360Bit Scoping

CSF Group Number: 24

Pedro Guerreiro - 78264
Instituto Superior Técnico
afonso.guerreiro.pedro@gmail.com

Pedro Santos - 78328
Instituto Superior Técnico
pedro.m.duarte@ist.utl.pt

Gonçalo Rodrigues - 78958
Instituto Superior Técnico
goncalo.alfredo.rodrigues@gmail.com

Abstract

The use of peer-to-peer has been increasing over the recent years due to the ease of distribution and low cost of data and files, such as updates distribution (e.g., Windows 10 or online games). The most common and known protocol for such networks is the BitTorrent protocol, in which an user wanting to send or receive files needs to have a BitTorrent client that implements this protocol. A lot of internet traffic comes from collective peer-to-peer networks, but the problem with this is that these networks are used to share illegal data.

1. Introduction

The goal of our tool is to detect and collect evidence of BitTorrent traffic within a given local network, more precisely in a workplace environment. This is motivated by the illegal and copyright-protected content downloaded by employees that may damage the employer institution. The purpose of this report is to explore our solution on finding out who is downloading data using the BitTorrent protocol. In this report we will define the basic architecture and the main components of our tool, as well as our implementation choices. Finally, we will present an objective evaluation of our tool.

2. Basic Architecture

2.1 Deep Packet Inspection Filters (DPI)

The easiest way to identify torrent traffic is to simply look at the contents of the packets. Protocols used by Bit-torrent are detailed and available publicly which makes this task even easier. This component of our system looks at the payload of each packet and outputs whether the packet is part of a torrent stream. It does this by using filters which are detailed below in section 3.1. If the output of a filter is False, it goes to the next filter until of them outputs True or there are no more filters. Once a packet is found, the whole stream is considered as torrent and won't be analyzed anymore. After this component has run, a new capture file is generated with the streams that weren't considered torrent for the next component (SPID) to analyze.

2.2 Statistical Protocol Identification (SPID)

There have been some attempts to obfuscate this traffic so ISPs cannot identify it and block it, using a protocol called Message stream encryption (MSE). Even tough MSE renders deep packet inspection useless, using statistical analysis and even machine learning, patterns in MSE streams can be found. SPID (Statistical Protocol Identification) [Hjelmvik and John 2010] is an algorithm that captures statistical features on streams generates a statistical model for them and then compares them with a pre-trained model of what we are trying to identify. To compute this model, some features (256-byte arrays) are calculated over many streams and they are normalized in order to create a probability distribution. To compare and classify a stream, it is used the relative entropy formula for each feature.

$$\sum_{i=1}^{n} P(i) log \frac{P(i)}{Q(i)}$$

Figure 1. Relative Entropy Formula (P is the vector of observed measurements and Q is the trained model)

An average is taken and if it exceeds a chosen threshold, then it is classified as negative, otherwise it is classified as positive.

3. Implementation

3.1 Chosen Filters

In our tool, filters are used by the DPI component to differentiate torrents from non-torrent streams, and directly involved with the characteristics of the BitTorrent protocol. In this section we specify the filters that we used.

3.1.1 Port Filter

Network ports are the endpoints of communication between two computers. Most of the time, these ports are specific to each application and we use this fact to try to identify in which computer in the network is the BitTorrent application. To do this, we look at the packet source and destination ports and we mark it as a potential torrent if any of this ports are not within a known port range. We assume that if an application uses a port between 0 and 1024 then it is a known application, i.e., it is not a BitTorrent application. Other known ports can be added easily in the code.

3.1.2 Tracker Protocol

In Bit-torrent peers communicate with trackers to get meta-info, such as getting a list of available peers, getting information of a node or just announcing their own information. To do this, it is used a special encoding called "bencoding" that allows the transmission of dictionaries and lists. Another thing we observed is that all messages in this protocol are sent as dictionaries. Here's an example of an error message:

```
error = {"t":"aa", "y":"e", "e":[201, "Error"]}
bencoded = d1:eli201e5:Errore1:t2:aa1:y1:ee
```

To take advantage of this unique encoding, we try to parse it and see if it is a valid message, and if it corresponds to a dictionary. In the positive case, we classify the stream containing this packet as torrent.

3.1.3 Handshake

The peer to peer communication itself is started by a ”handshake” which is composed by the character 19 followed by the string ’BitTorrent protocol’ followed by some metainfo with length-prefixed strings. This is quite easy to identify and it is a very effective filter as long as MSE is not being used. This filter can only be used to identify a torrent stream if the capture file contains the beginning of the communication when the handshake is performed.

3.2 Chosen Features

In the context of SPID, a feature is a distribution of measurements that define a behavior of a stream. Contrary to filters, these features do not care about the content of the packet itself. Instead, these try to fingerprint the behavior of streams and are used to find similarities. To split a pcap into streams we used a plublicly available tool called PcapSplitter [3]. The features used in this work were heavily based on the work of Hjelmvik, E and John, W. [Hjelmvik and John 2010]. In the following sections we will discuss these features and the motivation behind them.

3.2.1 Packet Length Distribution

The Lengths of Packets always have been used as good metric in this area because they give a lot of information about the protocol they belong to. For example, in HTTP most messages sent by the client are relatively small and of predictable size, while most Bittorrent messages are the size of the MTU when transmitting data. If a given stream constitutes of big packets, it is more likely to be a torrent than one that has small packets, and that is what this feature represents. So, we constructed a feature that computes the packet length distribution. We used exponential growing size buckets (128 buckets for each direction) to classify the packets. For example, bucket 0 contains the number of messages with 0 bytes, while bucket 70 contains the number of packets with lengths between 430 and 469.

3.2.2 Accumulated Bytes

Protocols regularly behave similarly in terms of packet sizes and directions in the first few packets of a TCP session, but there is nothing that guarantees that all the ”request - response” data is put in a single packet, this way checking the accumulation of bytes in each direction we can still detect this packets. It is known that BitTorrent’s MSE (Message Stream Encryption) splits this data into several packets in order to avoid automatic classification.

3.2.3 Nibble Popularity

Most internet protocols have predictable payload in the first few packets. Message stream encryption (MSE) as an obfuscation protocol hides these patterns, which provides a differentiation factor for those who don’t. This feature looks at the first 32 nibbles in the first 16 packets and computes the popularity among them for each offset into the payload. Unencrypted protocols which always start the same way such as HTTP will have big popularity ranking for the nibbles in the first few offsets into the payload, which will yield great entropy against the trained model of MSE where the distribution should be uniform.

3.2.4 Byte Frequency

The previously described feature won’t be effective if the messages exchanged do not contain static bytes. If that’s the case but the range of used bytes is limited (for example ASCII bytes) we can measure the frequency of each byte to predict or help predict the protocol being used. Of course as MSE looks random, this feature will in practice try to exclude all messages which favor some bytes more than others.

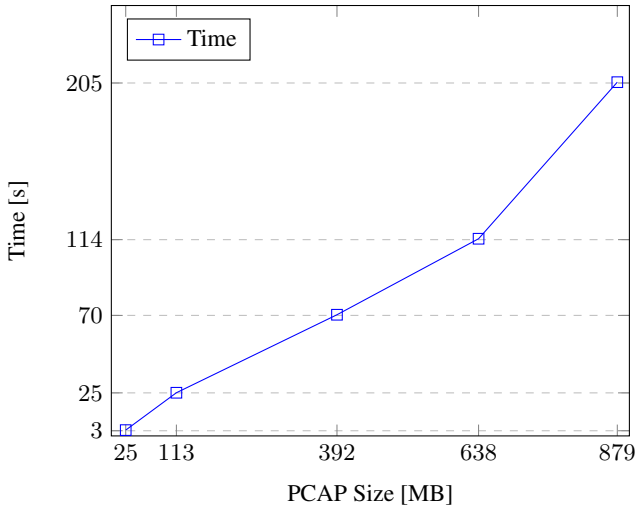

Figure 2. Processing rate

3.2.5 First Bit Positions

This meter tries to catch the behaviour of protocols usually having fixed byte values at fixed offsets into each packet, or making use of a limited set of possible 256 byte values. It does this by looking at the first 16 bytes in the 8 first packets of a session.

3.2.6 Bytes Equality Between First Packets

The motivation for this feature lies in the fact that the first few bytes of the first packets in a stream are the same (static bytes like in HTTP or SSH), while in MSE they are not. We compare every bit at every offset for the first 32 bytes, in the first 4 packets and generate a bit array of truth values. This bit array is converted to an integer and hashed and then counted towards the feature array. So, if the first 2 packets of a protocol always differ in the same bits, the streams of this protocol will have a high entropy relative to the trained model of MSE.

3.2.7 Ordered First Bit Positions

This is similar to 3.2.5 but takes the order of each packet (client-to-server or server-to-client) in account.

4. Evaluation

In this section we evaluate the system in metric which we considered important, such as performance, memory usage and correctness.

4.1 Performance

To evaluate the performance of our tool, we used Python’s time library. For each PCAP file with the sizes listed, we ran 3 times and took the mean of each result. The results are listed in Figure 2. As we can observe, the performance grows linearly with the size of the Pcap, and it is not very performant overall.

4.2 Memory

To evaluate memory usage we ran the python script with different sized PCAP files and measured the peak memory used using the python library memory profiler [2]. As we expected, we can the see in Figure 3 that the memory used is constant, as we do not save any state other than the final output, which is quite small. The pcap is read a packet at a time.

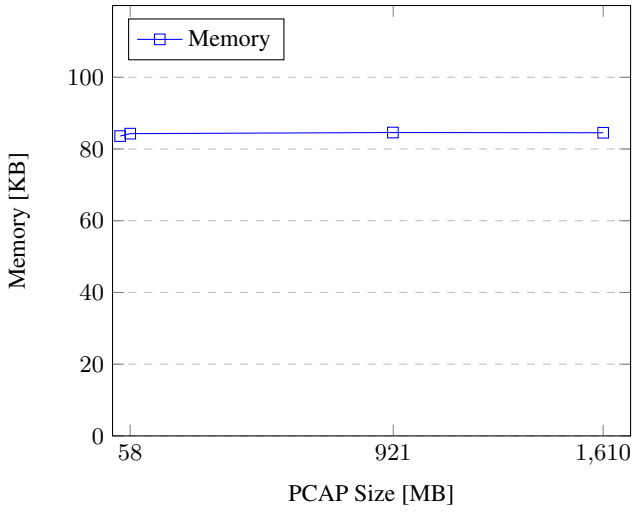

Figure 3. Peak memory used

	Positives	Negatives
Origin	No torrents	∼100% torrents
MB Identified	1.7/580	787.1/882
Streams Identified	24/214	142/156
Packets Identified	2K/500K	913K/950K

Table 1. Test results

	Marked as torrent	Not Marked
Torrent	142	14
Not Torrent	24	190

Table 2. Confusion matrix - Streams

4.3 Correctness

To evaluate the correctness of our solution, we used two PCAPs for testing the false positives and false negatives, and achieved the results shown in Table 1. With this results, we built a confusion matrix as shown in Table 2 and extracted the following scores:

$Precision = \frac{142}{166} = 0.86$

$Recall = \frac{142}{156} = 0.91$

$F - measure = 2 \times \frac{0.86 \times 0.91}{0.86 + 0.91} = 0.89$

Keep in mind that even though a lot of streams were false positives, this corresponds to less than 1% of the total size.

5. Future Work

Despite the good results, we think there is a lot of work to be done. We could improve the techniques used in this paper by using bigger and less noisy datasets and improving the quality and selection of features. We can also better the performance by avoiding reanalyzing the packets, which can be done by changing the architecture. Another area that could lead to promising results is applying machine learning to improve this which would require a slightly different representation of the features.

6. Conclusions

In this report we introduced the architecture of 360BitScope, our BitTorrent analysis tool, which features low memory usage as well as decent time response. During the course of the project we explored the difficult task of analyzing protocols and concluded that behavior analysis is a lot harder than simply looking at the content, not only because its a more complex approach but because getting meaningful training data for BitTorrent is very hard due to the ever-present noisy data when capturing packets.

References

[Hjelmvik and John 2010] Hjelmvik, E and John, W. Breaking and Improving Protocol Obfuscation. Department of Computer Science and Engineering, Chalmers University of Technology Technical Report No. 2010-05, ISSN 1652-926X, 2010.

[2] Memory Profiler, https://pypi.python.org/pypi/memory_profiler

[3] PCapSplitter, https://github.com/seladb/PcapPlusPlus/tree/master/Examples/PcapSplitter

[4] Proxocket, http://www.netresec.com/?page=Blog&month=2011-01&post=Proxocket—A-Winsock-Proxy-Sniffer