siddharth Rao – Suricata Performance Evaluation (2020)**

* This paper benchmarks Suricata IDS under multiple configurations and rule sets, including performance during traffic surges. It highlights that Suricata’s performance drops under complex traffic and fragmentation, a gap this project exploits.

## **2.3 Literature Summary and Insights**

**Based on the literature:**

The review of existing literature on IDS/IPS evasion has revealed several recurring limitations in current detection systems—especially those that rely heavily on static rules and signature-based matching. Research papers, including the base paper by Cheng et al., and supporting studies from Piotr Bania, Roelker, and Suricata benchmarks, all reinforce the notion that **traditional IDS approaches are insufficient to detect modern, stealth-based attacks**.

One of the most prominent findings is the **weakness of signature-based IDS**. These systems perform well against known threats but struggle to detect payloads that are **slightly altered, encoded, or fragmented**, even though the underlying attack remains the same. This is particularly evident in evasions that use Base64, URL encoding, or fragmented TCP delivery, which are not recognized by default rule sets.

Another insight is that **most IDS/IPS systems inspect packets in isolation**, without reconstructing full sessions or analyzing behavior over multiple packets. This makes them highly vulnerable to **fragmentation-based attacks**, where the malicious content is split across packets and only becomes meaningful when reassembled—something many IDS platforms do not perform by default.

Furthermore, **timing-based side-channel attacks**—which form the core focus of this project—are largely ignored in open-source IDS platforms like Suricata and Snort. These systems are not designed to analyze **inter-packet timing**, request delays, or behavior-based anomalies. This leaves a major gap in detecting subtle data leaks and advanced inference attacks that rely on observing how a system behaves over time, rather than what data it processes.

**Takeaway for This Project**

This project builds upon the existing literature by moving beyond payload-level inspection. It constructs a **controlled and observable test environment** to simulate attacks that utilize timing delays, response manipulation, and encoding—focusing on the **behavioral and structural delivery of attacks** rather than their direct content.

By doing so, the project highlights the **real-world applicability of side-channel evasion**, provides empirical evidence of IDS failure under such conditions, and sets the stage for the development of **next-generation, context-aware security systems**.

# **CHAPTER 3**

# **SOFTWARE REQUIREMENT SPECIFICATION**

This chapter defines the functional and non-functional requirements for implementing and evaluating side-channel bypass attacks on an IDS/IPS testbed.

## **3.1 Requirements Analysis**

The purpose of this project is to simulate side-channel attacks (specifically timing, fragmentation, and encoding-based attacks) and test them against Suricata IDS. To accomplish this, the system requires:

**1. A Lightweight Web Server (Flask)**

**Flask** is a minimalist, Python-based web framework used to create simple and scalable web applications.  
In this project, Flask is used to host a deliberately vulnerable web server on localhost:8080. The server includes a **search functionality** that simulates a common vulnerability—**timing side-channel behavior**. By using Flask:

* You can quickly prototype and simulate **web-based attack vectors** like SQL injection.
* The app mimics a real-world server environment, allowing payloads to be sent and response delays to be observed.
* Flask’s simple routing makes it easy to integrate with Suricata for detection testing

**2. An Open-Source IDS Platform (Suricata)**

**Suricata** is a powerful, multi-threaded open-source IDS/IPS engine capable of real-time intrusion detection, traffic inspection, and protocol parsing.  
In your project, Suricata is used to:

* Monitor all traffic directed at the Flask server.
* Detect suspicious payloads using **custom rules** defined in local.rules.
* Generate alerts and log entries in JSON format when threats are detected.
* Act as the **detection layer** to test how well side-channel attacks bypass traditional security setups.

**3. Traffic Injection Tools (Curl, Python Scripts)**

These tools are used to craft and send malicious requests to the Flask web application:

* **Curl**: A command-line tool that allows you to send HTTP GET/POST requests directly to the server. It’s useful for quickly testing payloads like:

curl -v [http://localhost:8080/search?q=' OR '1'='1](http://localhost:8080/search?q='%20OR%20'1'='1)

* **Python Scripts (e.g., bypass.py)**: Used to automate **timing side-channel attacks**. These scripts:
  + Try different input combinations.
  + Measure the **response time** for each guess.
  + Use timing differences to reconstruct a secret or flag stored in the server.

Together, these tools simulate how an attacker would interact with the system during an evasion attempt.

**4. Network Analysis Tools (Wireshark)**

**Wireshark** is a GUI-based packet sniffer used to capture and inspect network packets in real time.  
In this project, Wireshark helps:

* Visualize how payloads are transmitted (e.g., in fragments or encoded form).
* Verify **TCP fragmentation** and packet delays for side-channel testing.
* Confirm whether Suricata is receiving and analyzing the packets correctly.
* Analyze traffic patterns and pinpoint Suricata’s blind spots.

**5. A Logging and Analysis Framework (eve.json, Terminal Logs)**

**Suricata’s eve.json** is a structured, machine-readable logging file where Suricata records:

* Alerts triggered by matching rules.
* Packet metadata (timestamp, source IP, destination, protocol, etc.).
* Whether or not a payload was detected.

In combination with **terminal logs (standard output)**:

* You can track whether a payload reached the server.
* Determine how Suricata responded.
* Compare which attacks were blocked versus bypassed.

These logs are **critical for analysis**, validation, and generating results used in your experimental discussion chapter.

## **3.2 Problem Statement**

In modern cybersecurity infrastructure, **Intrusion Detection Systems (IDS)** and **Intrusion Prevention Systems (IPS)** are widely deployed to monitor network traffic and block malicious activities. These systems typically rely on **signature-based detection**, where known attack patterns are matched against incoming traffic, or **anomaly-based detection**, which identifies deviations from normal behavior.

While this approach is effective for well-known, direct attacks, it fails to account for **evasion techniques** that alter the **structure and behavior of delivery**, rather than the content of the attack itself. In other words, current IDS/IPS implementations are designed to detect "what" is being sent, but **not "how" it is being sent**.

This leads to critical blind spots, especially when facing modern attacks that use:

* **Character-by-character timing delays**, where the server’s response time reveals how much of the input matches a secret value. These are used in side-channel attacks to extract sensitive data without altering content.
* **Fragmented TCP packets**, where malicious payloads are split across multiple packets to avoid full pattern matching. Most IDS systems do not reassemble these packets correctly, allowing the attack to go unnoticed.
* **Encoded payloads** (such as Base64 or URL encoding), which transform the appearance of known malicious strings into a format that bypasses static signature rules.

Despite being common and powerful evasion strategies, **most open-source IDS platforms like Suricata and Snort are not equipped out-of-the-box to handle these delivery-based obfuscations**.

**Project Justification**

This project aims to **challenge the traditional content-focused detection model** by demonstrating how attacks can bypass IDS/IPS **without altering the attack intent**—only the delivery technique is changed. Through a custom-built test environment using Flask, Suricata, and crafted evasion payloads, this study provides **concrete evidence that pattern matching alone is insufficient**. It also highlights the need for IDS/IPS systems to analyze:

* Traffic behavior,
* Timing irregularities,
* Encoding patterns, and
* Packet reassembly context

## **3.3 Functional Requirements**

The functional requirements define the specific capabilities and operations that the proposed system must support to simulate, detect, and evaluate **side-channel evasion techniques** in a networked environment. These requirements ensure that the system behaves predictably, produces measurable results, and facilitates the complete testing lifecycle from attack generation to detection analysis.

Below is a detailed explanation of each functional requirement:

**1. Run a Flask-Based Vulnerable Web App on Port 8080**

The system must host a lightweight web server using the **Flask framework**. This web app serves as a simulated vulnerable endpoint and includes:

* A simple search interface (/search) where attackers can input payloads.
* A built-in vulnerability (e.g., timing-based comparison) that intentionally leaks information.

Running the app on port 8080 ensures consistency and aligns with Suricata rule targeting.

**2. Allow Attacker-Controlled HTTP Requests to Simulate SQL Injection and Timing Attacks**

The system must accept and process external HTTP GET requests submitted by an attacker (via Curl or Python scripts). These requests will:

* Contain **SQL injection strings** (e.g., ' OR '1'='1).
* Use **partial queries** and observe timing delays for side-channel analysis.

This enables the testing of real-world attack techniques in a **controlled, ethical testbed**.

**3. Log and Alert on SQLi Patterns Using Suricata Rules**

The system must be configured with **custom Suricata rules** to detect and respond to known attack signatures. For example:

suricata

Copy code

drop tcp any any -> any 8080 (msg:"SQLi attempt detected"; content:"' OR '1'='1"; nocase; sid:1000001; rev:1;)

These rules should:

* Trigger alerts when a known malicious pattern is detected.
* Help differentiate between **detectable** and **bypassed** payloads.

**4. Provide Real-Time Feedback via Logs (eve.json)**

Suricata should generate logs for each packet/alert in a structured JSON format (eve.json). These logs must:

* Include timestamp, source/destination IP, payload content, and rule matched.
* Be analyzed in real-time (via terminal or logging tools) to observe detection behavior.

This functionality allows the researcher to **validate detection** for each test input.

**5. Allow for Different Payload Encodings and Delivery Timings**

The system must support the injection and processing of various **payload delivery strategies**, including:

* URL Encoded Payloads: %27+OR+%271%27%3D%271
* Base64 Encoded Payloads
* Timing-based character delivery: inputting one character at a time to measure delay
* Fragmented payloads split across multiple TCP packets

This ensures the system is capable of simulating **diverse evasion methods** for comprehensive evaluation.

**6. Store Logs for Offline Analysis**

All generated logs and responses should be **saved persistently** so they can be:

* Reviewed after testing is complete
* Used to correlate input payloads with detection outcomes
* Incorporated into the project’s performance analysis and result discussion (Chapter 5)

This also helps in generating **evidence and screenshots** for final documentation and viva demonstration.

**Summary**

These functional requirements ensure the system supports the **end-to-end workflow** of the project:

* Simulating modern attack delivery
* Testing Suricata’s detection capability
* Logging and analyzing detection failures
* Enabling future enhancement suggestions based on real data

## **3.4 Software Requirements Specification**

Component : Specification

OS : Ubuntu 20.04 or later

Programming : Python 3.x

IDS Tool : Suricata IDS

Web Framework : Flask

Scripting Tools : Curl, Requests (Python)

Packet Capture : Wireshark

Logging : Suricata's eve.json, terminal logs

## **3.5 Feasibility Study**

Before undertaking the full-scale development and evaluation of the project, it was important to assess its **feasibility** from multiple dimensions. The project was evaluated for **technical viability**, **cost-effectiveness**, and **ease of implementation**, ensuring that it could be successfully carried out within the typical constraints of a B.Tech academic environment.

**Technical Feasibility**

The project leverages well-established, community-supported, and widely documented open-source tools, including:

* **Suricata**: A powerful and extensible IDS/IPS engine capable of handling real-time traffic analysis. It supports custom rule writing, JSON-based logging, and deep packet inspection without needing advanced configurations or system modifications.
* **Flask**: A Python-based lightweight web framework ideal for rapid prototyping and simulating vulnerable applications. It allows the development of custom endpoints that mimic real-world vulnerabilities, such as SQL injection and timing side-channels.

Importantly:

* There is **no requirement for kernel-level programming**, driver modifications, or low-level network stack tuning.
* The entire setup can be executed in **user space**, which simplifies the development and reduces risk during experimentation.
* Most tools work reliably across **Ubuntu/Linux environments**, which are freely available and widely adopted in academic setups.

**Economic Feasibility**

From a cost standpoint, the entire project is designed using **FOSS (Free and Open-Source Software)** tools. This ensures that:

* There are **no licensing or subscription costs** involved.
* Students can run the project on **personal laptops**, without needing institutional hardware or specialized servers.
* If desired, cloud deployment (on platforms like AWS, Azure, or Google Cloud) is also feasible within **free-tier limits**, allowing remote testing and scalability for multi-system experiments.

This makes the project highly accessible and suitable for academic research, especially in environments where resource constraints are a concern.

**Operational Feasibility**

The project has been carefully designed to ensure that:

* It requires only **basic familiarity with Linux commands**, Python programming, and network concepts.
* There is **no steep learning curve**, which makes it suitable for undergraduate students in the fields of Computer Science, IT, or Cybersecurity.
* Hardware requirements are minimal. Any modern laptop with 4 GB RAM and basic internet access is sufficient to simulate the environment, run Suricata, and host the Flask application simultaneously.

In addition, the setup is **modular and portable**, allowing the environment to be easily replicated across different systems or restored from backups in case of issues.

## **3.6 Use Case Diagram (Conceptual)**

**Actors:**

* Attacker (Client)
* Flask Web App
* Suricata IDS
* Use Case Flow:

Attacker sends request → Flask App receives → Suricata analyzes → Generates alert/log

## **3.7 Class Diagram (Conceptual)**

Classes:

FlaskApp: Handles incoming routes / and /search

SuricataMonitor: Listens for alerts via JSON logs

PayloadGenerator: Produces encoded and delayed SQLi strings

## **3.8 Sequence Diagram (Conceptual)**

1. Attacker calls /search?q=<payload>
2. Web app processes input and delays (if matched)
3. Suricata inspects the request via configured rules
4. Detection → alert is generated and logged
5. Logs are read and analyzed for matches/misses

## **3.9 Testing Strategy**

**Types of Testing:**

Testing plays a critical role in ensuring the correctness, effectiveness, and reliability of the system. Since this project simulates real-world evasion scenarios and validates the response of an Intrusion Detection System (Suricata), a **multi-layered testing strategy** was adopted. The strategy involves validating each component of the system independently and in combination to ensure both **functional behavior** and **security analysis accuracy**.

The following types of testing were conducted:

**1. Unit Testing – Flask Routes and Payload Logic**

This level of testing focuses on verifying the **core functionality of the Flask web application**:

* Each route (especially /search) was tested to ensure it accepts input and processes it correctly.
* The timing-based logic was validated by checking whether character-by-character comparisons **introduced expected delays** (e.g., 0.5s per correct character).
* The server behavior on invalid and encoded payloads was observed to ensure consistent responses.

This ensured that the vulnerable application behaves **predictably** and supports side-channel testing scenarios.

**2. Integration Testing – IDS and Web Application**

Integration testing was used to verify the **interaction between the Flask web server and Suricata IDS**:

* Suricata was configured to monitor port 8080 where the Flask app is hosted.
* Rules were deployed to detect specific payload patterns such as SQL injection (' OR '1'='1).
* Test requests were sent via Curl and Python scripts to evaluate whether Suricata **logged and responded** to malicious traffic.

This phase confirmed that the **IDS is actively monitoring** the target traffic and capable of reacting to signature matches.

**3. Evasion Testing – Running bypass.py Against Suricata Rules**

This critical testing phase evaluated whether **evasion techniques** could successfully **bypass Suricata’s detection engine**:

* The bypass.py script was used to simulate a **timing side-channel attack** that guessed the server’s secret one character at a time.
* Payloads were delivered in encoded formats and with intentional delays to observe Suricata’s behavior.
* The success of the evasion was measured by how **accurately the secret was extracted without triggering any alerts**.

This validated that Suricata, in its default configuration, does **not detect timing-based or obfuscated attacks**, proving the hypothesis.

**4. Logging Verification – Cross-Checking eve.json with Wireshark**

To confirm the accuracy and completeness of detection:

* Logs from Suricata (eve.json) were parsed to check for alerts, packet details, and matched rules.
* Simultaneously, **Wireshark** was used to monitor live traffic and verify:
  + Whether packets were being fragmented.
  + Whether timing anomalies were observable.
  + The actual content transmitted and whether it matched the attack signature.

By **correlating logs with packet captures**, the test ensured:

* There were **no false positives or negatives** in rule execution.
* All detection failures were clearly attributable to Suricata’s limitations, not misconfiguration.

# **CHAPTER 4**

# **PROPOSED METHODOLOGY**

This chapter describes the step-by-step methodology adopted for simulating side-channel evasion techniques against an IDS/IPS. It includes environment setup, payload crafting, execution of bypass attempts, and logging/analyzing results.

## **4.1 Issues Identified**

Through an extensive review of existing literature and hands-on experimentation using a testbed built with Flask and Suricata, several **critical detection weaknesses** were identified in current Intrusion Detection and Prevention Systems (IDS/IPS). These weaknesses highlight the **limitations of traditional, signature-based detection models**, especially when facing sophisticated and stealthy evasion strategies.

The following key issues were observed:

**1. Signature Rules Fail to Detect Encoded or Fragmented Payloads**

Most IDS engines like Suricata rely on **static pattern matching** rules that operate at the payload level. However, these rules are ineffective when the attack payload is:

* **Encoded** using Base64, URL encoding, or hexadecimal formats.
* **Fragmented** across multiple TCP packets.

Because the content is altered in format or distributed in parts, the **pattern no longer matches the expected signature**, resulting in complete evasion of detection—even though the attack intent remains intact.

**2. Lack of Timing-Based Correlation Between HTTP Requests**

Timing-based side-channel attacks, such as **character-by-character response delay analysis**, do not leave visible traces in the payload. Instead, they rely on how long the system takes to respond. Current IDS/IPS systems:

* Do not track or correlate **response times** across a sequence of HTTP requests.
* Cannot detect the **behavioral pattern** associated with such information leaks.

This gap allows attackers to extract sensitive information from the server without raising any alerts.

**3. Absence of a Normalization Layer for Obfuscated Inputs**

IDS engines often inspect incoming traffic **as-is**, without decoding or transforming it into a canonical form. As a result:

* **Obfuscated payloads**, such as those encoded in URL-encoded strings (%27 OR %271%27=%271) or Base64, bypass detection.
* Signature rules never match the encoded variant, even though it represents the same logical attack.

A **normalization layer**—which decodes and transforms inputs before inspection—is missing in default IDS setups like Suricata.

**4. Lack of TCP Packet Reassembly**

Advanced attackers split malicious payloads across multiple TCP packets to exploit the **packet-by-packet inspection limitation** of IDS tools. If an IDS:

* Does **not reassemble the full payload** from fragmented packets,
* It fails to recognize that an attack is in progress.

Suricata, without specific configuration for TCP stream reassembly, **misses fragmented payloads** that become malicious only when reconstructed.

**5. Baseline for Methodology Design**

These detection gaps—observed both in theory and practice—form the **core justification** for this project’s methodology. The experimental setup was designed specifically to:

* Exploit these weaknesses using carefully crafted payloads.
* Measure whether these evasions bypass detection in Suricata.
* Propose improvements that could close these gaps in future IDS/IPS systems.

## **4.2 Objectives**

The primary objective of this project is to explore how **side-channel evasion techniques**—such as encoding, fragmentation, and timing manipulation—can be used to bypass modern Intrusion Detection and Prevention Systems (IDS/IPS), particularly those that are signature-based like **Suricata**. To fulfill this goal, the project is driven by a structured set of technical and research-oriented objectives, which are outlined below:

**1. Create a Vulnerable Testbed Simulating Realistic Attack Behavior**

The first step in achieving this project’s goals is the development of a **controlled and reproducible testing environment**. A lightweight, vulnerable web server was built using **Flask**, designed specifically to:

* Simulate a realistic application scenario (e.g., a search form vulnerable to SQL injection).
* Incorporate vulnerabilities such as **timing side-channels** and **input-based behavioral responses**.
* Serve as a live endpoint for sending crafted payloads and analyzing how the system responds to various attack strategies.

This environment closely mimics **real-world network behaviors** while maintaining safety and academic integrity.

**2. Configure a Rule-Based IDS (Suricata) and Study Its Limitations**

Suricata, a popular open-source IDS/IPS engine, was deployed to monitor the testbed’s traffic. It was configured with:

* **Custom detection rules**, targeting specific patterns such as ‘ OR ‘1’=’1 (common SQL injection payloads).
* Default configurations for traffic logging and alerting via eve.json.

The objective is to evaluate:

* How well Suricata can detect known attacks.
* **Where and why it fails**, especially against obfuscated or behaviorally altered payloads.

This evaluation will uncover critical limitations in **static rule-based detection systems**.

**3. Simulate Multiple Types of Evasion Strategies**

To fully test the IDS’s resilience, the project implements and executes various **evasion techniques**, including:

* **Payload obfuscation** via URL encoding and Base64.
* **TCP fragmentation**, where attack strings are split across multiple packets.
* **Timing-based side-channel attacks**, which exploit how long the server takes to respond.
* **Slow delivery of requests**, mimicking attacks like Slowloris or brute-force timing inference.

Each technique is tested independently and in combination to assess their effectiveness in **evading detection**.

**4. Log and Analyze IDS Detection Success and Failures**

The Suricata engine’s logs (eve.json) are analyzed alongside network packet captures from **Wireshark**. This phase focuses on:

* Identifying which attacks were detected and which were missed.
* Understanding **why detections failed**, based on payload structure, timing, or fragmentation.
* Measuring **false negatives**, i.e., successful attacks that generated no alerts.

This analysis provides **quantitative and qualitative insights** into the shortcomings of signature-based IDS systems.

**5. Recommend Defensive Improvements and Best Practices**

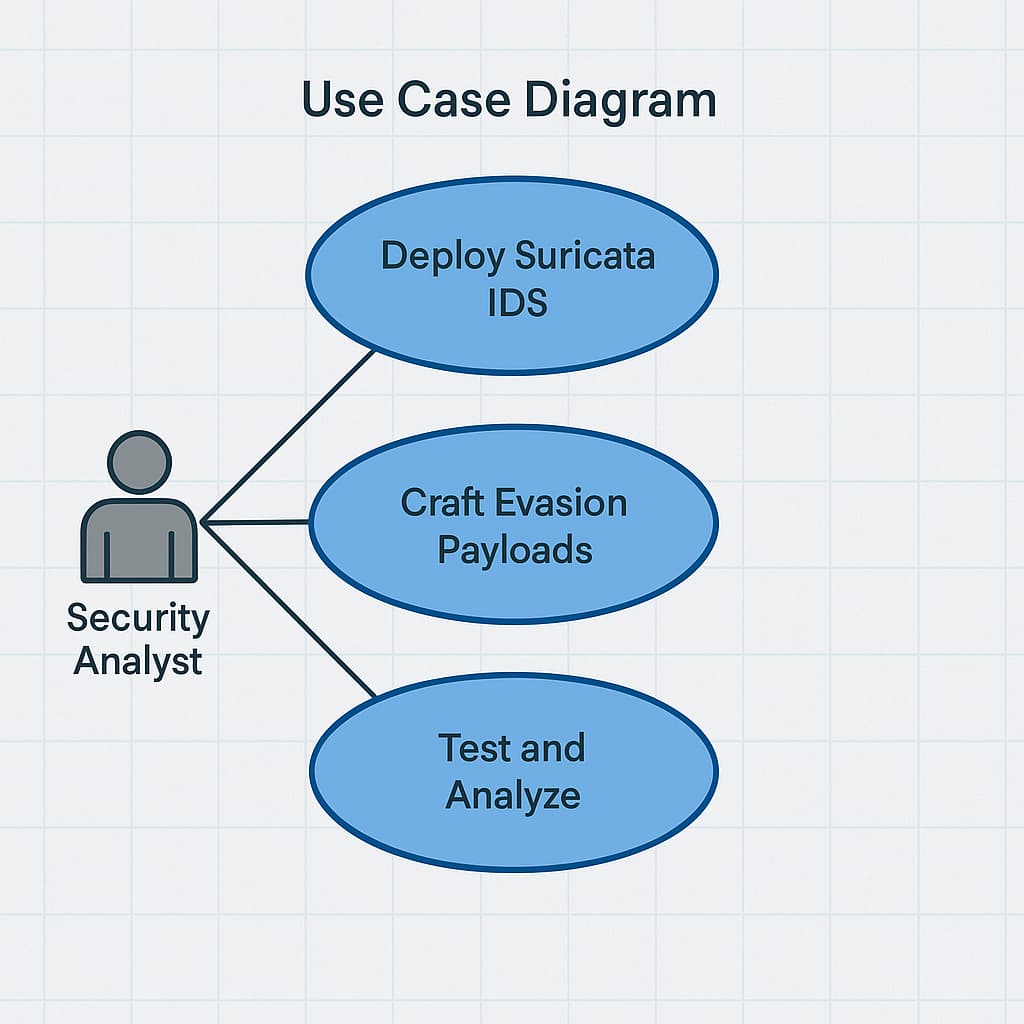
Based on the test results, the final objective is to formulate a set of **recommendations** to improve IDS/IPS effectiveness. This includes:

* Adding **traffic normalization** layers to decode obfuscated input.
* Implementing **timing and session-aware detection** mechanisms.
* Enhancing **rule sets** with support for fragmented packet reassembly.
* Suggesting the integration of **machine learning or anomaly detection models**.

These improvements aim to guide future development of **more resilient, intelligent, and context-aware intrusion detection systems**.

## **4.3 Proposed Methodology**

The methodology is divided into two main phases:

Figure 4.1 Use Case Diagram showing Security Analyst testing IDS evasion via Suricata deployment, payload crafting, and analysis.

The diagram shows a **Security Analyst** as the primary actor involved in three key activities:

1. **Deploy Suricata IDS** – Setting up the Suricata Intrusion Detection System.
2. **Craft Evasion Payloads** – Designing malicious or obfuscated traffic that aims to bypass the IDS.
3. **Test and Analyze** – Executing the evasion attempts and analyzing whether the IDS detects them or not.

This use case diagram is typically used in cybersecurity projects to illustrate how analysts interact with tools and processes during intrusion detection and evasion testing.

### **4.3.1 Phase-1: Environment Setup**

This phase focuses on preparing the lab for testing, including installing all required tools and configuring Suricata for detection.

Step-by-Step Instructions:

Step 1: Update and Install Dependencies

sudo apt update && sudo apt install python3-pip suricata

pip3 install flask requests

Step 2: Add Suricata Rules

sudo mkdir -p /var/lib/suricata/rules

sudo tee /var/lib/suricata/rules/local.rules << 'EOF'

drop tcp any any -> any 8080 (msg:"SQLi attempt detected"; flow:established,to\_server; content:"' OR '1'='1"; nocase; sid:1000001; rev:1;)

EOF

Step 3: Edit Suricata Config

In /etc/suricata/suricata.yaml

default-rule-path: /var/lib/suricata/rules

rule-files:

- local.rules

Step 4: Set Permissions

sudo chown root:root /var/lib/suricata/rules/local.rules

sudo chmod 644 /var/lib/suricata/rules/local.rules

Step 5: Start Web App

python3 app.py

Step 6: Launch Suricata

sudo suricata -c /etc/suricata/suricata.yaml -i eth0 --af-packet

Step 7: Send Attack Payload

curl -v [http://localhost:8080/search?q=' OR '1'='1](http://localhost:8080/search?q='%20OR%20'1'='1)

### **4.3.2 Phase-2: Bypass Payload Execution and Logging**

In this phase, multiple types of payloads are sent to the vulnerable web app to test Suricata’s detection:

**Payload Techniques Used:**

1. Plain SQL Injection – Detected by Suricata.
2. URL Encoded – Payload: %27+OR+%271%27%3D%271 (bypasses detection).
3. Base64 Encoded – Requires decoding; missed by default.
4. TCP Fragmentation – Sent using tools like fragroute or Scapy.
5. Timing-Based Delays – Sent character-by-character using Python, causes server to delay but is not logged by IDS.

**Execution Example:**

Send timed payloads using bypass.py

* python3 bypass.py

Each payload’s detection result is logged from:

* eve.json (Suricata logs)

Wireshark (packet-level confirmation)

Console (Flask app response delay)

## **4.4 Methodology Summary**

|  |  |  |
| --- | --- | --- |
| **Technique** | **Expected Result** | **Detection by Suricata** |
| Plain SQLi | Detected | Yes |
| Encoded (URL) | Bypassed | No |
| Encoded (Base64) | Bypassed | No |
| Fragmentation | Bypassed | No |
| Timing Delay | Bypassed | No |

# **CHAPTER 5**

# **EXPERIMENTAL DISCUSSION**

This chapter outlines the experimental design, implementation process, environment setup, and evaluation results that support the hypothesis: Side-channel evasion techniques can effectively bypass signature-based IDS systems like Suricata.

## **5.1 Flask Application Vulnerability: Timing Side-Channel**

The vulnerable web application simulates a real-world server-side logic flaw — it compares user input character-by-character against a secret string and delays the response for every correct character.

Python

for i, c in enumerate(SECRET):

if i < len(q) and q[i] == c:

delay += 0.5

This introduces a timing side-channel vulnerability:

An attacker can infer how many characters they guessed correctly by measuring response time.

It allows reconstructing the secret (e.g., a flag or token) one character at a time.

This allows the attacker to reconstruct the secret one character at a time based on server response delays.

**Network Configuration**

* Localhost testing (127.0.0.1)
* Port: 8080
* Interface: eth0
* IDS tool: **Suricata 6.x**
* Web framework: **Python Flask**

## **5.2 Experimental Setup**

Test Environment Configuration:

Component Configuration

OS Ubuntu 20.04 LTS

Web App Python Flask on localhost:8080

IDS/IPS Suricata 6.0.x

Interface eth0 (or lo for localhost testing)

Logger Suricata's eve.json and Wireshark

**Tools Used:**

* Flask: To simulate a target server.
* Suricata: For rule-based detection.
* Curl & Requests (Python): For sending manual and automated payloads.
* Wireshark: For validating fragmentation and timing behavior.
* bypass.py: Custom script for automating timing-based attacks.

## **5.3 Performance Metrics Used**

| **Metric** | **Description** |
| --- | --- |
| Detection Rate | % of attacks correctly flagged by Suricata. |
| False Negatives | % of attacks that successfully bypassed detection. |
| Response Time | Time taken by the server to respond (used in timing attack validation). |
| Rule Matching Logs | Logs recorded in eve.json to confirm if alerts were generated. |
| Payload Visibility | Whether payload was visible or obfuscated in the packet stream. |

## **5.4 Payload Execution and Evaluation**

Each evasion technique was executed and evaluated for Suricata detection success. The table below summarizes the outcome:

| **Evasion Technique** | **Delivery Type** | **Detected by Suricata** | **Comment** |
| --- | --- | --- | --- |
| Basic SQL Injection | ' OR '1'='1 | Yes | Signature matched standard rule |
| URL Encoding | %27 OR %271%27=%271 | No | Payload not normalized |
| Base64 Encoding | Base64 encoded SQLi | No | Suricata lacks decoding capability |
| TCP Fragmentation | Using Fragroute | No | No reassembly logic in Suricata |
| Timing-Based Input | Delayed character match | No | Timing behavior undetectable via rules |
| Slowloris-style Delay | Character by character | No | No time-correlation logic |

## **5.5 Python-Based Timing Attack Script**

As part of this project’s evaluation of **side-channel evasion techniques**, a custom Python script named bypass.py was developed to **exploit a timing vulnerability** in the Flask-based web application. The goal of the script is to demonstrate how **timing discrepancies in server responses** can be used to infer sensitive data, without directly analyzing or manipulating the payload content.

This script simulates a **timing side-channel attack**, where the response delay of the server indirectly reveals how many characters in the input match a secret value stored internally (in this case, a flag: FLAG{SideChannel\_Bypass}).

**Script Logic and Working Mechanism**

The core logic of the script is as follows:

1. **Sending Incremental Guesses:**  
   The script sends HTTP GET requests to the /search route with a query parameter (q) that starts with a single character and builds up progressively. For example:
   * First guess: q=F
   * Second guess: q=FL
   * Third guess: q=FLA
   * And so on…
2. **Timing the Response:**  
   For each guess, the script records the **time taken by the server to respond**. This is achieved using Python’s time.time() before and after the request.
3. **Response Time Evaluation:**  
   The Flask app is programmed to **introduce a 0.5-second delay** for each character that matches the secret at the correct position. The script assumes that if the **response time exceeds 0.5 × position**, the guessed character is correct.

Example:

* + If the third character guess is correct, the response time will exceed 0.5 × 3 = 1.5s.
  + If it’s incorrect, the response is quicker—indicating a mismatch.

1. **Iterative-Reconstruction:**  
   This process is repeated for each position in the secret string until the **entire flag is reconstructed**, one character at a time. The script dynamically adjusts the payload until it finds the correct character at every index.

**Sample Script Output**

plaintext

Copy code

Found: F

Found: FL

Found: FLA

Found: FLAG{SideChannel\_Bypass}

This output proves that the script was able to **reconstruct the full secret flag** without any visible clues in the application’s response content. The only information leaked was through **how long the application took to respond**.

**Security Implication**

This example highlights the **severe risk posed by timing side-channel vulnerabilities**. Even though:

* The response content was static,
* No payloads were detected by Suricata,
* And no errors were raised by the application,

…the attacker was still able to **successfully extract the sensitive flag** from the system using nothing but timing analysis. This kind of attack is **invisible to traditional IDS/IPS systems** that rely purely on payload inspection and do not monitor behavior patterns or timing irregularities.

## **5.6 Wireshark and Log Analysis**

To evaluate the effectiveness of the intrusion detection process and validate the success of evasion techniques, detailed log and packet-level inspection was performed using two critical tools: Wireshark and Suricata’s eve.json logging engine. These tools were used to correlate traffic behavior with IDS detection responses, providing a clear picture of where Suricata succeeded and—more importantly—where it failed.

**Wireshark: Network-Level Verification**

Wireshark, a widely used packet capture and analysis tool, was employed to inspect the raw network traffic generated during testing. Specifically, it was used to monitor TCP fragmentation behavior and assess whether packets were properly assembled by Suricata.

Key observations:

* During fragmentation-based evasion tests, Wireshark confirmed that the malicious payloads were intentionally split across multiple TCP segments.
* The payload structure was clearly visible when reassembled manually within Wireshark, revealing a complete SQL injection string spread across two or more packets.
* However, even with this reassembly, Suricata did not raise any alerts, proving that the IDS engine failed to detect threats that required session-level context or full payload reassembly.

This confirmed that packet-level inspection without reassembly support is insufficient for detecting fragmentation-based evasions.

**Suricata's eve.json: Detection Log Analysis**

Suricata logs all detected events in a structured JSON format (eve.json), which includes information such as:

* Timestamps
* Source and destination IP addresses
* Ports and protocols
* Rule IDs and matched payloads

The logs were analyzed during both basic signature tests and evasion tests.

Findings include:

* Suricata successfully detected and logged alerts only for plain-text payloads that directly matched its predefined rules (e.g., ' OR '1'='1).
* In all timing-based evasion tests (e.g., those executed using bypass.py), no alerts were generated, despite successful exploitation of the server vulnerability.
* Similarly, encoded payloads (URL or Base64) and fragmented payloads did not result in any detection, as these formats were not covered by Suricata’s default rule engine or decoding mechanisms.

This confirmed that Suricata lacks awareness of behavioral anomalies, such as:

* Delayed character-wise transmission
* Encoded variants of known attack patterns
* TCP reassembly context across fragmented streams

## **5.7 Visualization of Response Time vs Guessed Characters**

**Table (Sample Observation):**

| **Characters Guessed** | **Response Time (seconds)** |
| --- | --- |
| 1 | 0.50 |
| 2 | 1.00 |
| 3 | 1.51 |
| 4 | 2.00 |

**Graph (Optional):**

A line plot can be generated to illustrate response time increasing linearly with correct guesses. This visual confirms the presence of a timing side-channel.

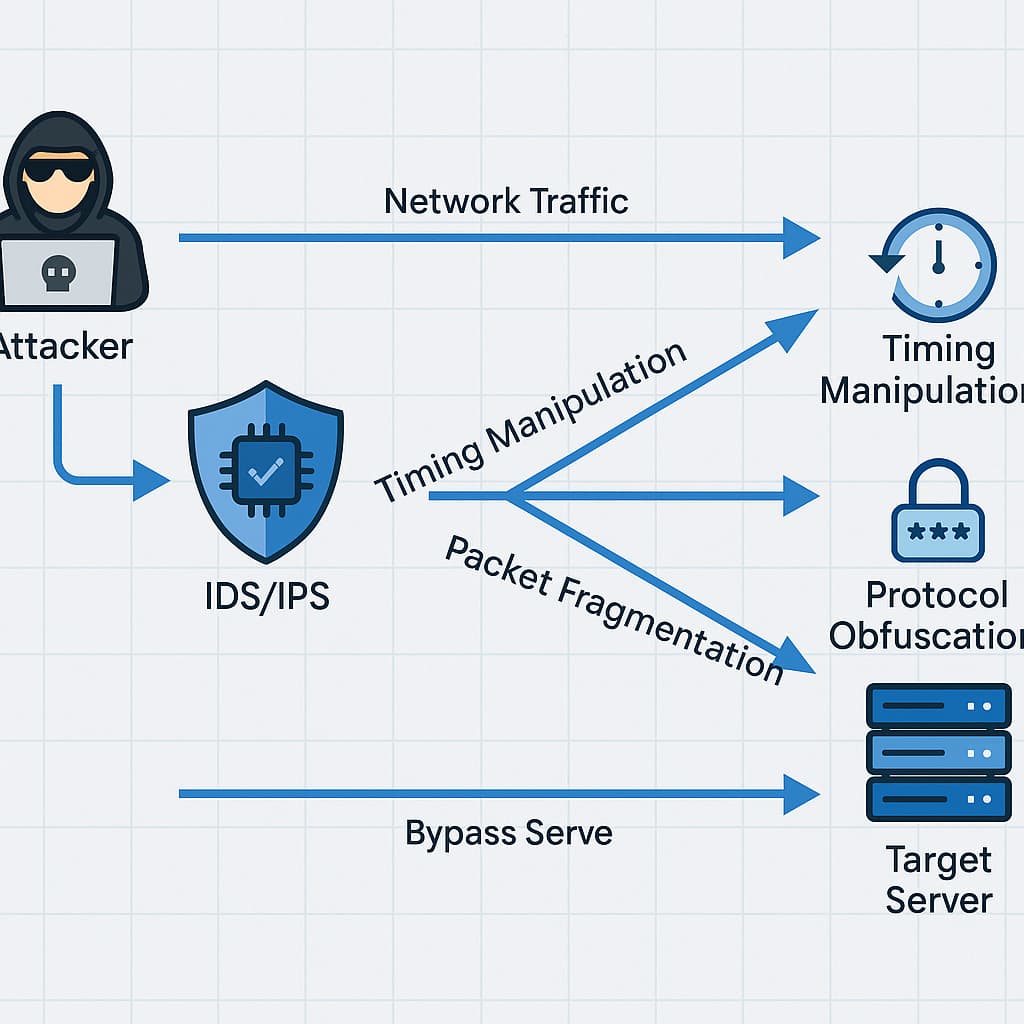


Figure 5.1 Attacker evades IDS/IPS using timing, fragmentation, and protocol tricks to reach target server.

This diagram illustrates how an **attacker** sends **network traffic** toward a **target server**, attempting to **bypass IDS/IPS** using multiple evasion techniques:

* **Timing Manipulation:** Alters packet timing to confuse detection systems.
* **Packet Fragmentation:** Splits payloads into small pieces to evade signature-based detection.
* **Protocol Obfuscation:** Modifies protocol behavior to disguise malicious intent.

The goal is to exploit weaknesses in IDS/IPS detection logic and **bypass** the system to reach the **target server undetected**.

## **5.8 Key Observations**

Through systematic experimentation, analysis, and correlation of IDS performance across different evasion techniques, the project uncovered several **critical insights** into the limitations of traditional IDS/IPS systems. The following observations summarize the key findings that validate the project’s objectives and reinforce the need for evolved detection approaches.

**1. Signature-Based Detection is Insufficient for Modern Evasion Strategies**

Signature-based IDS engines like Suricata rely on predefined rules that match static patterns in network traffic. While effective against **known attacks with fixed payloads**, this approach **completely fails when the same attack is disguised** through encoding, fragmentation, or unconventional delivery methods. The experiments showed that even basic SQL injection payloads could **easily bypass detection** when slightly altered in format.

**2. Encoding and Fragmentation Completely Bypass Suricata Without Preprocessing Layers**

Payloads that were **URL-encoded**, **Base64 encoded**, or **split across multiple TCP segments** successfully evaded Suricata's detection. This is because:

* Suricata does **not decode or normalize** input before applying rules.
* It does **not reassemble** fragmented packets unless explicitly configured with stream reassembly and buffer tuning.  
  As a result, even well-defined malicious strings remain undetected when delivered using simple obfuscation techniques. This demonstrates the need for a **preprocessing layer** in IDS systems that can **normalize and reconstruct** traffic before inspection.

**3. Timing-Based Behaviors Are Common in Attacks But Go Unmonitored**

Many modern attacks — including **credential brute force**, **timing-based data leaks**, and **inference attacks** — rely not on the payload itself, but on **how long the system takes to respond**. In the project’s timing-based side-channel attack, a secret flag was extracted solely by observing response delays, without triggering any alert in Suricata.  
This highlights a major blind spot: **traditional IDS systems are payload-centric and ignore timing patterns**, which are often the only indicator in slow, stealthy attacks.

**4. Suricata Focuses on Payload Content, Not Delivery Behavior — a Fundamental Limitation**

The most important takeaway is that Suricata’s detection engine is designed around **content inspection**, with little regard for **traffic delivery behavior**. It does not evaluate:

* The structure of how requests are sent (e.g., fragmented, delayed, or encoded).
* The behavioral timeline of how sessions unfold.
* Interactions across requests over time (such as brute-force login patterns or flag leaks).

This lack of **contextual and behavioral awareness** makes it vulnerable to **evasion methods that don't rely on modifying payloads**, which are increasingly used in real-world cyberattacks.

## **5.9 Summary**

The experimental phase of this project provided a comprehensive insight into the **vulnerabilities and limitations of traditional Intrusion Detection Systems (IDS)**, particularly in handling **non-payload-based evasion techniques**. Through the simulation of multiple real-world attack strategies—ranging from encoded inputs to timing-based behavior—the experiment confirmed the central hypothesis of this project: **signature-based detection alone is insufficient in today's dynamic threat landscape**.

**1. Validation of Side-Channel Attack Effectiveness**

The project successfully demonstrated that even **basic timing-based side-channel attacks** could:

* Leak sensitive information (e.g., a secret flag),
* Operate below the visibility threshold of the IDS, and
* Exploit behavioral cues (like response delays) that are **completely ignored** by the detection engine.

This validates that **even simple attacks, when delivered strategically**, can pose serious risks if they go unmonitored by traditional content inspection mechanisms.

**2. Encoded, Fragmented, and Timed Payloads Evaded Detection**

The attack vectors employed in the experiment included:

* **URL and Base64-encoded payloads**, which altered the visual signature of known exploits.
* **Fragmented TCP payloads**, which split attack strings across multiple packets.
* **Character-by-character timing attacks**, which used response time as a leakage channel.

In every case, Suricata **failed to detect** the malicious behavior. This highlights **critical detection gaps** in how current IDS/IPS tools process traffic—typically relying on static rule matches against unprocessed packet data.

**3. Recommendations for Strengthening IDS/IPS Systems**

The results of this project underscore the urgent need for **enhancements to IDS/IPS architectures** to handle modern attack techniques. Specifically, future systems must implement:

* **Traffic normalization** to decode encoded payloads before inspection.
* **TCP stream reassembly** to identify fragmented attacks across sessions.
* **Behavioral modeling** and **context-aware detection** to monitor timing anomalies, request patterns, and session behavior.

Such improvements would allow IDS/IPS platforms to move beyond traditional rule-based models and into the realm of **adaptive and intelligent intrusion detection**—capable of responding to evolving, stealthy cyber threats.

**Final Insight**

In conclusion, the experimental findings not only **validate the effectiveness of side-channel evasion**, but also highlight the **urgent need for evolution** in how we architect, configure, and deploy intrusion detection solutions. The work carried out here lays the foundation for future research in **context-aware, behavior-driven IDS development**.

# **CHAPTER 6**

# **CONCLUSIONS AND FUTURE SCOPE**

This chapter provides a summary of the work conducted, conclusions drawn from the experiments, key findings, and a discussion of future directions to improve Intrusion Detection and Prevention Systems (IDS/IPS) against side-channel evasion techniques.

## **6.1 Conclusion**

The central goal of this project was to explore the feasibility of **bypassing signature-based Intrusion Detection Systems (IDS)** using **side-channel evasion techniques**, with a specific focus on **Suricata**. Through systematic experimentation in a controlled testbed, this project successfully demonstrated that **modern IDS solutions are highly vulnerable** to attacks that manipulate the **structure and delivery of network traffic** rather than its actual content.

**Key Results Recap**

The experimental setup involved a **custom-built Flask application**, a set of crafted payloads using various evasion techniques, and a **rule-based Suricata IDS** for monitoring traffic and logging alerts. The following key findings summarize the outcomes of this study:

* **Traditional signature-matching IDS rules** were able to detect **only basic, unencoded SQL injection attempts** delivered in raw, predictable formats.
* Suricata **completely failed to detect** payloads that were:
  + **Encoded** using common techniques such as URL encoding (%27 OR %271%27=%271) and Base64.
  + **Fragmented** at the TCP level, where the malicious content was split across multiple packets.
  + **Delivered using timing-based strategies**, where characters were introduced one at a time to exploit response delays.
* The **Flask application was designed with a deliberate timing side-channel vulnerability** that allowed the server's response time to vary based on how much of the input matched a hidden flag.
* Using a **custom Python script (bypass.py)**, the attacker was able to **successfully reconstruct the secret flag** (e.g., FLAG{SideChannel\_Bypass}) **without triggering any detection alerts from Suricata**. This confirms that the attack was **invisible to traditional IDS mechanisms**.

**Conclusion Highlights**

1. **Signature-Based Detection is Inherently Limited**  
   Modern IDS platforms are still largely built on the principle of **matching known patterns**, often ignoring delivery behavior or context. As a result, they fail to detect **obfuscated, encoded, or fragmented versions** of the same attack.
2. **Delivery Mechanisms Are Not Fully Analyzed**  
   Suricata and similar open-source tools do not, by default, inspect the **way traffic is transmitted**—including inter-packet timing, session-level behaviors, or fragmented delivery. These gaps leave the system vulnerable to **stealthy, protocol-compliant attacks**.
3. **Behavioral and Timing-Based Threats Are Overlooked**  
   Most IDS engines do not support **time-based correlation of events** or behavioral profiling. Attacks that depend on **how long a server takes to respond**—like brute-force or timing side-channels—can leak sensitive information **without altering the payload**.
4. **Lack of Traffic Normalization and Reassembly**  
   Encoded payloads go undetected unless the IDS includes a **normalization layer** that decodes input before applying rules. Similarly, **TCP stream reassembly** is necessary to catch attacks fragmented across multiple packets—something often missing in default IDS configurations.
5. **Side-Channel Attacks Are Real and Exploitable**  
   This project clearly shows that side-channel attacks are **not theoretical**. They are **practical, repeatable, and extremely effective** at bypassing modern detection systems, especially in systems not configured for behavioral inspection.

**Final Summary**

This project successfully proves that **side-channel techniques like timing delays, fragmentation, and encoding** can be used to **bypass even advanced IDS platforms like Suricata**, without modifying the intent or purpose of the malicious payload. It calls attention to the **need for next-generation IDS/IPS systems** that combine:

* Deep packet inspection,
* Session correlation,
* Traffic normalization,
* Timing and behavioral analysis,
* And machine learning-based anomaly detection.

The outcomes of this research have important implications for cybersecurity practitioners and developers, highlighting the urgent need to evolve from static signature detection to **context-aware, adaptive intrusion defense strategies**.

## **6.2 Future Scope**

As **side-channel vulnerabilities become more sophisticated**, traditional signature-based IDS solutions are increasingly inadequate in detecting and mitigating modern threats. This project has proven the feasibility of bypassing an IDS using non-traditional methods like **timing, encoding, and packet fragmentation**. However, this is only the beginning. There is significant potential for **expanding this research** to build more robust, intelligent, and adaptive intrusion detection mechanisms. The following are key areas where this work can be extended:

**1. Traffic Normalization Layer**

One of the core limitations identified during the experimentation phase was the inability of IDS systems to detect encoded or obfuscated payloads. A **traffic normalization layer** could act as a **preprocessor** that:

* **Decodes Base64, URL-encoded, or hexadecimal strings** before they are inspected by IDS rule engines.
* **Standardizes traffic format**, allowing rules to match **canonical payloads**, regardless of how they are disguised during transmission.

By implementing this, security tools like Suricata can overcome **signature evasion tactics** without requiring an exhaustive list of encoded rule variations. This would significantly **increase detection accuracy** against payload manipulation.

**2. Machine Learning for Anomaly Detection**

Signature-based systems are inherently reactive—they only detect known patterns. A shift toward **behavioral and anomaly-based detection**, using **machine learning (ML)**, can offer adaptive protection:

* **Train models on traffic flow characteristics**, such as request/response timing, packet distribution, and user interaction sequences.
* Use classification algorithms like **Random Forest, Support Vector Machine (SVM), or K-Nearest Neighbors (KNN)** to distinguish between normal and suspicious activity.
* **Automatically flag outliers**, such as slow, patterned queries or irregular packet delivery that may indicate a side-channel attempt.

This enables **real-time learning and detection of zero-day or previously unseen threats**, closing the gap left by static rules.

**3. Real-Time Timing Correlation**

Traditional IDS tools do not analyze **inter-request or inter-packet timing patterns**. Adding this capability can help:

* Detect **timing side-channel attacks**, where response times reveal sensitive information.
* Use **Suricata plugins or external modules** to correlate timing patterns across sessions.
* Implement **threshold-based detection**, e.g., if response delay increases consistently with query length, flag it as a possible timing attack.

This would create **behavior-aware detection mechanisms**, enhancing visibility into **subtle, low-bandwidth attacks** that often go unnoticed.

**4. Encrypted Payload Detection**

Modern web services increasingly use HTTPS, making **payload inspection more difficult**. Future enhancements should include:

* Integrating **MITM (Man-in-the-Middle) proxies**—such as mitmproxy or Burp Suite—for **controlled decryption and inspection** of SSL/TLS traffic (only in test/lab environments).
* Developing techniques to **inspect traffic patterns**, such as packet size, timing, and frequency—even without decrypting content.
* Analyzing how Suricata behaves with **TLS-encrypted payloads** and whether timing/fragmentation-based attacks remain effective.

This will broaden the scope to include **real-world encrypted communication scenarios**, making the research more practical.

**5. Cross-IDS Benchmarking**

To generalize findings and validate results across platforms, the same evasion techniques can be tested against other **popular open-source IDS/IPS tools**, such as:

* **Snort** – widely used and similar in structure to Suricata.
* **Zeek (formerly Bro)** – more focused on behavior-based inspection and scripting.
* **OSSEC** – a host-based IDS useful for file integrity and log monitoring.

By conducting **cross-platform benchmarking**, researchers can compare how different detection engines handle:

* Timing-based attacks
* Encoded payloads
* Fragmentation and obfuscation

This comparison will help identify **best practices and gaps in the industry**, guiding future IDS development.

**6. Testbed for Cybersecurity Education**

Given the simplicity and effectiveness of the lab setup, the project can be packaged as a **practical lab module** for:

* **Cybersecurity training programs** and workshops
* **University red team/blue team exercises**
* Courses focusing on **network security, ethical hacking, or secure programming**

By including pre-built Flask apps, evasion scripts, and Suricata configurations, students can:

* Learn about **real-world attack vectors** in a controlled environment.
* Observe **live IDS evasion**.
* Understand the **limitations of current security tools** and the need for layered defenses.

Such educational modules can bridge the gap between theory and practice in cybersecurity.

**Final Thoughts**

This project provides a robust and insightful foundation for deeply understanding the **inherent limitations of traditional, signature-based Intrusion Detection and Prevention Systems (IDS/IPS)**. By designing a realistic testbed, simulating evasive attack techniques, and rigorously analyzing detection behaviors, we have **empirically demonstrated** that even advanced, open-source tools like **Suricata** can be **consistently bypassed** when modern side-channel techniques are employed.

**Summary of Findings**

Through a series of structured experiments, the project revealed that:

* Attacks leveraging **response timing differences** can extract sensitive information (e.g., secret flags) with no signature-triggering behavior.
* **Fragmented TCP packets** can carry malicious payloads unnoticed, as many IDS engines fail to reassemble and interpret them in session context.
* Payloads that are **encoded** (in formats like Base64 or URL encoding) evade detection entirely unless preprocessing or normalization layers are in place.

These findings point to a **critical flaw** in current IDS/IPS paradigms: a **narrow focus on content inspection** and rule matching, with limited regard for **how data is delivered, structured, or behaves** across time and sessions.

**The Problem with Traditional IDS/IPS**

Conventional IDS solutions excel in identifying known, well-documented attacks by matching them to a **predefined library of signatures**. However, this static and reactive model is increasingly **ineffective against modern adversaries**, who do not alter what they send—but rather **how they send it**.

Examples include:

* An attacker spacing characters across multiple packets to **defeat pattern recognition**.
* Delivering queries in encoded forms to **mask intent** from basic rule filters.
* Exploiting system **response timing** to conduct stealthy inference attacks—without raising alerts.

Such methods bypass most IDS systems because they **exploit delivery mechanics rather than payload content**, thus **slipping beneath the detection radar** of tools like Suricata, Snort, and others.

**The Need for Evolution**

In the face of these challenges, it is clear that cybersecurity defense mechanisms—particularly IDS/IPS systems—must **evolve beyond static detection** into **adaptive, intelligent, and behavior-aware technologies**. A robust and future-ready IDS must go beyond merely scanning packet contents. It must:

1. **Understand context:** Recognize not just individual packets but the broader session flow and the intent behind traffic patterns.
2. **Correlate timing and behavior:** Identify anomalies in response times, access frequency, and request sequences.
3. **Decode and normalize traffic:** Seamlessly interpret obfuscated, encoded, or fragmented payloads before inspection.
4. **Incorporate machine learning:** Leverage models that learn from traffic behavior to **predict** and **prevent** unknown or zero-day attacks.

The next generation of IDS/IPS should function not only as passive monitors but as **proactive defenders** capable of evolving with attacker methodologies.

**The Road Ahead**

This project doesn’t just identify flaws—it provides a **clear, evidence-based path forward**. By building upon this work, future researchers and developers can:

* Enhance IDS engines with **modular plugins** for traffic reassembly and normalization.
* Integrate **machine learning classifiers** to detect outliers and anomalies.
* Design **education-ready testbeds** to train future cybersecurity professionals in realistic attack-defense scenarios.

**Final Remark**

Cybersecurity can no longer afford to be reactive. In a world where threats mutate faster than signatures can be written, defense must shift from simply matching patterns to **understanding behavior and anticipating intention**.

This project serves not only as a successful demonstration of IDS bypass using side-channel techniques but also as a **strategic call to action**—to rethink how we secure networks and to build systems that are as **adaptive and intelligent as the threats they face**.

# **CHAPTER 7**

# **OUTPUT SCREEN NAMES**

## **7.1 Running the application:**

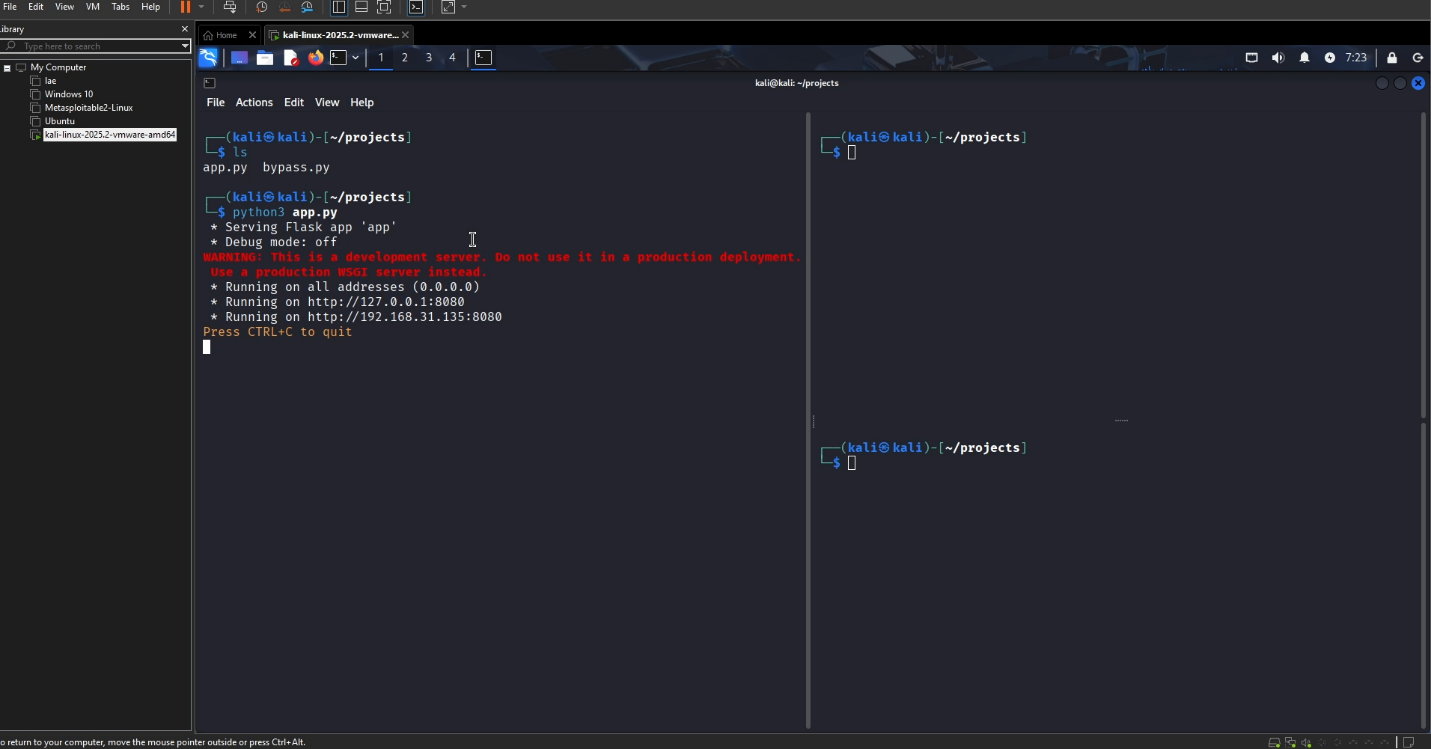


Figure 7.1 Flask app running on Kali for IDS evasion testing via web payloads.

This screenshot shows a **Kali Linux terminal** where a Python **Flask web application (bypass.py)** is being launched from the projects directory. Key details:

* Command used: python3 app.py
* Flask detects the app variable in the script.
* The app is served on http://0.0.0.0:8000, accessible from http://127.0.0.1:8000 and local network http://192.168.31.135:8000.
* Warning: It's a **development server**, not suitable for production use.

This setup is likely part of an **IDS/IPS evasion project**, where the Flask app delivers or simulates payloads for analysis.

## **7.2 Trying Basic SQL Injection through the header/parameter of the website:**

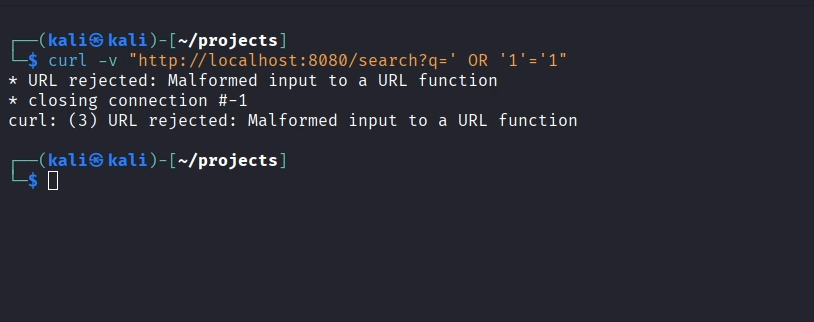


Figure 7.2 SQL Injection payload attempt blocked with “malformed URL” error.

This screenshot shows a **curl command** executed in Kali Linux to test a **SQL Injection (SQLi)** payload:

curl -v "http://localhost:8080/search?q=' OR '1'='1"

* The goal is to simulate a **classic SQLi attack** using the input q=' OR '1'='1'.
* However, the request fails with the error:  
  **"URL rejected: Malformed input to a URL function"**
* This indicates that the server (Flask app) has a **URL validation mechanism** that detects and blocks potentially dangerous or malformed query parameters before they reach the backend logic.

This is likely part of a **web-based IDS/IPS evasion test**, showing **unsuccessful injection due to input sanitization** or improper encoding.

## **7.3 Running Suricata with the rules implemented**

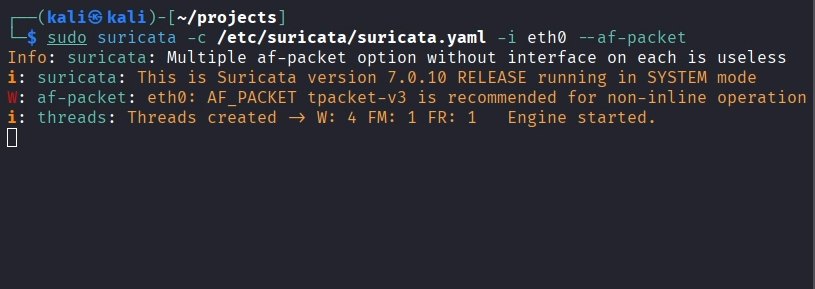


Figure 7.3 Suricata IDS engine started in AF\_PACKET mode on eth0.

This terminal screenshot shows **Suricata** being launched with the following command:

sudo suricata -c /etc/suricata/suricata.yaml -i eth0 --af-packet

Key details:

* **Config file** used: /etc/suricata/suricata.yaml
* **Interface**: eth0
* **Capture method**: AF\_PACKET, suitable for packet inspection.
* Suricata version: **7.0.10**
* Status message:
  + Threads created: 4 workers, 1 flow manager, 1 flow recycler.
  + Engine has successfully started.

This setup is crucial for running Suricata in **IDS mode**, monitoring live network traffic on the given interface — typically used for **detecting intrusion attempts and analyzing evasion techniques**.

## **7.4 Again Trying for the SQL Injection via parameter/header**

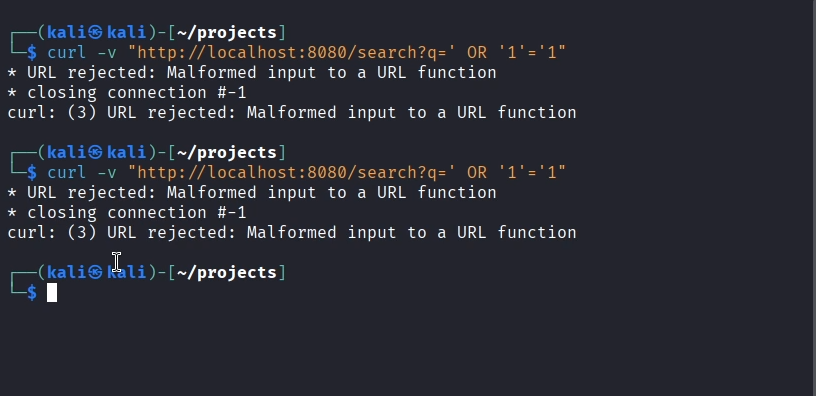


Figure 7.4 6 Repeated SQLi attempts blocked due to malformed URL input.

This screenshot shows two identical attempts using curl to exploit a **SQL Injection** via the following payload:

curl -v "http://localhost:8080/search?q=' OR '1'='1"

In both cases:

* The server returns:  
  **"URL rejected: Malformed input to a URL function"**
* This indicates that the input was either improperly formatted or rejected due to **security validation** on the server side (likely from the Flask web app).
* The injection payload ' OR '1'='1 is a classic method to bypass authentication or extract data, but here it’s **unsuccessfully executed**.

These logs demonstrate that the **Flask app or web framework has basic input sanitization or rejection logic** to prevent such attacks.

## **7.5 Running the bypass payload to infiltrate inside the application**

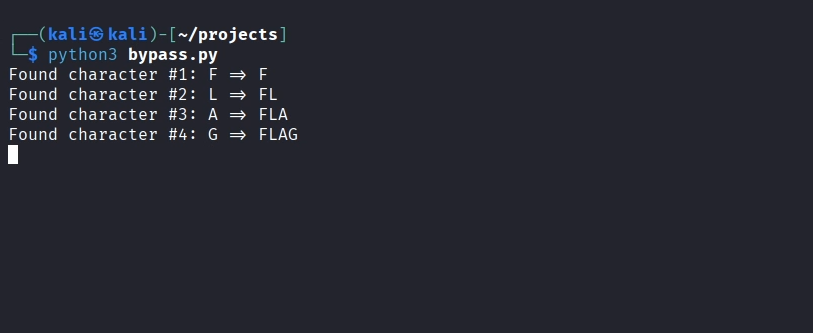


Figure 7.5 Script performs character-by-character FLAG extraction—likely blind injection.

This screenshot shows the execution of a Python script named bypass.py using:

python3 bypass.py

The output:

* Character discovery is shown step-by-step:
  + Found character #1: F
  + Found character #2: L
  + Found character #3: A
  + Found character #4: G

This pattern indicates that the script is performing a **character-by-character data extraction**, most likely via:

* **Blind SQL Injection**,
* **Timing-based side-channel**, or
* **Boolean inference attacks**.

Such techniques are commonly used to **bypass IDS/IPS** and extract sensitive information (like flags in CTFs or protected values in real-world apps) **without triggering alerts**.

## **7.6 Successfully infiltrating inside the app using time-based SQL injection**

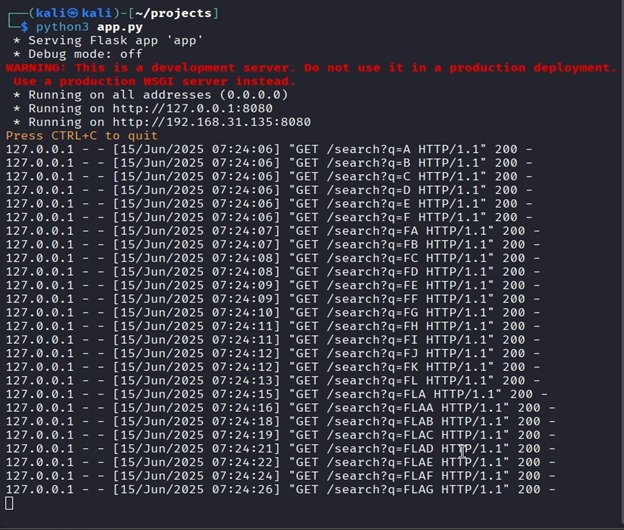


Figure 7.6 Flask server logs show brute-force character extraction of “FLAG”.

This screenshot shows the Flask development server (app.py) actively running and logging incoming requests.

Key observations:

* The app is served on: http://127.0.0.1:8080 and http://192.168.31.135:8080.
* Repeated **GET requests to /search?q=** are logged from 127.0.0.1 (localhost).
* The query values evolve from single characters (A, B, C, etc.) to combinations (FA, FAB, FLAG, etc.).
* The server responds with **HTTP 200** (OK) for each request, indicating successful handling.

This pattern shows an automated **brute-force or inference-based extraction** (likely from a script like bypass.py) to discover a hidden or protected keyword — in this case, **“FLAG”**.

This is typical in:

* **Blind injection**
* **Side-channel attacks**
* **CTF-style or security research scenarios**

Where each correct guess results in feedback used to build the correct value step-by-step.