# Attack Fingerprints based on the Activity and EventNetwork(AEN) Model

by

**Chenyang Nie**

A Report Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Engineering

In the department of Electrical and Computer Engineering


University of Victoria

# Attack Fingerprints based on the Activity and EventNetwork(AEN) Model

by

**Chenyang Nie**

**Supervisory Committee**

Dr. Issa Traoré

Department of Electrical and Computer Engineering

Dr. Tao Lu

Department of Electrical and Computer Engineering

# Abstract

The Activity and Event (AEN) graph is a new framework that enables capturing ongoing security-relevant activity and events occurring at a given organization using a large random time varying graph model. The graph is generated by processing various network security logs, such as network packets, system logs, and intrusion detection alerts. In this report, we show how known attack methods can be captured generically using attack fingerprints based on the AEN graph. The fingerprints are constructed by identifying attack idiosyncrasies under the form of subgraphs that represent indicators of compromise (IOCs), and then encoded using PGQL queries. Among the many attack types, three main categories are implemented in our model: Probing, Denial of Service(DoS), and authentication breaches; Each category contains its common variations. The experimental evaluation of the fingerprints was carried using a combination of intrusion detection datasets and yielded very encouraging results.

# Contents

# List of Figures



# List of Tables

# Dedication

Dedicated to Dr. Issa Traoré for all the guidance and encouragement throughout the project; and Paulo Gustavo Quinan, who is the main contributor of Activity and EventNetwork(AEN) Model, has provided me great technical help.

# Chapter 1 Introduction

## 1.1 Background

A network attack is an attempt to gain unauthorized access to an organization's network, intending to steal data or perform other malicious activity, and has been an increasing threat[10] to the Internet.

Network attacks consists of a wide range of types which can happen at any layer of the OSI[9] model, and most attacks follow some generic steps. A typical network attack usually involves two phases: Active Reconnaissance and System Compromise[16]. Active Reconnaissance is the phase where the attacker proactively gathers information, and identifies the vulnerabilities. Some of the most common things that an attacker would do are finding the hosts' accessibility, the geolocation, the operating system information, the open ports, the running applications, and services, etc.

System compromise is the phase where the attacker would exploit the vulnerabilities that have been identified in phase one to the target host.Some major steps of this phase are as follows:

- Gaining access + DoS(Denial of Service)[1]: Gaining access is done by executing some exploits such as operating system attack, application-level attack. DoS is an attack that makes the resources and services which are connected to the internet unavailable to the user of the host. This is one of the most common network attacks and will be discussed further.
- Privilege Escalation: An further attempt to escalate access to the local computer, such as getting root privilege.
- Maintaining Access: Leaving a backdoor application on the host after gaining access.
- Covering tracks: The followup of an attack. It's usually done by deleting the system logs to make the attack untraceable.

## 1.2 Project Objective

The Activity and Event Network (AEN) is a new graph model developed at the Information Security and Object Technology (ISOT) to capture various network security events occurring in the network perimeter. The purpose of the AEN model is to provide a basis to detect both novel and known attack patterns, as well long-term and stealth attack methods, which have been on the rise but have proven difficult to detect.

The AEN graph is developed on a graph database: Oracle's PGX[14] engine. graph is constructed by processing various network security data sources such as network traffic files: packets, syslogs, alerts, etc. Property Graph Query Language (PQGL)[15] is a query language that is designed to query results based on a graph pattern, with similar syntax to SQL. The PGQL gives us access to construct attack fingerprints by analyzing their graph patterns.

In this project, we illustrate how known attack signatures can be expressed and implemented using the AEN graph framework. It is expected that the patterns will be stored as generic queries or fingerprints that would be matched against subgraphs to identify attack occurrences.

Obviously, there is a large number of known attack signatures. Attempting to express all these signatures will be overwhelming. Because our primary goal is to illustrate the process of defining attack fingerprints using the AEN model, in the current report we focus only on a small subset of attack fingerprints covering scanning[7], denial of service (DoS)[1] and authentication breaches[4]. The proposed fingerprint database can be extended easily to incorporate other known attack fingerprints.

## 1.3 Report Outline

The remaining chapters in this report are structured as follows. In Chapter 2, we present the structure of the fingerprint and discuss the implementation challenges and solutions. In Chapters 3 - 5, we present the fingerprint models for scanning, DoS, and password attacks, respectively. In Chapter 6, we conduct the experimental evaluation of the proposed attack fingerprinting scheme on two datasets: ISOT-CID Phase I and CICIDS 2017[17]. Finally, in Chapter 7, we make concluding remarks.

# Chapter 2 Implementation of attack fingerprints in AEN graph model

## 2.1 Program structure

The sample fingerprints presented in this report target packets/sessions in OSI[9] layer 3 (network layer) and layer 4 (session layer).

The structure of the fingerprint package consists of two components: specific attack pattern component and fingerprint execution component. The attack pattern is knowledge-based and falls under a particular category. As proof of concept, we study different attack methods in each family; these can easily be extended by adding more attack methods. We consider the following three families: Scanning, Denial of Service, and Password guessing. For each family, the most common attack types are studied:

- Scanning: different forms of scanning attacks, e.g. vertical, horizontal, etc.;
- Denial of Service: SYN flood[5], UDP flood[19], Fragmented Packet attack[2];
- Password Guessing: different forms of guessing attacks, e.g. basic, distributed, and stuffing[8];

The fingerprint execution component consists of a set of generic classes (i.e.*FingerprintExecutor*, *FingerprintFactory*, and *FingerprintMatchResult*) that call each attack fingerprint and run that fingerprint over the AEN graph. Specifically, *FingerprintExecutor* runs a fingerprint from *FingerprintFactory*, and returns a set of *FingerprintMatchResult* containing the matching result information, which is sent to the server.

We use the PGQL query language to implement the fingerprints. However, PGQL has certain limitations which hamper its usage in our case. In the next subsections, we discuss these limitations and discuss our solutions to address them.

## 2.2 Limitations of PQGL in implementing Fingerprints

PGQL provides a mechanism to retrieve results from the graph with queries. The syntax of PQGL is mostly similar to SQL, although we need to specify a pattern (e.g. host-edge-host) to match. A simple example of PGQL query is as follows:

Listing 1: PGQL query example

```
SELECT s,d
MATCH (s)-[e]->(d)
```

The SELECT clause specifies the return values of the query and MATCH clause specifies the pattern to match. The enclosed brackets represent a vertex, while the enclosed square brackets represent an edge. The arrow specifies the direction. The above example matches any pattern in the graph that involves two vertices connected by a directed edge denoted by e, with a vertex denoted by s pointed to a vertex denoted by d. By expressing the patterns using PQGL, we can retrieve the host vertex from the graph.

As mentioned above, the attack fingerprints will be implemented mainly through PGQL. However, PGQL has the following key limitations:

1. Subquery is not supported in the FROM clause;
2. Only compare operators are supported for Datetime variable;
3. Array aggregation only aggregates paths.

As a result, solely relying on PGQL makes it very difficult to fully express some attack patterns as explained in the following:

- Datetime issue: In volumetric DoS attack scenarios, a typical characteristic is that a massive number of sessions may occur in a short time frame. However, PGQL does not allow getting the time difference between two consecutive sessions. If we can only use comparator on Datetime variables, a better option might be to select a set of pivot time windows and aggregate the sessions into different time windows.
- Array aggregation issue: When we are investigating distributed DoS, it is desirable to group by destination, and get an array of sources. However, this is not currently supported by PGQL. If we want to get the result in one pass, we can only group by destination, thus the attackers information will not be covered in the result.
- Inability to query from subquery:A possible solution for the above two problems could be by querying in two-pass. However, not being able to query from subqueries makes the two-pass method almost impossible.

## 2.3 Addressing PGQL Limitations

The PGQL limitations discussed previously can only be solved outside the query language. In this section, we present our approach to address the aforementioned limitations.

### 2.3.1 Getting distributed attackers list

With the *ARRAY_AGG()* function unable to return a list of attackers, we have to first get a set of attacker-victim results, and aggregate attackers by processing the pre-result. The pre-result has three columns: victim, attacker, and the number of sessions. In the case where the attack is a distributed attack, there is likely a large number of rows which have the same victim and different attackers. In our implementation, two HashMaps[3] are used to process the pre-result: *hostmap* keeps the list of attackers for each victim, and *countmap* keeps the total number of sessions for each victim. If the number is beyond the threshold, that victim-attackers item will be passed to the final result.

### 2.3.2 Grouping by time window

DoS attack usually happens in a very short time window. The program provides a *TimeService* class which keeps track of the earliest and latest timestamp of the current graph. By dividing the whole timeline to consecutive time windows, a fingerprint gets executed in each time window separately.

## 2.4 Graph Example

The AEN model can process network traffic files and generate a graph. Figure 1 shows a part of a visualized graph generated from an example dataset which contains different attacks that are carried out between different hosts.

The traffic modeled in the figure involves both benign and malicious activities. For instance, it can easily be observed that the host 172.16.174.93 establised a lot of sessions to different IPs, e.g. 224.0.0.252 and 172.16.174.1. This graph will serve as an example in the following sections where the fingerprint module will be run against it.

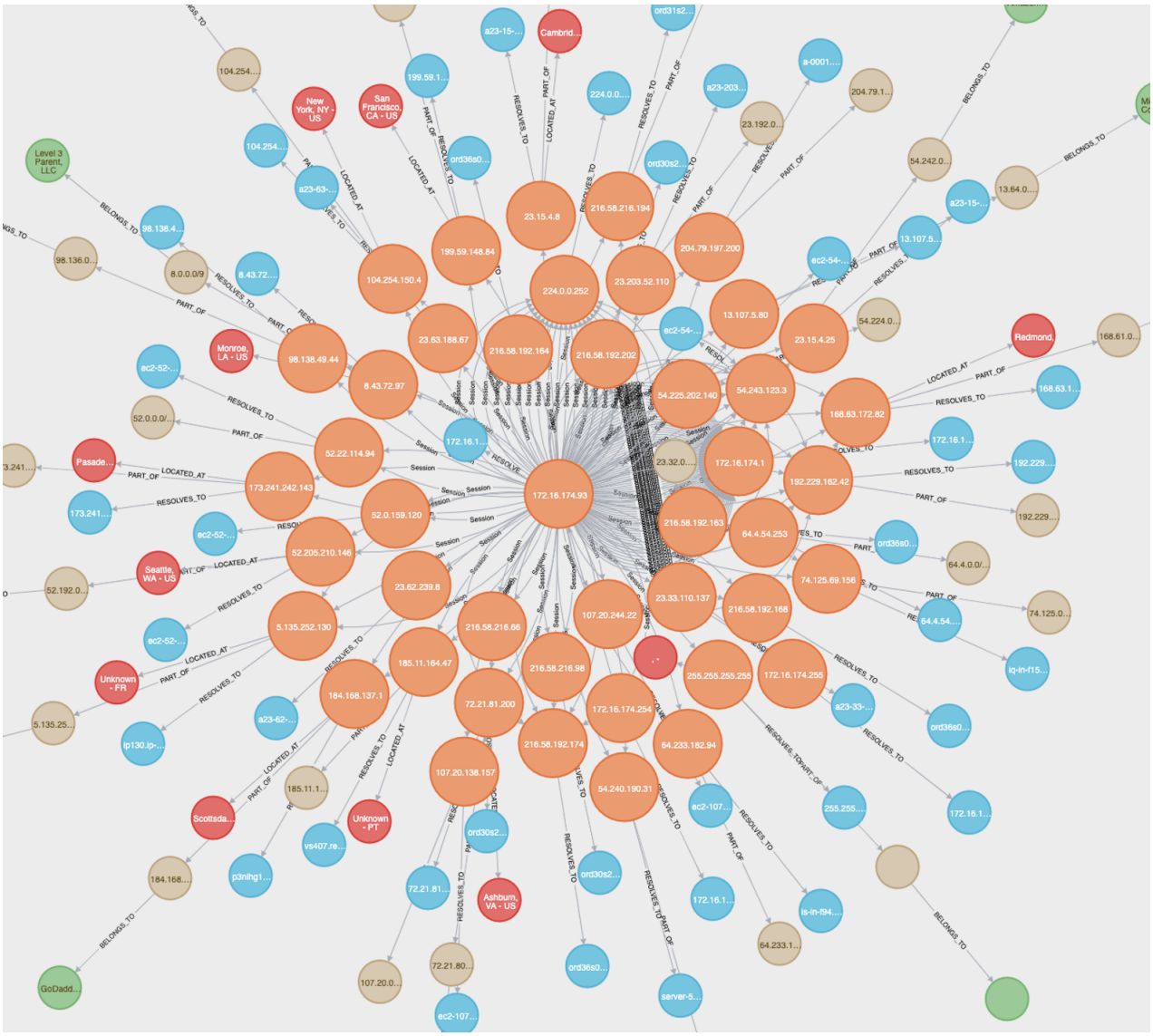
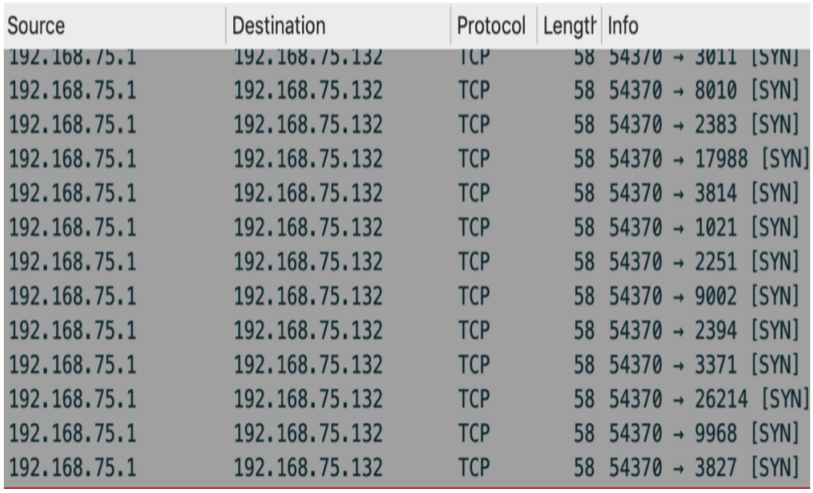



Figure 1: Example visualized AEN graph



| Source | Destination | Protocol | Lengtl | Info |
|---|---|---|---|---|
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 3011 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 8010 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 2383 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 17988 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 3814 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 1021 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 2251 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 9002 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 2394 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 3371 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 26214 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 9968 [SYN] |
| 192.168.75.1 | 192.168.75.132 | TCP | 58 | 54370 → 3827 [SYN] |

Figure 2: An example of port scanning attack

# Chapter 3 Scanning Attacks

Scanning attacks can be classified in 3 major types: vertical scans, horizontal scans and block scans[7].

- Vertical scan: a scan that targets multiple destination ports on a single target host.
- Horizontal scan: a scan that targets the same destination port by sweeping across multiple target hosts.
- Block scan: a hybrid of vertical and horizontal scan.

Dependent on the time frame over which the port scan takes place, it can be classified as slow scan or a fast scan. Fast scans are more common and easier to spot. In contrast slow scans are stealthier and more difficult to detect.

A typical vertical scan called port scanning, is an attack that sends client requests to a range of server port addresses on a host, with the goal of finding an active port and exploiting a known vulnerability of that service. An example PCAP file from NMAP port scanning is shown in 2. We illustrate below the fingerprint for fast vertical scans. The fingerprints for the other types of scanning can be derived by slightly modifying the fingerprint parameter.

Vertical scanning occurs by targeting a specific destination and sweeping across the port space looking for open ports and running services. Common characteristics of fast vertical scanning could be summarized as follows:

- the packets are sent from one source IP to one destination IP
- The packets have a single source port and several different destination ports.

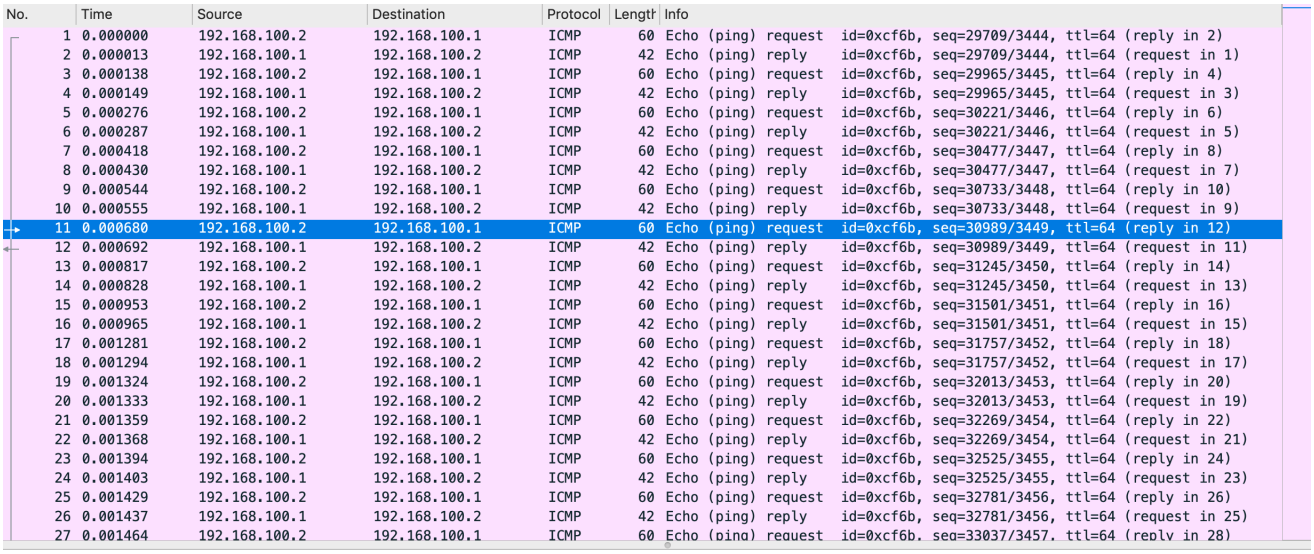
- The packets length/sizes are relatively fixed, and mostly the same.
- The timeframe of one single session is very short.

These patterns can be expressed using the following query language:

Listing 2: Fingerprint for scanning attack

```
SELECT s, d
MATCH (s:HOST)-[e:SESSION]->(d:HOST)
WHERE e.destSize < sizeThreshold
AND e.deltaTime < durationThreshold
GROUP BY s,d
HAVING count(DISTINCT e.destPort) > numThreshold;
```

The above query returns all source host and destination host pairs, which are grouped by a one-to-one relation. There are 3 thresholds used to match the above-mentioned pattern. The size threshold means the size limit of each session, the duration threshold means the duration limit of each session, and the num threshold, which is the most distinctive feature of this fingerprint, means the minimum number of ports on the destination host. In a typical attack, the number of ports would be a large number, while the size would generally be very small otherwise the attack will be too heavy. Also, in the case of a fast scan, the duration would be relatively small.



| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=29709/3444, ttl=64 (reply in 2) |
| 2 | 0.000013 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=29709/3444, ttl=64 (request in 1) |
| 3 | 0.000138 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=29965/3445, ttl=64 (reply in 4) |
| 4 | 0.000149 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=29965/3445, ttl=64 (request in 3) |
| 5 | 0.000276 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=30221/3446, ttl=64 (reply in 6) |
| 6 | 0.000287 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=30221/3446, ttl=64 (request in 5) |
| 7 | 0.000418 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=30477/3447, ttl=64 (reply in 8) |
| 8 | 0.000430 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=30477/3447, ttl=64 (request in 7) |
| 9 | 0.000544 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=30733/3448, ttl=64 (reply in 10) |
| 10 | 0.000555 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=30733/3448, ttl=64 (request in 9) |
| 11 | 0.000680 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=30989/3449, ttl=64 (reply in 12) |
| 12 | 0.000692 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=30989/3449, ttl=64 (request in 11) |
| 13 | 0.000817 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=31245/3450, ttl=64 (reply in 14) |
| 14 | 0.000828 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=31245/3450, ttl=64 (request in 13) |
| 15 | 0.000953 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=31501/3451, ttl=64 (reply in 16) |
| 16 | 0.000965 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=31501/3451, ttl=64 (request in 15) |
| 17 | 0.001281 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=31757/3452, ttl=64 (reply in 18) |
| 18 | 0.001294 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=31757/3452, ttl=64 (request in 17) |
| 19 | 0.001324 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=32013/3453, ttl=64 (reply in 20) |
| 20 | 0.001333 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=32013/3453, ttl=64 (request in 19) |
| 21 | 0.001359 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=32269/3454, ttl=64 (reply in 22) |
| 22 | 0.001368 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=32269/3454, ttl=64 (request in 21) |
| 23 | 0.001394 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=32525/3455, ttl=64 (reply in 24) |
| 24 | 0.001403 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=32525/3455, ttl=64 (request in 23) |
| 25 | 0.001429 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=32781/3456, ttl=64 (reply in 26) |
| 26 | 0.001437 | 192.168.100.1 | 192.168.100.2 | ICMP | 42 | Echo (ping) reply id=0xcf6b, seq=32781/3456, ttl=64 (request in 25) |
| 27 | 0.001464 | 192.168.100.2 | 192.168.100.1 | ICMP | 60 | Echo (ping) request id=0xcf6b, seq=33037/3457, ttl=64 (reply in 28) |

Figure 3: An example of ICMP ping flood attack

# Chapter 4 Denial of Service

Denial of Service (DoS)[1] is a cyber-attack in which the perpetrator seeks to make a machine or network resource unavailable to its intended users by temporarily or indefinitely disrupting services of a host connected to the Internet. Denial of service is typically accomplished by flooding the targeted machine or resource with superfluous requests in an attempt to overload systems and prevent some or all legitimate requests from being fulfilled.

DoS is one of the most well-known and harmful network attacks on today's Internet. There is a wide variety of DoS attacks, which can be categorized according to the specific OSI layer. Most of the attack types are in layer 3 (network layer), layer 4 (session layer) and layer 7 (application layer).

In this section, we will illustrate DoS fingerprinting by focusing on selected attacks under layers 3, 4 and 7 DoS categories.

## 4.1 Layer 3 DoS Attacks

### 4.1.1 ICMP ping flood[20]

ICMP Ping flood is one of the most common DoS attacks. It is aimed at consuming computing power and saturating bandwidth. It is executed by sending a very high volume of ping requests via a boradcast host by spoofing the source IP to match the target IP. Ping responses are sent to the victim continuously until it gets overwhelmed. A PCAP example from ICMP ping flood is shown in Figure3.

The main characteristics of a typical ICMP Flood attack can be summarized as follows:

- The attacker host keeps sending ping requests (i.e. ICMP packets) via a broadcast host to the target host
- The packets length/sizes are relatively fixed, and mostly the same.
- The timeframe of one single session is very short.

The above patterns can be expressed in the following query:

Listing 3: Fingerprint for ICMP Flood DoS attack

```
SELECT s, d, count(e)
MATCH (s:HOST)-[e:SESSION]->(d:HOST)
WHERE e.protocol = 'icmp'
AND e.destSize < sizeThreshold
AND e.startTime > TIMESTAMP 'timewindow_start'
AND e.stopTime < TIMESTAMP 'timewindow_stop'
GROUP BY s,d
```

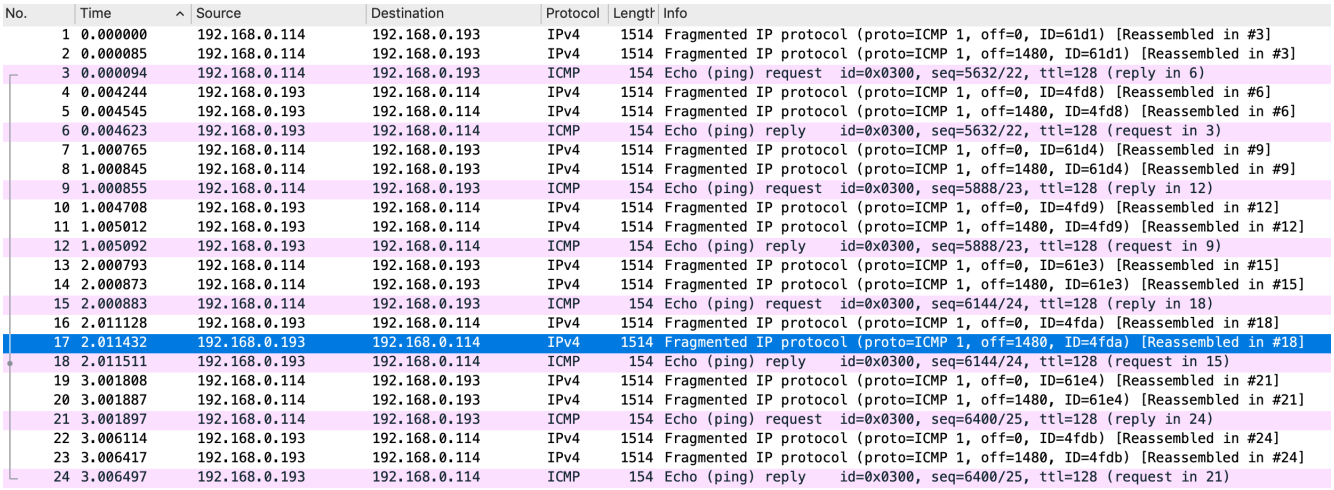
The above query aggregates by source and destination host, which returns hosts with one-to-one relations, and the number of sessions between them. The protocol should be ICMP, and the destination size should be within the size threshold. The time window constraint is applied by letting the start time and stop time within the time window. The results of the query will be further processed by aggregating source hosts with the same destination hosts and applying the number threshold. This will finally give the result for a distributed ICMP ping flood attack.

Unfortunately, the library *io.packets* used in our implementation to parse PCAP files does not provide an ICMP framer, and as result the program unable to identify ICMP packets. Nevertheless, this query should well represent our fingerprint. Should some future work be done to support ICMP packets, the fingerprint could be in use.

### 4.1.2 IP Fragmented attack[2]

IP Fragmented Flood, as the name suggests, is a DoS attack which consists of sending a high volume of fragmented IP packets to the target, aiming at consuming computing power and saturating bandwidth. Also, the target is sometimes likely to crash due to bugs triggered during packet parsing. A PCAP sample of IP fragmented attack is shown in Figure 4.

In most common circumstances, the fragmented IP packets have no identifiable layer 4 protocol, this is the reason why the attack is categorized as layer 4 attack type.



| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 192.168.0.114 | 192.168.0.193 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=0, ID=61d1) [Reassembled in #3] |
| 2 | 0.000085 | 192.168.0.114 | 192.168.0.193 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=1480, ID=61d1) [Reassembled in #3] |
| 3 | 0.000094 | 192.168.0.114 | 192.168.0.193 | ICMP | 154 | Echo (ping) request id=0x0300, seq=5632/22, ttl=128 (reply in 6) |
| 4 | 0.004244 | 192.168.0.193 | 192.168.0.114 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=0, ID=4fd8) [Reassembled in #6] |
| 5 | 0.004545 | 192.168.0.193 | 192.168.0.114 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=1480, ID=4fd8) [Reassembled in #6] |
| 6 | 0.004623 | 192.168.0.193 | 192.168.0.114 | ICMP | 154 | Echo (ping) reply id=0x0300, seq=5632/22, ttl=128 (request in 3) |
| 7 | 1.000765 | 192.168.0.114 | 192.168.0.193 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=0, ID=61d4) [Reassembled in #9] |
| 8 | 1.000845 | 192.168.0.114 | 192.168.0.193 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=1480, ID=61d4) [Reassembled in #9] |
| 9 | 1.000855 | 192.168.0.114 | 192.168.0.193 | ICMP | 154 | Echo (ping) request id=0x0300, seq=5888/23, ttl=128 (reply in 12) |
| 10 | 1.004708 | 192.168.0.193 | 192.168.0.114 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=0, ID=4fd9) [Reassembled in #12] |
| 11 | 1.005012 | 192.168.0.193 | 192.168.0.114 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=1480, ID=4fd9) [Reassembled in #12] |
| 12 | 1.005092 | 192.168.0.193 | 192.168.0.114 | ICMP | 154 | Echo (ping) reply id=0x0300, seq=5888/23, ttl=128 (request in 9) |
| 13 | 2.000793 | 192.168.0.114 | 192.168.0.193 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=0, ID=61e3) [Reassembled in #15] |
| 14 | 2.000873 | 192.168.0.114 | 192.168.0.193 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=1480, ID=61e3) [Reassembled in #15] |
| 15 | 2.000883 | 192.168.0.114 | 192.168.0.193 | ICMP | 154 | Echo (ping) request id=0x0300, seq=6144/24, ttl=128 (reply in 18) |
| 16 | 2.011128 | 192.168.0.193 | 192.168.0.114 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=0, ID=4fda) [Reassembled in #18] |
| 17 | 2.011432 | 192.168.0.193 | 192.168.0.114 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=1480, ID=4fda) [Reassembled in #18] |
| 18 | 2.011511 | 192.168.0.193 | 192.168.0.114 | ICMP | 154 | Echo (ping) reply id=0x0300, seq=6144/24, ttl=128 (request in 15) |
| 19 | 3.001808 | 192.168.0.114 | 192.168.0.193 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=0, ID=61e4) [Reassembled in #21] |
| 20 | 3.001887 | 192.168.0.114 | 192.168.0.193 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=1480, ID=61e4) [Reassembled in #21] |
| 21 | 3.001897 | 192.168.0.114 | 192.168.0.193 | ICMP | 154 | Echo (ping) request id=0x0300, seq=6400/25, ttl=128 (reply in 24) |
| 22 | 3.006114 | 192.168.0.193 | 192.168.0.114 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=0, ID=4fdb) [Reassembled in #24] |
| 23 | 3.006417 | 192.168.0.193 | 192.168.0.114 | IPv4 | 1514 | Fragmented IP protocol (proto=ICMP 1, off=1480, ID=4fdb) [Reassembled in #24] |
| 24 | 3.006497 | 192.168.0.193 | 192.168.0.114 | ICMP | 154 | Echo (ping) reply id=0x0300, seq=6400/25, ttl=128 (request in 21) |

Figure 4: An example of fragmented packet attack

To characterize IP Fragmented attack, we start by checking all IPv4 packets in one session. Practically, if the session contains a high percentage of fragmented packets (i.e., fragmented packets/all packets), there is a great chance that it is part of IP Fragmented attack.

To capture the above characteristic, we use two properties of the session class: the number of fragmented packets and the number of all packets. Upon receiving a new packet in the session, the number of all packets is increased by one. And if the packet is fragmented, increment the number of fragmented packets accordingly.

Listing 4: Getting fragmented packets

```
public boolean addPacket(Packet packet){
  // ....
  packetNum++;
  if(packet.getFragmented()){
   fragmentedNum++;
  }
  //...
 }
```

The common characteritics of the IP Fragmented flood can be summarized as follows:

- A high ratio of fragmented packets over all packets can be observed.
- The packets length/sizes are relatively fixed, and mostly the same.
- The timeframe of a single session is very short.

The above patterns can be expressed in the following query:

Listing 5: Fingerprint for IP Fragmented DoS attack

```
SELECT s, d, count(e)
MATCH (s:HOST)-[e:SESSION]->(d:HOST)
WHERE e.fragmentedPacketCount/e.packetCount > ratioThreshold
AND e.destSize < sizeThreshold
AND e.startTime > TIMESTAMP 'timewindow_start'
AND e.stopTime < TIMESTAMP 'timewindow_stop'
GROUP BY s,d
```

The above query aggregates by source and destination host, and returns hosts with one-to-one relations and the number of sessions between them. The most predominant feature in this fingerprint is the percentage of fragmented packets over all packets in a session. A high ratio predicates a high likelihood of an IP Fragmented attack. Other features extracted are: the total number of sessions and the destination size. The query results will further be processed by aggregating source hosts with the same destination hosts and applying the number threshold and yields a decision of whether a distributed fragmented packets attack has taken place or not.

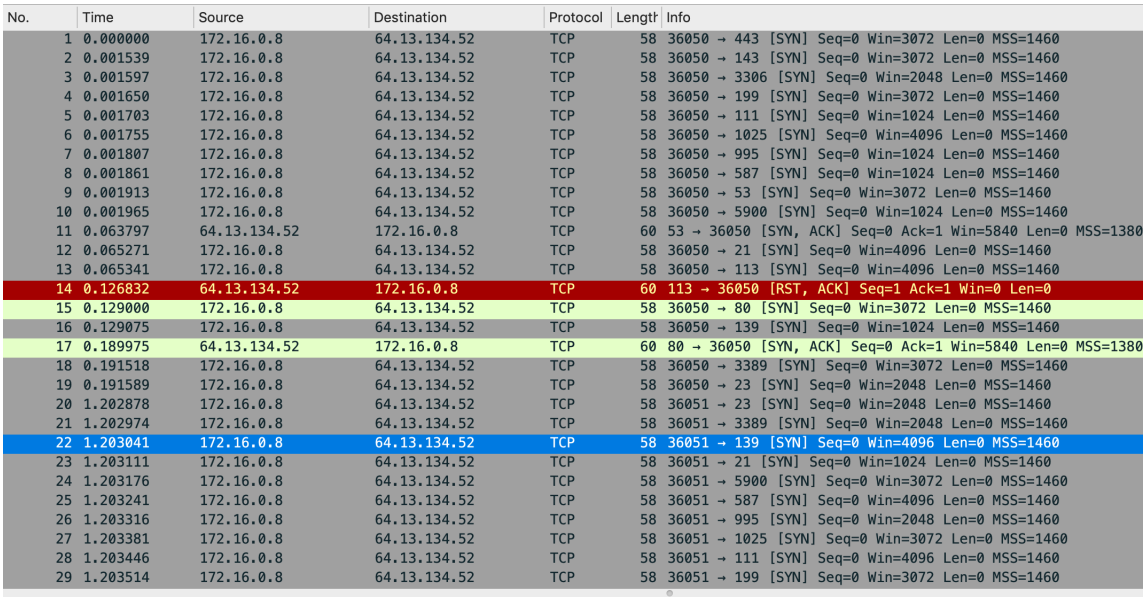
## 4.2 Layer 4 DoS attacks

Layer 4, also called transport layer, is responsible for end-to-end communication over a network. It involves different protocols such as TCP and UDP, which can be abused by attackers to launch a wide range of attacks, including DoS attacks.

### 4.2.1 SYN Flood

SYN flood is a type of DoS attack consisting of sending a large volume of SYN requests to a target host, in an attempt to establish TCP connections, and then decline to acknowledge (ACK) the SYN acknowledgment (SYN-ACK) sent back by the target.[5] As the target server will wait for the ACK packet for some time, the TCP connection buffer will become full and other incoming requests will be dropped creating a denial of service. Some systems may also malfunction or crash when other operating system functions are starved of resources in this way. A PCAP sample of SYN flood attack is shown in Figure 5.

To identify the fingerprint for this attack, we must record the state of the TCP connection, i.e., which stage of the three-way handshake[12] it stays in. Without loss of generality, we can view the establishment of TCP connection as a simple state machine, which involves at



| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 443 [SYN] Seq=0 Win=3072 Len=0 MSS=1460 |
| 2 | 0.001539 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 143 [SYN] Seq=0 Win=3072 Len=0 MSS=1460 |
| 3 | 0.001597 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 3306 [SYN] Seq=0 Win=2048 Len=0 MSS=1460 |
| 4 | 0.001650 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 199 [SYN] Seq=0 Win=3072 Len=0 MSS=1460 |
| 5 | 0.001703 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 111 [SYN] Seq=0 Win=1024 Len=0 MSS=1460 |
| 6 | 0.001755 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 1025 [SYN] Seq=0 Win=4096 Len=0 MSS=1460 |
| 7 | 0.001807 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 995 [SYN] Seq=0 Win=1024 Len=0 MSS=1460 |
| 8 | 0.001861 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 587 [SYN] Seq=0 Win=1024 Len=0 MSS=1460 |
| 9 | 0.001913 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 53 [SYN] Seq=0 Win=3072 Len=0 MSS=1460 |
| 10 | 0.001965 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 5900 [SYN] Seq=0 Win=1024 Len=0 MSS=1460 |
| 11 | 0.063797 | 64.13.134.52 | 172.16.0.8 | TCP | 60 | 53 → 36050 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1380 |
| 12 | 0.065271 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 21 [SYN] Seq=0 Win=4096 Len=0 MSS=1460 |
| 13 | 0.065341 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 113 [SYN] Seq=0 Win=4096 Len=0 MSS=1460 |
| 14 | 0.126832 | 64.13.134.52 | 172.16.0.8 | TCP | 60 | 113 → 36050 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0 |
| 15 | 0.129000 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 80 [SYN] Seq=0 Win=3072 Len=0 MSS=1460 |
| 16 | 0.129075 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 139 [SYN] Seq=0 Win=1024 Len=0 MSS=1460 |
| 17 | 0.189975 | 64.13.134.52 | 172.16.0.8 | TCP | 60 | 80 → 36050 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1380 |
| 18 | 0.191518 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 3389 [SYN] Seq=0 Win=3072 Len=0 MSS=1460 |
| 19 | 0.191589 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36050 → 23 [SYN] Seq=0 Win=2048 Len=0 MSS=1460 |
| 20 | 1.202878 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36051 → 23 [SYN] Seq=0 Win=2048 Len=0 MSS=1460 |
| 21 | 1.202974 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36051 → 3389 [SYN] Seq=0 Win=2048 Len=0 MSS=1460 |
| 22 | 1.203041 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36051 → 139 [SYN] Seq=0 Win=4096 Len=0 MSS=1460 |
| 23 | 1.203111 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36051 → 21 [SYN] Seq=0 Win=1024 Len=0 MSS=1460 |
| 24 | 1.203176 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36051 → 5900 [SYN] Seq=0 Win=3072 Len=0 MSS=1460 |
| 25 | 1.203241 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36051 → 587 [SYN] Seq=0 Win=4096 Len=0 MSS=1460 |
| 26 | 1.203316 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36051 → 995 [SYN] Seq=0 Win=2048 Len=0 MSS=1460 |
| 27 | 1.203381 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36051 → 1025 [SYN] Seq=0 Win=3072 Len=0 MSS=1460 |
| 28 | 1.203446 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36051 → 111 [SYN] Seq=0 Win=4096 Len=0 MSS=1460 |
| 29 | 1.203514 | 172.16.0.8 | 64.13.134.52 | TCP | 58 | 36051 → 199 [SYN] Seq=0 Win=3072 Len=0 MSS=1460 |

Figure 5: An example of SYN flood attack

least 4 states as follows:

- Waiting for SYN-ACK: this means the first step is complete, the source host has sent SYN to the destination host, and the source host is waiting for the destination host to send back SYN-ACK.
- Waiting for ACK: this means the first two steps are complete. In addition to the first step, the destination has sent back SYN-ACK, and the destination is waiting for the source to send back ACK.
- Established: the three-way handshake is complete, and the connection is established. Once established, any incoming packet will not change the state.
- Other: indicates any other scenario such as the first packet of the session isn't a SYN packet.

The core implementation of the state machine is as follows:

Listing 6: TCP connection state machine implementation

```
public boolean addPackdet(Packet packet){
 //...
 if(packetTime.isAfter(getStopTime()) ||
                    packetTime.equals(getStopTime())) {
      switch (tcpStatus) {
```

```
        case OTHER:
          if (packet.isACK()) {
            tcpStatus = TcpStatus.WAITING_FOR_SYN_ACK;
          }
          break;

        case WAITING_FOR_SYN_ACK:
          if (packet.isACK() && packet.isSYN() &&
                  isReverseDirection(packet)) {
            tcpStatus = TcpStatus.WAITING_FOR_ACK;
          }
          break;

        case WAITING_FOR_ACK:
          if (packet.isACK() && !packet.isSYN() &&
                  isSameDirection(packet)) {
            tcpStatus = TcpStatus.ESTABLISHED;
          }
          break;

        case ESTABLISHED:
          // if the TCP connection is established, any
          // incoming packet will not change the state
          break;

        default:
          // other scenarios will be added without
          // affecting the status of tcp
          break;
      }
    }else{
      return false;
    }
  //...
}
```

The pattern in a SYN flood attack involves a large number of sessions that stay at the

state *waiting for the SYN-ACK*. The charactertics of the pattern are summarized as follows:

- The attacker keeps sending SYN packets to the victim and never replies back to the ACK-SYN packet from the victim.
- The packets length/sizes are relatively fixed and mostly the same.
- The time frame of a single session is very short.

The above pattern can be expressed in the following query:

Listing 7: Fingerprint for SYN Flood DoS attack

```
SELECT s, d, count(e)
MATCH (s:HOST)-[e:SESSION]->(d:HOST)
WHERE e.tcpState = Session.TCPState.WAITING_FOR_SYN_ACK
AND e.destSize < sizeThreshold
AND e.startTime > TIMESTAMP 'timewindow_start'
AND e.stopTime < TIMESTAMP 'timewindow_stop'
GROUP BY s,d
```

The query aggregates by source and destination host, and returns hosts with one-to-one relations and the number of sessions between them. The most predominant characteristic in this fingerprint is the high volume of SYN packets from attacker host to victim host. This characteritic is represented in the graph through the number of sessions remaining in the *WAITING_FOR_SYN_ACK* TCP state. Other features extracted are the total number of sessions and the destination size. The results of the query will further be processed by aggregating source hosts with the same destination hosts and applying the number threshold.

### 4.2.2 UDP Flood

In contrast to SYN flood, UDP flood is aimed at UDP datagrams. It differs from TCP based attack in that it doesn't require a connection. A common UDP flood attack can be initiated by sending a large number of UDP packets to random ports on a remote host, eventually making the target unreachable by other clients.[19] A PCAP sample of UDP flood attack is shown in Figure 6.

The key characteristics of UDP food attack are summarized as follows:

- The attacker keeps sending UDP packets to the victim;
- The packets length/sizes are relatively fixed and mostly the same;

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44594 → 80 Len=500 |
| 2 | 0.000391 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44595 → 80 Len=500 |
| 3 | 0.000782 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44596 → 80 Len=500 |
| 4 | 0.001181 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44597 → 80 Len=500 |
| 5 | 0.001580 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44598 → 80 Len=500 |
| 6 | 0.001978 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44599 → 80 Len=500 |
| 7 | 0.002374 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44600 → 80 Len=500 |
| 8 | 0.002770 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44601 → 80 Len=500 |
| 9 | 0.003190 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44602 → 80 Len=500 |
| 10 | 0.003587 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44603 → 80 Len=500 |
| 11 | 0.003983 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44604 → 80 Len=500 |
| 12 | 0.004379 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44605 → 80 Len=500 |
| 13 | 0.004776 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44606 → 80 Len=500 |
| 14 | 0.005172 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44607 → 80 Len=500 |
| 15 | 0.005569 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44608 → 80 Len=500 |
| 16 | 0.005965 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44609 → 80 Len=500 |
| 17 | 0.006357 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44610 → 80 Len=500 |
| 18 | 0.006748 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44611 → 80 Len=500 |
| 19 | 0.007141 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44612 → 80 Len=500 |
| 20 | 0.007535 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44613 → 80 Len=500 |
| 21 | 0.007928 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44614 → 80 Len=500 |
| 22 | 0.008324 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44615 → 80 Len=500 |
| 23 | 0.008719 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44616 → 80 Len=500 |
| 24 | 0.009112 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44617 → 80 Len=500 |
| 25 | 0.009504 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44618 → 80 Len=500 |
| 26 | 0.009896 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44619 → 80 Len=500 |
| 27 | 0.010288 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44620 → 80 Len=500 |
| 28 | 0.010680 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44621 → 80 Len=500 |
| 29 | 0.011074 | 223.251.186.236 | 235.127.67.235 | UDP | 542 | 44622 → 80 Len=500 |

Figure 6: An example of UDP flood attack

- The time frame of a single session is very short.

The above pattern can be expressed in the following query:

Listing 8: Fingerprint for UDP Flood DoS attack

```
SELECT s, d, count(e)
MATCH (s:HOST)-[e:SESSION]->(d:HOST)
WHERE e.protocol = 'UDP'
AND e.destSize < sizeThreshold
AND e.startTime > TIMESTAMP 'timewindow_start'
AND e.stopTime < TIMESTAMP 'timewindow_stop'
GROUP BY s,d
```

Similar to SYN flood, the above query aggregates sessions by the same source and destination hosts. Different thresholds are set for the time frame between two consecutive sessions, packet size, and the number of sessions between two hosts, respectively. The only difference is that the protocol should be UDP instead of TCP.

### 4.2.3 Other transport layer flooding attacks

There are several other flooding DoS attack types, mainly, TCP connection-based attacks. As discussed before, if the first packet of the session is not SYN, we set its state to OTHER. This is the step where other types of attacks happen. Examples of attacks include the following:

- ACK-PSH flood: By continuously sending ACK-PSH packets to a target, stateful defenses can go down (in some cases through a fail-open mode). This type of attack could also be used as a smokescreen for more advanced attacks. This is also true for other out of state flooding attacks.

- PSH-RST-SYN-FIN Flood: PSH-RST-SYN-FIN packets are considered an illegal packet by the original TCP RFC. The attack consists of continuously sending PSH-RST-SYN-FIN packets to a target. Different systems can react differently to these packets and may cause unexpected issues and behavior which could create the condition of denial of service.

- ACK-RST Flood: consists of continuously sending ACK-RST packets to a target.

- URG-ACK-PSH-FIN Flood: consits of continuously sending URG-ACK-PSH-FIN packets to a target.

- URG Flood: consists of continuously sending URG packets towards a target.

All the abovementioned types of flooding attacks have similar patterns, except that different TCP flags are used. Because in principle all the crresponding packets are sent as the first packets of the sessions, we can add more states to our TCP connection so that the different types can be identified. This is done using the following implementation:

Listing 9: Implementation for gettting TCP status

```
public TcpStatus initialStatus(Packet firstPacket){
    if(firstPacket.isPSH() && firstPacket.isRST() &&
        firstPacket.isFIN() && firstPacket.isSYN())
      return TcpStatus.INITIAL_PSH_RST_SYN_FIN;
    else if(firstPacket.isURG() && firstPacket.isACK() &&
        firstPacket.isPSH() && firstPacket.isFIN())
      return TcpStatus.INITIAL_URG_ACK_PSH_FIN;
    else if(firstPacket.isACK() && firstPacket.isPSH())
      return TcpStatus.INITIAL_ACK_FSH;
    else if(firstPacket.isACK() && firstPacket.isRST())
      return TcpStatus.INITIAL_ACK_RST;
    else if(firstPacket.isSYN())
      return TcpStatus.WAITING_FOR_SYN_ACK;
    else if(firstPacket.isURG())
      return TcpStatus.INITIAL_URG;
```

```
    else
      return TcpStatus.OTHER;
  }
```

As a result, for each of the above attack types, we define a template query after replacing the *INITIAL_STATE* with the corresponding state as follows:

Listing 10: Fingerprint for Other flag-based DoS attacks

```
SELECT s, d, count(e)
MATCH (s:HOST)-[e:SESSION]->(d:HOST)
WHERE e.tcpState = Session.TCPState.INITIAL_STATE
AND e.destSize < sizeThreshold
AND e.startTime > TIMESTAMP 'timewindow_start'
AND e.stopTime < TIMESTAMP 'timewindow_stop'
GROUP BY s,d
```

## 4.3 Layer 7 DoS attacks

The most common type of Layer 7 attacks is HTTP flood attack[6]. It is a type of layer 7 DDOS that consists of overwhelming a targeted server with HTTP requests. By saturating the target with a huge volume of requests, it becomes unable to respond to normal traffic; regular requests from actual legitimate users are then denied.

There are two varieties of HTTP flood attacks:

1. *HTTP GET attack:* in this form of attack, multiple computers or other devices are coordinated to send multiple requests for images, files, or some other asset from a targeted server. When the target is inundated with incoming requests and responses, denial-of-service will occur to additional requests from legitimate traffic sources.

2. *HTTP POST attack:* typically when a form is submitted on a website, the server must handle the incoming request and push the data into a persistence layer, most often a database. The process of handling the form data and running the necessary database commands is relatively intensive compared to the amount of processing power and bandwidth required to send the POST request. This attack utilizes the disparity in relative resource consumption, by sending many post requests directly to a targeted server until its capacity is saturated and denial-of-service occurs.

# Chapter 5 Password Guessing

Password guessing is the process of attempting to gain access to a system through the systematic guessing of a valid password.[8] It is often called brute force attack as it keeps trying a sequence of passwords from a dictionary. This type of attack typically creates voluminous amounts of system logs, particularly failed login attempts.

The AEN graph model processes system logs and extracts the information into the graph. This introduces the vertex type *Account*. The connection of an account to a host is represented as *(:ACCOUNT)-[:SESSION]->(:HOST)* in PGQL. Without loss of generality, three types of password guessing fingerprints are investigated: basic, distributed, and stuffing.

- Basic: The basic password guessing scenario occurs when one account targets one host.
- Distributed: The attacker launches the attack from distributed sources, with different accounts, targeting one specific host.
- Stuffing: The attacker uses a single account and repeat the attack over different hosts.

## 5.1 Basic password guessing

This is the most common case of password guessing. Because it only focuses on one-to-one relations, the graph patterns should be Account-AuthAttempt->Host. The predominant feature of password guessing is the high ratio of failed login attempts, so it is necessary to calculate the number of failed login attempts over the whole number of logins. Besides, password guessing does not necessarily happen in a short time frame. Sometimes the whole process could last for days, so there is no need to consider time frame. The following query describes these features:

Listing 11: Fingerprint for basic password guessing attack

```
SELECT s, d, count(e.successful=false)
MATCH (s:ACCOUNT)-[e:AUTHATTEMPT]->(d:HOST)
GROUP BY s,d
HAVING count(e.successful=false) > attemptNumThreshold
AND count(e.successful=false)/count(e) > ratioThreshold
```

The above fingerprint returns three columns: source account, destination host and the number of successful authentication attempts. Two constraints are applied here. The ratio of failed login attempts over the total number of logins should be greater than some threshold.

Also, the number of failed attempts should be greater than a pivot to screen out the regular cases where a user tries multiple times because she is unsure of the password.

## 5.2 Distributed password guessing

This is the case where the attacker launches the attack from distributed sources. The pattern shows several different accounts trying to authenticate on one host. To get the number of attackers, it is ideal to group only by the destination host (victim host) and apply another constraint on the number of different accounts on top of the basic password guessing attack. The query is defined as follows:

Listing 12: Fingerprint for distributed password guessing attack

```
SELECT d, count(DISTINCT s)
MATCH (s:ACCOUNT)-[e:AUTHATTEMPT]->(d:HOST)
GROUP BY d
HAVING count(e.successful=false) > attemptNumThreshold
AND count(e.successful=false)/count(e) > ratioThreshold
AND count(DISTINCT s) > accountNumThreshold
```

The above fingerprint returns two columns: destination host (victim host) and the number of distinct sources. Three constraints are apllied here. In addition to the threshold for the number of failed login attempts and the ratio of failed attempts, there is a new constraint which is the number of distinct source accounts. This latter constraint is the predominant feature of a distributed attack.

## 5.3 Stuffing password guessing

This is the case where the attacker only uses one account to attack different victim hosts. Unlike distributed password guessing, the need arises for grouping attackers. Also, a new threshold is needed for the number of distinct hosts. The query is defined as follows:

Listing 13: Fingerprint for stuffing password guessing attack

```
SELECT s, count(DISTINCT d)
MATCH (s:ACCOUNT)-[e:AUTHATTEMPT]->(d:HOST)
GROUP BY s
HAVING count(e.successful=false) > attemptNumThreshold
AND count(e.successful=false)/count(e) > ratioThreshold
AND count(DISTINCT d) > hostNumThreshold
```

The fingerprint returns two columns: source account (attacker) and the number of distinct victim hosts. Three constraints are apllied here. In addition to the threshold for the number of failed login attempts and the ratio of failed attempts, a new constraint is the number of distinct victim hosts. The latter constraint is where the above attack differs from distributed password guessing.

# Chapter 6 Experimental Evaluation

In this chapter, we present the experimental evaluation of our attack fingerprint scheme using two different datasets: the ISOT Cloud Detection Benchmark Dataset(ISOT-CID) and the 2017 CIC IDS (CICIDS2017)

## 6.1 Using ISOT-CID Phase I

ISOT Cloud Detection Benchmark Dataset(ISOT-CID) consists of over 8 terabytes data combining various logs such as network traffic, system logs, system calls, etc. It involves various attack and benign usage scenarios that were executed in a real cloud environment over several days for hypervisors and several months. The data was collected in two phases: phase 1 in 2016 and phase 2 in 2018. The evaluation presented in this report is based on phase 1 data. Further evaluation will be conducted in the future on phase 2 and other datasets. We were provided a graph-serialized version of the dataset, which contains all the basic information in the graph model, plus two labels: classification (benign or malicious) and attackType. This could be used to evaluate how well our fingerprint performs on a massive dataset.

### 6.1.1 Dataset exploration

The raw dataset was converted into AEN graph. In the graph, the nodes have different properties, among which their types, such as host, account, Domain, etc. Similarly, the edges, which represent the relations between the nodes, have several properties, among which their types, such as session (one set of traffic between two hosts), authentication attempt (an account attempts to authenticate on a host), etc.

The above are the basic properties of the AEN graph model. Apart from that, the dataset also has labeled properties named "classification" and "attackType". The labels are important to us because they are the comparison criteria for the matching results of the fingerprints. To get a general picture of the labels, we extract the labels and their numbers, as shown in tables 1 and 2.

The AEN graph derived from the dataset (ISOT CID Phase I) contains in total 109,415 edges. But in the AEN graph, only the session edges are labelled via their connections to alerts. We could possibly label, for instance, *authenticationAttempt* edges but that labelling would be derived from the session label, meaning that is possible to figure out from the query if the attempt is related to a malicious or a benign session. On the other hand, there is no benefit in labelling, let's say, *belongsTo* edges since those edges only specify that an IP

| classification | count of edges |
|---|---|
| malicious | 30381 |
| benign | 9462 |

Table 1: Edge classification labels and their numbers from ISOT CID Phase 1

| attackType | count of edges |
|---|---|
| port scanning | 26 |
| ping | 1 |
| unauthorized login | 21 |
| password guessing | 79 |

Table 2: Edge attackType labels and their numbers from ISOT CID Phase 1

belongs to an organization. It is not something that is malicious or benign. In short, there is no compelling benefit in assigning to the other types of AEN edges. Among the 109,415 edges, 39,843 are session edges split between benign and malicious edges as listed in Table 1.

Furthermore, part of the data consists of unsollicited attacks, whose specific nature are unknown apriori. Because of that, only a subset of malicious have specific attack type associated with them as shown in Table 2. The remaining malicious have unknown attack type, but they will still be processed using the fingerprint scheme.

We can start by scanning the dataset against all the fingerprints in our knowledge base, and compare with the benign/malicious labels.

### 6.1.2 Overall Results

In our current fingerprint database, as outlined in this report, there are 3 major types of fingerprints: Probing, DoS and password guessing. Despite the fact that we have specialized fingerprints for each major type, we use the generalized fingerprint.

The threshold that we adopted for the experiment are:

- packet size: less than 100
- packet number: greater than 100
- password guessing failure rate: greater than 0.9
- distributed source number: greater than 100
- duration: less than 1

| | |
|---|---|
| benign edges (label) | 9462 |
| malicious edges(label) | 30381 |
| malicious edges(fingerprint) | 19567 |
| true positive (valid detection) | 18631, 60% |
| false positive | 936, 9.9% |

Table 3: Overall result

The results obtained, after running the fingerprints against the dataset, are presented in Table 3.

It can be observed that 19,567 edges are detected as malicious, among which 18,631 are correctly classified (i.e. true positive); 936 benign are wrongly as malicious out 9,462 edges. These result in a detection rate of 60% and false positive rate of 9.9%. The obtained detection rate is promising considering that the fingerprints are designed for specific type of attacks, and not all attack types covered by the dataset were represented by a fingerprint. Normally, the false positive rate should be zero. The current value can be explained by the parameter values used. By tuning appropriately the parameter values, the false positive rate can be lowered considerably.

### 6.1.3 Malicious traffic breakdown

Considering the attacks are either a one-on-one attack or many-to-one distributed attack, we can aggregate the traffic by attacker-victim pair or by victim host. We did the aggregation by victim host because because most of the attacks are distributed. Table 4 shows the breakdown of the malicious edges per monitored host. It can be observed that some monitored hosts are associated with a very small number of edges. However, the typical attacks covered by our current fingerprint are expected to involves several edges. Hence, the monitored hosts wih fewer edges can automatically be filtered out from the analysis. Here the monitored hosts with a number of malicious edges below 50 are ignored. Table 5 shows the results broken down per monitored host after running the fingerprints.

The results can be interpreted in two ways: Firstly, in terms of victim host, 6 out of 8 hosts are detected, which give 2 false-negative cases and 0 false-positive cases. Secondly, the data for each victim host looks normal except 142.104.64.196. It has a significant number of undetected malicious edges: 10560 undetected. It is likely that this contains an attack-type that is not in our knowledge base. Through further investigate, it is discovered that the host 142.104.64.196 is the log server that was being used to collect the logs during the attacks. It is marked as malicious because all communication between the log server and a malicious host is marked malicious. However, in reality it's not likely that the log server would be

| host ip | number of malicious edges |
|---|---|
| 172.16.1.17 | 1 |
| 192.168.0.8 | 1 |
| 172.16.1.10 | 4 |
| 172.16.1.26 | 158 |
| 172.16.1.28 | 11 |
| 172.16.1.24 | 12709 |
| 172.16.1.20 | 36 |
| 142.104.64.196 | 10948 |
| 91.189.91.157 | 5 |
| 23.227.183.204 | 1 |
| 209.126.122.15 | 1 |
| 172.16.1.23 | 5072 |
| 142.104.6.1 | 799 |
| 163.172.210.63 | 1 |
| 172.16.1.16 | 1 |
| 172.16.1.21 | 212 |
| 172.16.1.27 | 191 |
| 192.168.0.9 | 171 |
| 206.12.96.149 | 1 |
| 255.255.255.255 | 2 |
| 172.16.1.19 | 28 |
| 91.189.89.198 | 2 |
| 91.189.94.4 | 2 |
| 93.158.200.232 | 1 |
| 192.168.0.10 | 22 |
| 154.16.132.187 | 1 |

Table 4: Monitored hosts and related malicious traffic

| monitored host | detected | labelled as malicious | alerts by fingerprint | true positive | false positive | |
|---|---|---|---|---|---|---|
| 172.16.1.26 | no | | | | | |
| 172.16.1.24 | yes | 12709 | 12452 | 12445 | 7 | |
| 142.104.64.196 | yes | 10948 | 440 | 388 | 2 | |
| 172.16.1.23 | yes | 5072 | 5029 | 5028 | 1 | |
| 142.104.6.1 | yes | 799 | 705 | 497 | 208 | |
| 172.16.1.21 | yes | 212 | 122 | 103 | 19 | |
| 172.16.1.27 | no | | | | | |
| 192.168.0.9 | yes | 171 | 219 | 170 | 49 | |

Table 5: Performance results broken down per monitored host. Only host with more than 50 malicious edges are considered.

| | |
|---|---|
| benign edges (label) | 9462 |
| malicious edges (label) | 18963 |
| malicious edges(fingerprint) | 18725 |
| true positives | 18243 |
| false positives | 482, 2.5% |
| false negative | 720, 3.8% |
| false positive rate | 5.1% |
| detection rate | 96.2% |

Table 6: Revised global performance

compromised. So we can safely remove the host 142.104.64.196 and the traffic associated with it for better precision.

After removing those edges and recalculating the performance, we obtain the overall results shown in Table 6, which include DR=96.2% and FPR=5.1%.

## 6.2 Using UNB CICIDS2017

### 6.2.1 Dataset Introduction

The CICIDS dataset is provided by Canadian Institute for Cybersecurity (CIC), which is a training and research institute based at the University of New Brunswick(UNB) in Fredericton, known for its research exellence in cybersecurity. It has been building sophisticated and comprehensive network attack datasets over the years.

CICIDS2017 dataset contains benign and the most up-to-date common attacks[17]. It has both the original packet files in pcap format and generated netflow files. The netflow file is CSV formatted, and includes network traffic analysis using CICFlowMeter with labeled flows based on the time stamp, source, and destination IPs, source and destination ports, protocols and attack. The dataset greatly resembles real-world network traffic.

CICIDS2017 dataset covers a comprehensive set of attack scenarios including the following: brute Force SSH, brute force SSH, DoS, heartbleed exploitation, Web Attack, Infiltration, botnet and DDoS. It also covers an extended timeframe: It has a total of 5 days which started at 9 a.m., Monday, July 3, 2017 and ended at 5 p.m. on Friday July 7, 2017. Table 7 Shows the attack scenarios of each day. In regards to the scope of this evalution, data of Tuesday and Thurdays will not be analyzed in this project, because our current fingerprints do not cover application layer attacks or PCAP-based FTP/SSH brute force.

| | |
|---|---|
| Monday | Normal Activity: All benign |
| Tuesday | Normal Activity + attack: FTP-Patator + SSH-Patator |
| Wednesday | Normal Activity + attack: DDoS slowloris, Slowhttptest, Hulk, GoldenEye |
| Thursday | Normal Activity + attack: Infiltration + WebAttack |
| Friday | Normal Activity + attack: Port Scanning + DDoS(LOIT) |

Table 7: CICIDS2017: Attack scenario for each day

### 6.2.2 Netflow file preprocessing

For the sake of efficiency the netflow files are chosen over the PCAP files. The size of the original PCAP data being too large, at about 50 GB, processing it would have quickly drained out memory if no special optimization was applied to the engine. Also it would take huge amount of time while the benefit it gives does not worth the efforts.

The netflow files are in csv format, generated from CICFlowMeter. Each netflow is a collection of network traffic that takes place between two hosts. The concept is similar to the "Session" in the AEN graph, but with more than 80 statistic features: Duration, Number of packets, Number of bytes, Length of packets, etc. What's more, all features are calculated separately in the forward and backward directions.

Although the concept of netflow and session is very similar, we could not directly convert a netflow into a session. Some properties are hard to convert: In session we could determine the TCP state by chronologically analyzing the packets; However, netflow only records the number of packets with SYN/ACK flag. What's more, netflow contains bidirectional network traffic between two hosts, thus keeping track of every statistics with both forward and backward directions. There should be an intermediate process that helps us convert a Netflow edge to a Session edge.

To address this issue, a new type of edge is added to the engine, namely "Netflow". This edge records the exact same properties we read from netflow files. To keep consistency in the program, the functions supporting the process of netflow files are developed as well: CSV parser, netflow reader, and netflow receiver.

Some of the differences of fields in Session edge and Netflow edge are easy to convert. Protocol, source port, destination port, label, timestamp, these could be directly assigned as they are exactly the same in both edge fields. The packet count field in Session edge should be converted by adding the forward packets and backward packets in Netflow edge. Destination size field in Session is obtained by accumulating all packets, whereas in Netflow edge, we need to get this value by multiplying packet count with average packet size.

The tricky field to handle is "TCP State". As explained above, we couldn't determine the TCP state from the properties of Netflow edge. So the Session edges converted from Netflow

| Day | Number of Vertices | Number of Edges |
|---|---|---|
| Monday, benign | 65,207 | 529,918 |
| Wednesday, DDoS | 24,918 | 506,000 |
| Friday, DDoS | 5,779 | 225,745 |
| Friday, port scan | 10,123 | 286,467 |

Table 8: CICIDS2017: Attack scenario for each day

will have the "TCP State" field set as "invalid". This is a special value that indicates this Session edge is converted from Netflow. To make up for this missing information, two new fields: "SYN flag count" and "ACK flag count" are added to the session edge. When the fingerprint examines the TCP State field in an edge, and gets an "invalid" value, it will use the "SYN flag count" and "ACK flag count instead."

The netflow concept mentioned above serves as a Data Transfer Object(DTO). Ideally we would like that the graph only keeps Session edge. The conversion of netflow DTO to Session edge is performed in the constructor of Session, so converting Netflow edge to Session edge won't have edge conversion overhead. When the option "processNetflowAsSession" is set to true, the engine would construct graph with only session edges from netflow data.

After processing the dataset, the resulting graph-speific information is shown in Table 8:

### 6.2.3 Evaluation Results

The three selected datasets are processed one by one, and the corresponding fingerprints were ran against them: A general DDoS fingerprint was ran against friday and wednesday's data, and port scanning fingerprint was ran against friday's data. For Monday, which consists of only benign traffic, all fingerprints were ran against the data. The results are shown in Table 9. The results show acceptable false positive rate and false negative rate. It can also be observed that false positive rate is generally higher than false negative rate, the reason is that our fingerprint tests more general behavior, attacks with specific features get filtered unrecognized.

| dataset | labelled as malicious | alerts by fingerprint | true positive | false positive | false negative |
|---|---|---|---|---|---|
| Monday, benign: Edges | 0 | 38834 | 0 | 38834, - | 0 |
| Monday, benign: Vertices | 0 | 5 | 0 | 5, - | 0 |
| Wednesday, DDoS: Edges | 243492 | 247976 | 243492 | 4484, 1.8% | 0 |
| Wednesday, DDoS: Vertices | 31 | 25 | 25 | 0 | 6, 19.3% |
| Friday, DDoS: Edges | 128027 | 160174 | 128024 | 32150, 20.1% | 3. 5.2% |
| Friday, DDoS: Vertices | 18 | 20 | 17 | 3, 14.2% | 1, 0.0% |
| Friday, Port Scan: Edges | 158930 | 163519 | 158930 | 4586, 2.8% | 0 |
| Friday, Port Scan: Vertices | 22 | 59 | 19 | 50, 69.4% | 3, 12.0% |

Table 9: Evaluation result of Monday, Wednesday, Friday

# Chapter 7 Conclusion

In this report, a collection of fingerprints or patterns are presented for detecting known attacks and variation on these attacks based on the AEN graph model. The fingerprints are constructed based on particular graph pattern, and expressed using the PGQL query language. PGQL provides a mechanism to retrieve results from the graph with queries. However, it has some limitations when used to express attack patterns. We addressed these limitations by developing some new modules. Using the new modules, we developed a series of fingerprints covering known attack types such as scanning, denial of service (DoS) and authentication breaches. The proposed fingerprint model was evaluated using phase I of ISOT-CID.

Our future work will consist of three parts: Improving the performance of the AEN model, improving and extending the proposed fingerprint database to incorporate other known attack fingerprints, and conducting further evaluation using additional datasets, specifically the ISOT CID phase II and the CICIDS 2018 dataset collected by the University of New Brunswick.

Firstly, The current AEN model works by adding nodes and edges to the graph when processing network traffic files. While the graph could be stored in an OPG database, the generation process in in-memory. The current performance only allows us to process around 10000 edges on a 16GB-RAM machine. It can be inferred that a considerable amount of memory is leaked. The Java Virtual Machine (JVM) performance should be improved to get rid of the memory issue.[18]

Secondly, the current fingerprint only covers the most basic ones, and they are mostly in Transport Layer and Network Layer and cover Port scanning for probing, 3 types of DDoS, and 3 types of password guessing. The fingerprints could be extended to support more Application Layer attacks, such as Web brute force attack, SQL injection[11], etc. Analysis of HTTP content could be added to identify different Web attack types.

Finally, the evaluation will be performed on other datasets. ISOT CID phase II consists of more complicated attack scenarios and more comprehensive attack types, which can improve the evaluation of the detection rate. The dataset compiled by Communications Security Establishment and the Canadian Institute for Cybersecurity has other datasets such as CSE-CIC-IDS2018[13]. It consists of greater netflow traffic: 10 days of attack data, with a total size of 200 GB packet files and CICFlowMeter files. These datasets could be used to broaden the scope of our evaluation results.

# References


[1] US-CERT. 6 February 2013. Retrieved 26 May 2016. Understanding denial-of-service attacks.

[2] Antonios Atlasis. Attacking ipv6 implementation using fragmentation. *Blackhat europe*, pages 14–16, 2012.

[3] Dinesh Bajracharya and Nepal Kathmandu. A review on java hashmap and treemap.

[4] Olivier Blazy, Céline Chevalier, and Damien Vergnaud. Mitigating server breaches in password-based authentication: secure and efficient solutions. In *Cryptographers' Track at the RSA Conference*, pages 3–18. Springer, 2016.

[5] Mitko Bogdanoski, Tomislav Suminoski, and Aleksandar Risteski. Analysis of the syn flood dos attack. *International Journal of Computer Network and Information Security (IJCNIS)*, 5(8):1–11, 2013.

[6] Avi Chesla. Generated anomaly pattern for http flood protection, November 10 2009. US Patent 7,617,170.

[7] Marco De Vivo, Eddy Carrasco, Germinal Isern, and Gabriela O de Vivo. A review of port scanning techniques. *ACM SIGCOMM Computer Communication Review*, 29(2):41–48, 1999.

[8] Yun Ding and Patrick Horster. Undetectable on-line password guessing attacks. *ACM SIGOPS Operating Systems Review*, 29(4):77–86, 1995.

[9] International Organization for Standardization (15 November 1989). Iso/iec 7498-4:1989 – information technology – open systems interconnection – basic reference model: Naming and addressing.

[10] World Economic Forum. The global risks report 2018 13th edition.

[11] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, volume 1, pages 13–15. IEEE, 2006.

[12] Jessica A Lopez, Yali Sun, Peter B Blair, and M Shahid Mukhtar. Tcp three-way handshake: linking developmental processes with plant immunity. *Trends in plant science*, 20(4):238–245, 2015.

[13] University of New Brunswick. Cse-cic-ids2018 on aws: A collaborative project between the communications security establishment (cse) the canadian institute for cybersecurity (cic).

[14] Oralce. Oracle labs pgx: Parallel graph analytix.

[15] Oralce. Property graph query language. view data as a graph, discover insights, unlock endless querying possibilities.

[16] Margaret Rouse. Introduction to network security.

[17] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *ICISSP*, pages 108–116, 2018.

[18] Martin J Trotter. Memory leak detection, November 2 2010. US Patent 7,827,538.

[19] Li Xiaoming, Valon Sejdini, and Hasan Chowdhury. Denial of service (dos) attack with udp flood. *School of Computer Science, University of Windsor, Canada*, 2010.

[20] Virendra Kumar Yadav, Munesh Chandra Trivedi, and BM Mehtre. Dda: an approach to handle ddos (ping flood) attack. In *Proceedings of International Conference on ICT for Sustainable Development*, pages 11–23. Springer, 2016.