Advanced Network Security (CS4155)
Project 5, Q3 2017/2018

Network Flow Anomaly Detection

Abstract
In this project part, you will learn about algorithms to analyze and fingerprint network traffic flows. This is an important skill as during network monitoring or the forensic analysis of an incident you might come across some protocol that is not recognized by you and your toolchains. We will describe a selection of simple algorithms that can aggregate network packets into higher level behavior, and thus allow to fingerprint the characteristic behavior of network protocols to detect anomalies.

In the previous parts of the term project you have implemented packet parsers for common protocols such as DNS, ARP or 802.11w. This allowed your protocol parser to take the raw packet payload and transfer it into an easily readable format allowing you to directly interpret the interaction between hosts and spot transactions which indicated a problem. In practice, when monitoring flows in a network you will observe a variety of protocols, many of which are not parsed by your toolchains. This could simply because a particular protocol has not been implemented so far, or that the network protocol is proprietary and no definitive reference exists on the packet format based on which a packet parser could dissect it. Disregarding unknown or unparseable traffic is however not a desirable option, as this would allow adversaries an easy escape route to avoid detection.

In this part of the term project, we will look into algorithms that learn behavioral characteristics from arbitrary network flows. As you will see in the following, already very basic algorithms can turn trains of individual packets into patterns which allow the creation of fingerprints for certain network applications and by detecting deviations allow an analyst to spot anomalies to investigate further. We will introduce this concept based on fingerprinting of TLS traffic using Markov chains as shown by Korczynski and Duda, and then extend this concept to generate application behavior fingerprints without knowledge of the header structure. We will finally put our system to test by detecting some secret handshake an intruder uses to initiate some communication with a compromised host.

Traffic Fingerprinting of TLS Flows

We will begin our investigation on traffic fingerprinting with the SSL/TLS protocol which we have introduced in chapter 6. As you remember from our discussion, SSL/TLS actually consists out of four subprotocols, each performing one specific function in the tunneling. Which subprotocol is invoked can identified by the first byte in the SSL record which is unencrypted. During the handshake process, when encryption is still being negotiated, we can also peek into the negotiation itself, and the first byte of the handshake protocol message reveals which stages the negotiation is currently in. Table 1 lists the meaning of the handshake protocol messages.

Table 1. The first byte in the SSL record payload belonging to the handshake protocol reveals which stage of the handshake is being performed through the record.

Handshake Message Type	Byte	Decimal
hello_request	0x00	0
client_hello	0x01	1
server_hello	0x02	2
certificate	0x0b	11
server_key_exchange	0x0c	12
certificate_request	0x0d	13
server_done	0x0e	14
certificate_verify	0x0f	15
client_key_exchange	0x10	16
finished	0x14	20

This makes TLS a well suited candidate for our fingerprinting purposes: first, the protocol is clearly defined and the state it is currently in can be determined just by reading at most two bytes in every packet. Second, as the payload is encrypted anyway we can completely ignore for now any steps to mine the application data – and at the same time demonstrate the power of fingerprinting on this minimal information. Our first fingerprinting technique will be based on first order homogeneous Markov chains, meaning that the transition between state i and j does not depend on time and our process does not have any history, thus we only look at how the protocol switches from the current state it is in to the next.

Given the protocol being in state i at a time t we will thus simply count the number of instances the protocol has moved into a state j during the next packet, which we write as $C(X_{t+1} = j | X_t = i)$ or in short $C_{i,j}$. If we repeat this for all possible states (there are only 13 in total in case of our analysis of SSL/TLS), we obtain a transition count matrix

$$C = \begin{bmatrix} C_{1,1} & C_{1,2} & C_{1,3} & \dots & C_{1,13} \\ C_{2,1} & C_{2,2} & C_{2,3} & \dots & C_{2,13} \\ C_{3,1} & C_{3,2} & C_{3,3} & \dots & C_{3,13} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C_{13,1} & C_{13,2} & C_{13,3} & \dots & C_{13,13} \end{bmatrix},$$

which we normalize row-wise by the sum of entries in each row to obtain the transition probabilities $P_{i,j}$ from state i to j:

$$P = \begin{bmatrix} \frac{C_{1,1}}{\sum_{j=1}^{13} C_{1,j}} & \frac{C_{1,2}}{\sum_{j=1}^{13} C_{1,j}} & \frac{C_{1,3}}{\sum_{j=1}^{13} C_{1,j}} & \cdots & \frac{C_{1,13}}{\sum_{j=1}^{13} C_{1,j}} \\ \frac{C_{2,1}}{\sum_{j=1}^{13} C_{2,j}} & \frac{C_{2,2}}{\sum_{j=1}^{13} C_{2,j}} & \frac{C_{2,3}}{\sum_{j=1}^{13} C_{2,j}} & \cdots & \frac{C_{2,13}}{\sum_{j=1}^{13} C_{2,j}} \\ \frac{C_{3,1}}{\sum_{j=1}^{13} C_{3,j}} & \frac{C_{3,2}}{\sum_{j=1}^{13} C_{3,j}} & \frac{C_{3,3}}{\sum_{j=1}^{13} C_{3,j}} & \cdots & \frac{C_{3,13}}{\sum_{j=1}^{13} C_{3,j}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{C_{13,1}}{\sum_{j=1}^{13} C_{13,j}} & \frac{C_{13,2}}{\sum_{j=1}^{13} C_{13,j}} & \frac{C_{13,3}}{\sum_{j=1}^{13} C_{13,j}} & \cdots & \frac{C_{13,13}}{\sum_{j=1}^{13} C_{13,j}} \end{bmatrix}$$

Task 1 *Implement a module for your IPS that monitors the state of each packet belonging to a TLS connection and increments a counter for each transition it observes. At periodic intervals the counter values are turned into probabilities and send to an external file which you can specify through the configuration settings. In order to obtain a clean trace, maintain a separate counter matrix per source-destination IP and port combination. You can accomplish an easy visualization of your transition probability matrix using the command line tool "dot" which you can script from the command line.* ■

When you monitor and aggregate the behavior of different flows over a period of time, you will see that distinct behavioural patterns begin to emerge. Figure 1 shows part of the transition matrix for connections of a web browser with the social media platform Twitter. As you see in the graph each connection attempt always goes through a sequence of client and server hellos, which comes at no surprise given your knowledge of the SSL/TLS protocol. In this example you will see that the client application on the local host can in the majority of the cases cut the handshake short by resuming a previous session. When the connection is negotiated, consecutive packets will only with a low probability contain application data, but most sessions will terminate rapidly. Depending on the type of application and its use of the network, these transition probabilities do significantly vary per application, allowing you to obtain a behavioral characteristic of a network protocol.

Task 2 *Prepare fingerprints for at least 5 different applications. Plot the fingerprints you have obtained and compare the transition graphs of these applications to identify commonalities and differences. What can you say about how these services use the network?* ■

As the fingerprints between applications demonstrate significant differences, we will now try to extend the fingerprint extraction towards application recognition. We will again discuss the simplest possible mechanism here, which you can shape and evolve to obtain better detection. In order to match and compare behaviors which are defined by the transition probability matrix P, we would first need a distance metric d that can capture the similarity between two matrices P^A and P^B. There are multiple techniques for this, depending on the characteristics of the data. For now we compute the distance as the quadratic distance of the individual elements of the two matrices,

$$d(P^A, P^B) = \sqrt{\sum_{i=1}^{k} \sum_{j=1}^{k} (P^A_{i,j} - P^B_{i,j})^2},$$

and given an unknown behavioral pattern, we will select the closest matching known fingerprint on file as the candidate application that has generated the network flow. If you have the time, you can also experiment with better classifiers as well as other distance metrics such as matrix norms and compare their performance.

Task 3 *Collect for each of your applications at least 30 traces of 15 seconds each. Implement the distance metric and perform a 5-fold cross validation, returning for each item the closest match from the training data. Report the performance of your trivial classifier, listing the true positive rate and true negative rate.* ■

Although the first order Markov chain could already deliver impressive fingerprinting results, you will find this technique's inability to incorporate history into the prediction somewhat limiting. After all, it does make a difference whether a protocol stays in one particular state for long durations, or whether it transitions quickly to the next one. The next level of generating simple fingerprint models that does consider past transitions are n-grams. n-grams consider sequences of n transitions as a state, if we analyze the example of figure 1 as a 2-gram we would find a transition from $[22/1, 22/1] \rightarrow [22/11, 22/16]$. Thus, we could also call the first order homogeneous Markov chain a 1-gram. While the 2-gram approach would not provide much benefit in the beginning of the graph, it provides more insight into the more complex transition patterns around the application data protocol state. We can now distinguish sequences of up to four application data packets in a transaction as $[23, 23] \rightarrow [23, 23]$. As n grows and the transaction gain a longer history horizon, the fingerprints will become more specific and sharply tuned to a particular application, which is useful to more reliably identify it in a large corpus and also detect slight variations in application behavior. There is however a trade-off, as we over-extend the inclusion of past states, we will run the risk of overfitting and training the fingerprint database on exactly the exchanges previously recorded instead of obtaining a general, application-specific characteristic.

Protocol Analysis

As you have seen, already the sequence of header fields allows some significant insight into the behavior of a protocol and may be used to create a distinctive fingerprint. For now, what exactly happens inside the SSL/TLS tunnel is hidden to us, but we will revisit this issue later in chapter **??** and see strategies an IPS can use to gain further insights into the content of an encrypted connection. We will now turn our attention beyond basic fingerprinting of an application flow based on the transport layer header, and try to learn functional characteristics of

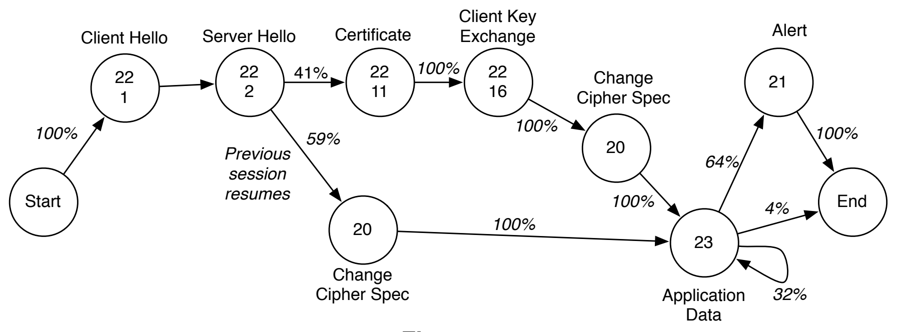

Figure 1

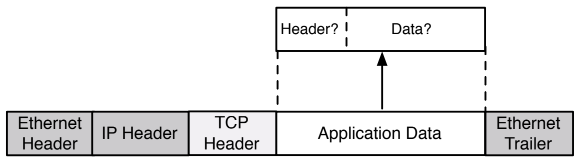

Figure 2. As packet headers are managed by the OS, profiling of application behavior must rely predominantly on the application data payload from which patterns have to be detected and extracted.

the application layer protocol from a network flow. In order to keep the scope manageable for this assignment, we will turn to a rather verbose protocol, the Post Office Protocol (POP), and assume for now that we would have to gain insight into its behavior without knowing any detail about its design.

If we want to characterize the semantic behavior of an application, we have to realize that most of the header fields we have analyzed so far are not suited for this task. As an application opens a socket to send data over the network, the operating system's networking stack and hardware transparently adds and removes all layer headers and trailers, meaning that most of these values are actually not under the control of the application itself. While certainly useful to single out individual connections and groups of flows, from a perspective of application behavior network stack headers turn out to be irrelevant. Although not directly permutable, only the TCP and UDP header are under some level of control from the application. As it opens and closes an end-to-end connection, this behavior becomes visible in the TCP header through the flags that negotiate and tear down the tunnel. Whenever a socket is opened or closed, we can also infer based on the source port number, which is dynamically allocated and likely changes.

As shown in figure 2, an investigation thus has to mainly focus on the application data payload, of which we might know nothing. Since the host application however needs to make sense of the incoming data as well, we can assume that the application payload will contain some structure by itself, either horizontally within the same packet as part of the data is reserved as an application protocol header or temporally as the meaning of a packet can be determined from those before it. Let us for now assume that the application data does contain

```
< +OK POP server ready
> USER mike
< +OK
> PASS thisisincleartext
< +OK
> LIST
< +OK
< 1 130
< 2 100
< 3 80
< .
> RETR 1
< server sends message 1
< +OK
> RETR 2
< server sends message 2
< +OK
> QUIT
< +OK
```

some form of header or control bytes that indicate how to deal with the remaining payload.
This could in principle also be accomplished in two ways: one or more bytes at specific positions in the packet could be reserved for signaling purposes, or specific keywords or markers in the payload itself describe meaning of the remaining data.

The listing on the left shows an example exchange between a client and a server using the Post Office Protocol, where messages sent by the client are marked by $>$, those sent by the server through $<$. While the application layer protocol runs in plaintext and there are apparently no reserved status bytes, we do immediately see keywords that add semantics to the data that follows. Instead of giving in to the temptation to write a protocol parser based on the keywords we spot in such a session, we will write a very simple algorithm that finds control tags automatically and structures flows accordingly. When we collect all the inputs sent by the client to the server and cluster transmissions based on similarity, we will find that strings such as “RETR 1” and “RETR 2” are grouped together, while other clusters contain elements such as “LIST” and “QUIT”. When we combine the elements of a cluster through an exclusive-OR, we detect those bits that are shared across all items as those bits are set to 0 in the output. This allows us to determine that the tag “RETR” is a significant control element in the protocol, while the digit indicating the message number depends on the session. This principle also works when applied to n-grams. In response to a “LIST”, the server will always respond in a sequence of "+OK", followed by the incrementing integers up to the length of the n-gram and the maximum number of emails encountered so far on the mail server. Comparing the chain of elements for each observed session with each other, the parts that vary per instance get dropped, leaving only the integers 1, 2, 3 etc. but not the message size. While until now the algorithm actually also learns some incorrect control words such as "USER mike", these patterns get further refined as we zoom out from the flows of one particular user and compare sets across sessions. As we expect any data other that the static control words at the fixed positions to be random, these will average out on the long run.

> **Task 4** *For this assignment you are provided with a pcap file containing the POP sessions between the clients in a local network and the central mail server. Implement the algorithm outlined above to detect the protocol's current state, and train a first order homogenous Markov chain on the commands sent by the client. Draw your findings in a graph.* ■

Covert Channel Detection

In this part, you will put your new skills to practice. Your organization's incident response team has detected a compromise as internal confidential information that was stored on the internal network storage was leaked to the public. The investigation results so far indicate that the breach was done remotely via the network, but a netflow listing of the network traffic has not revealed any suspicious connection attempts made from the outside. Luckily, the suspected begin of the breach is still within the time window for which the pcaps data for the network are still available.

> **Task 5** *Using the software you have built until now, extract from the pcap files interaction patterns per host and per application. Compare the fingerprints with each other to detect any machine that behaves in an unusual way, and based on these anomalies find how the adversary communicates with the compromised hosts and how the data is transmitted to the outside. Describe your procedure and your findings in a short incident investigation report.* ■