## 📌 What Is the Project?

This is a Python-based simulation of a **lightweight intrusion detection system (IDS)** that uses **entropy** to detect potential **Distributed Denial of Service (DDoS) attacks** based on traffic patterns—specifically, the diversity of IP addresses sending traffic.

It simulates both **normal traffic** (healthy, diverse network activity) and **attack traffic** (overloaded, malicious traffic from a few sources), and applies information theory (entropy calculation) to distinguish between them.

### 1. ✅ First: What is a DDoS Attack?

A **Distributed Denial of Service (DDoS)** attack is when **multiple systems flood a target server or service** with excessive traffic, aiming to overwhelm it and render it unusable.

Typical characteristics:

- Huge volume of requests
- Requests often come from a **small set of IP addresses** (botnets or spoofed)
- Traffic pattern is repetitive and lacks diversity

### 2. 📊 Why Entropy? What Is It?

**Entropy**, in this context, measures the **randomness** or **unpredictability** in a set of values.

From Shannon's Information Theory:

$$H = -\sum p(i) \cdot \log_2 p(i)$$

Where:

- $p(i)$ is the **probability** of each unique IP address in the traffic
- $H$ is the total entropy

More IP diversity = **higher entropy**
Fewer repeated IPs = **lower entropy**

🧠 **Intuition:**

- A crowd of people from all countries = high entropy (normal traffic)
- 1,000 clones of the same person = low entropy (attack traffic)

### 3. 🧪 Simulation Logic

This project **simulates traffic** (not actual packet sniffing):

- **Normal traffic**: randomly selected from a wide IP pool → higher entropy
- **Attack traffic**: mostly from a few repeated IPs → lower entropy

You simulate this using Python's `random.choices()` function to generate packets coming from a list of IPs.

### 4. 🔍 The Detection Pipeline

Here's the actual logic pipeline of the code:

| Step | Description |
|---|---|
| `send_packets()` | Simulates traffic with a uniform (normal) or concentrated (attack) IP distribution |
| `compute_entropy()` | Calculates entropy of the traffic based on frequency of each IP |
| `detect_ddos()` | Compares entropy to a **threshold** value to classify the traffic |
| `find_accuracy()` | Repeats this process for multiple test cases to measure how accurate the classifier is |

### 5. 🎯 What's the Threshold?

You need a **cutoff point** to decide when traffic is suspicious.

The project sets the **threshold entropy** as the average of:

- Normal traffic entropy
- Attack traffic entropy

Then:

- If current traffic entropy < threshold → classify as DDoS
- Otherwise → classify as normal

This approach is **unsupervised**, meaning no labeled training data is needed—it learns the threshold dynamically during simulation.

### 6. ⚙️ Key Parameters You Can Tweak

| Parameter | What It Controls |
|---|---|
| `num_packets` | Total number of simulated packets in a session |
| `num_ips` | Total number of unique IPs to simulate normal diversity |
| `attack=True/False` | Whether traffic simulates an attack |
| `num_tests` | How many total test iterations are run |
| `threshold_entropy` | The cutoff value for detection logic |

### 7. 🧪 Accuracy & Evaluation

You run tests with both attack and normal traffic.
Each time:

- It checks whether the classifier labels the traffic correctly.
- Over multiple test cycles, it computes a **detection accuracy score**.

You can also turn on `verbose=True` to **see the classification output** for every test.

## 💥 Limitations of This Approach

- It's a **simulation** — no real-time packet sniffing
- **Spoofed or distributed attacks** with many IPs could bypass this (entropy evasion)
- The threshold is **hardcoded** from an average—no learning or adaptability
- Doesn't consider **packet rate**, **payloads**, or **timestamps**

# 🧰 Phase 1: From Simulation to Real-Time Detection

**Option B: Use Packet Sniffers (If You Want Deep Packet Inspection)**

- Tools like `scapy`, `tcpdump`, or `pyshark` can capture traffic directly.
- Risk: requires root access, slows down your server, not ideal for production apps unless absolutely needed.

**3. Why use entropy instead of rate limiting or signature-based detection?**

- **Signature-based**: Fails for novel attacks.
- **Rate limiting**: Works per IP, doesn't catch distributed attacks.
- **Entropy**: Doesn't rely on known patterns, adapts to new threats, works at the aggregate level across IPs.

**9. Could an attacker bypass this system by spoofing IPs?**

Yes. If the attacker uses many spoofed IPs to artificially inflate entropy, the system might classify it as normal. That's why real-world systems combine entropy with behavioral analysis or deep packet inspection.

---

## 🔷 EDGE CASES & STABILITY

**11. What if traffic is very low (e.g., <100 packets)?**

Low sample size can skew entropy. To handle this:

- Require minimum packet count before computing entropy
- Use time-window buffering
- Apply entropy smoothing or normalization

**12. What happens if normal traffic has low entropy?**

That can happen, e.g., a few power users or internal traffic. In that case:

- Adjust the threshold dynamically
- Compare current entropy against historical baseline instead of static threshold

**13. How does your model handle false positives/negatives?**

It doesn't—yet. That's one limitation. To handle them:

- Add confidence intervals
- Introduce a feedback loop for misclassifications
- Use multiple features, not just entropy

**14. What is the minimum entropy value possible in your system?**

If all traffic comes from **one IP**, entropy = 0.

This represents the most deterministic distribution—complete certainty about the source.

**15. What kind of real-world data would break this system?**

- Load balancers rotating IPs rapidly
- IoT devices with bursty but uniform traffic
- CDN nodes all funneling traffic from a few IPs
  These could be flagged as DDoS even if legit.