The SPID Algorithm

Statistical Protocol IDentification

Erik Hjelmvik

Gävle, Sweden. October 2008

Abstract

Identifying which application layer protocol is being used within a network communication session is important when assigning Quality of Service priorities as well as when conducting network security monitoring. Currently most protocol identification is performed through signature matching algorithms that rely on strings or regular expressions as signatures. This report presents a protocol identification scheme called the Statistical Protocol Identification (SPID) algorithm, which reliably identifies the application layer protocol by using statistical measurements of flow data as well as application layer data. The SPID algorithm utilises Kullback-Leibler divergence measurements to compare probability vectors created from observed network traffic to probability vectors of known protocols.

Acknowledgements

I am grateful for the valuable feedback on this report that has been provided by Wolfgang John, a PhD student at the Computer Communications and Computer Networks group at Chalmers University. I would also like to thank Jörgen Eriksson, of the Swedish Internet Infrastructure Foundation, for his work to ensure that this report is made publicly available so that it can support future Internet related research and development. I am also grateful for the project funding, granted by the Swedish Internet Infrastructure Foundation, which provides me with the possibility of performing the research presented in this report. Last but not least I would like to thank my wonderful wife Sara for supporting and motivating me in the effort of inventing the SPID algorithm and writing this report.

About the Author

Erik Hjelmvik is an independent network security researcher and open source developer. He holds a M.Sc. degree in engineering from the Royal Institute of Technology (KTH) in Stockholm, Sweden. In the past, Erik served as an R&D engineer at one of Europe's largest electric utility companies, where he worked with IT security for SCADA and process control systems.

Erik is also the creator of the network forensic analysis tool NetworkMiner, which is freely available as open source at SourceForge.net.

About the Sponsor

The Internet Infrastructure Foundation (.SE) is responsible for the top-level Swedish Internets domain, .se. The core business is the registration of domain names and the administration and technical operation of the national domain name registry, at the same time as .SE promotes the positive development of the Internet in Sweden.

.SE is an independent public utility standing on two legs: domain name operations and development of the Internet. The surplus from the registration of domain names is used to finance projects that contribute to the Internet's development in Sweden. Financing projects that contribute to the development of the Internet is a prerequisite for the operations, according to the Foundation's Statutes.

.SE is working hard to be an active research and development financier and player within Internet development. The efforts are intended to benefit domain registrants. .SE has established a long-term objective that financing projects for research and development annually will total SEK 25 million as of 2009.

Table of Contents

1 Introduction

Network intrusion detection systems (NIDS) analyse application layer data in order to detect illicit network traffic, such as attacks and security policy violations. Different analysis engines (protocol parsers) need to be used depending on which application layer protocol is being used in a network communication session. Most NIDS deduce which application layer protocol is being used in a session by using the list of well-known ports assigned by the International Assigned Number Authority (IANA) (Dreger et al., 2006). Other solutions that often use port numbers to deduce the application layer protocol are systems that assign Quality of Service (QoS) priorities and traffic shaping algorithms.

Several studies have, however, reported that only 50-70% of the Internet traffic is classifiable using the IANA port number list (Moore and Papagiannaki, 2005) (Madhukar and Williamson, 2006). One reason for this is that some applications do, for example, try to avoid detection from network security monitoring solutions by not using standard port numbers (Dreger et al., 2006) (Haffner et al., 2005). Peer-to-peer (P2P) protocols belong to one such traffic type that intentionally uses random port numbers for communication (Karagiannis et al., 2005) in order to evade traffic filters as well as legal implications (John and Tafvelin, 2008).

Backdoors, installed by hackers on compromised systems, are often disguised by using other ports than the well-known IANA assigned ports for Telnet, Rlogin, SSH or whatever protocol the backdoor is using (Zhang and Paxson, 2000). There are also several other applications that leech on well-known ports of other protocols for their communication (Li and Moore, 2007), often with the purpose of traversing firewalls (Haffner et al., 2005). Some of the more commonly leeched-on TCP ports are therefore ports 80 (HTTP) and 443 (HTTP over SSL/TLS).

1.1 SPID Overview

This report presents a scheme designed to perform protocol identification based on statistical measurements of various protocol attributes. The concept of protocol identification is also known as application identification (Haffner et al., 2005) (Bernaille et al. 2006) (Juniper Networks), Port Independent Protocol Identification (Bejtlich, 2006), Protocol Discovery (Cisco, 2006) and Application Recognition (Cisco, 2007).

The herein described protocol identification algorithm, which is developed by Erik Hjelmvik, is called the Statistical Protocol IDentification (SPID) algorithm. The SPID algorithm is designed to reliably identify which protocol is being used in a network communication session. Key requirements for the algorithm are:

1. Small protocol database size
2. Low time complexity
3. Early identification of the protocol in a session
4. Reliable and accurate protocol identification

The motivation for these requirements 1 and 2 are that it shall be possible to run the SPID algorithm in real-time on an embedded network device, which has limited memory and processing capabilities. The motivation for requirement 3 is that it shall be possible to use the results from the SPID algorithm in a live traffic capturing environment to automatically take measures in order to, for example, provide quality of service (QoS) to an active session, block

illicit traffic or store related traffic for off-line analysis[1]. Haffner et al. (2005) also point out the need for enterprises to degrade P2P services (via rate-limiting, service differentiation and blocking) on their networks in favour of the performance for business critical applications. I have therefore required that the protocol must be identifiable, with the SPID algorithm, based on only the four first application data packets (i.e. not counting the flow control signalling packets) in a session. Requirement 4 does not need any further motivation than the obvious "in order to provide a high quality service".

The SPID algorithm performs protocol identification by using statistical fingerprints. The statistical fingerprints, of a session or known protocol, are part of an object called "Protocol Model". The application layer protocol in a session is identified by comparing its protocol model to protocol models of known protocols.

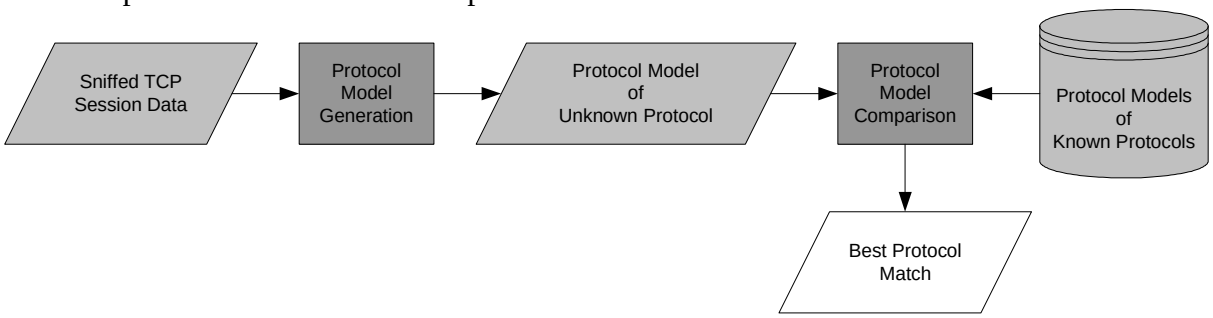

Figure 1 Protocol Identification Data Flow

The SPID algorithm present a reliable method for identifying protocols based on both flow and application data. The strong identification feature is a result of the innovative protocol attribute metering functions, the protocol models' rich data format and the model comparison algorithm.

The SPID algorithm does not require any manual creation of protocol signatures; but it does require training data that is pre-classified based on protocol, in order to generate a protocol model database. The protocol model data format allows for the protocol models to be updated as new training data becomes available, without having access to the previously used training data.

2 Related Software Applications

2.1 L7-filter

L7-filter (l7-filter.sourceforge.net) is an open source project which focuses on identifying the application level (layer 7 in the OSI reference model) protocol. It does so by using manually crafted pattern files, which contain regular expressions (RegEx) for the protocols they are designed to identify. There are at the time of writing somewhere over 100 RegEx pattern files available at the L7-filter project page.

L7-filter is the de-facto open source application used in order to identify the application layer protocol without relying on IANA-assigned port numbers. The L7-filter application does, however, have some shortcomings such as being prone to produce false positives. The L7-filter project's wiki page (www.protocolinfo.org) does, for example, mention the following statement regarding the protocol pattern for the P2P protocol eDonkey:

[1] Performing off-line analysis of traffic is useful in, for example, network forensic investigations

"This is a difficult protocol to match with regular expressions. The l7-filter pattern will falsely identify about 1% of random data as eDonkey."

Patterns that makes extensive use of wildcards and logic operators tend to require a great deal of processing power, according to the L7-filter Pattern Writing HOWTO. Hence, the required time to compare network data to a protocol pattern is not deterministic and might vary a great deal depending on how the pattern is written.

Another problem with the L7-filter patterns is that the fingerprints have to be manually created, which means that network traffic and/or protocol specifications need to be studied and abstracted in order to create a reliable protocol identification pattern. The task of writing patterns that minimise the required processing power, when performing pattern matching, increases the manual labour furthermore.

2.2 Fl0p

Michal Zalewski has developed a proof-of-concept application, called Fl0p, which is designed to extract interesting information from network flow data. The input data to the Fl0p application is flow statistics, such as packet sizes and packet inter-arrival times (i.e. without inspecting application payloads). Fl0p can be used in order to, for example, tell users apart from automated bots even for encrypted protocols.

The fingerprint database for Fl0p does unfortunately only contain a very limited set of fingerprints, but there are some independent initiatives to create additional fingerprints for Fl0p (C.S. Lee, 2008). A fair amount of manual labour is required in order to generate a Fl0p fingerprint, just as for L7-filter.

2.3 PISA

PISA is a yet-to-be-publicly-released open source application, which uses application layer data (application data entropy) as well as flow statistics (packet sizes and packet inter-arrival times) to identify the application layer protocol (Dhamankar and King, 2007). The PISA application is developed by Rohit Dhamankar and Rob King of TippingPoint, and was first presented at Black Hat USA 2007.

PISA uses a 10-dimential representation of each fingerprint, where the dimensions are made up of average values and standard deviations of packet sizes and response times as well as the Shannon entropy of the application layer data. One nice feature of PISA is that protocol fingerprints can be generated automatically directly from pre-classified pcap files.

The PISA implementation uses a rather slim representation for the fingerprints, i.e. only average values and standard deviations, instead of using a more rich representation such as a probability density function or a probability vector. PISA also does not utilise several inherent properties of many protocols, such as consistently having the same byte values at a fixed offset in each packet as well as the order and direction in which the packets in a session are exchanged between the client and server. Because of these limitations the protocol of an observed session cannot be reliably identified by using PISA until some hundreds of packets have been observed. Fairly good protocol detection could, according to the creators of PISA, however be performed after just a dozen packets, given a large enough training set.

2.4 Proprietary Implementations

There are several commercial solutions that perform protocol identification, some examples are NetScout's Sniffer Application Intelligence (NetScout, 2008), Cisco's NBAR (Cisco, 2007) and Juniper's Application Identification. There is unfortunately very little information available regarding how the protocol identification is performed in these proprietary systems. A qualified guess is, however, that the implementations make use of string matching in a similar manner as L7-filter.

3 Related Academic Research

3.1 Traffic Classification Research

Traffic classification is the science of automatically assigning a traffic class, such as P2P, web, email, chat or gaming, to a network session. The classification algorithms do in most cases make use of flow statistics (Erman, 2007) (Crotti et al., 2007a) (Auld et al., 2007) (De Montigny-Leboeuf, 2005) (Bernaille et al. 2006), such as duration, bytes transferred, packet inter-arrival time and packet size.

Using flow statistics rather than inspecting full packet data (including application data) has several advantages:

- Network flow data can easily be extracted from most network routers
- Handling and manual inspection of network flows does not intrude on the privacy of the network users to the same extent as when application data is handled
- Network flows require a significant lesser amount of data compared to full content packet data

The algorithms used for traffic classification are in most cases based on linear approximation algorithms, such as naïve Bayes and Hidden Markov Models.

A shortcoming of many traffic classification algorithms is that they cannot reliably identify the traffic type until all, or at least several thousand packets (Erman, 2007), of the traffic from a network session has been analysed.

Some researchers use connection patterns (Kargiannis et al., 2005) (John and Tafvelin, 2008) in order to tie an IP-port pair to a traffic category. These methods make use of host behaviours such as number of connected hosts, if the host is a service provider or consumer and which other IP-port pairs the host is connected to. This is a powerful method since the classification can be based on several flows rather than a single session or flow. One drawback of using connection patterns is that the algorithm with time will be required to hold an unreasonably large database of IP-port information. Connection pattern classification schemes can also not be used if a single session is to be analysed, without having access to additional data about the hosts' connection pattern behaviours.

It should also be noted that the classification methods used for traffic classification are based on a simplistic model where there are only 4 to 9 different types of traffic classes to choose from. The task of identifying the actual protocol of a network session, on the Internet or on a local area network, involves choosing the correct protocol out of a set of several hundreds of protocols.

3.2 Protocol Identification Research

The field of protocol identification seems to have received less attention, in the academic research community, compared to the field of traffic classification. This is surprising since the problem of traffic classification can effectively be reduced to the problem of protocol identification; hence a protocol identification algorithm can be used to provide traffic classification, but not vice versa. Protocol identification does also have a larger practical value for real-world applications compared to the more abstract task of performing traffic classification.

The research performed on protocol identification often rely on application data matching, either by using manually created payload byte-pattern signatures (Dreger et al., 2006) or by automatically creating application layer byte-level signatures (Haffner et al. 2005) (Ma et al. 2006). Others (Bernaille et al. 2006) (Crotti et al. 2007b) use flow statistics, such as packet-size and inter-arrival time, to detect the protocol used in a session.

4 SPID Algorithm Details

The SPID algorithm performs protocol identification by comparing fingerprints of the investigated session to pre-calculated fingerprints of known protocols. An open source proof-of-concept implementation of the SPID algorithm will be made available on SourceForge.net. The proof-of-concept application only allows for protocol identification of protocols which use the TCP protocol as transport layer. The proof-of-concept application does therefore make use of the 5-tuple[2] to identify the bi-directional flow[3], which constitutes the TCP session. The SPID algorithm can, however, be used to identify protocols in any communication scheme where there is a notion of a session, i.e. a bi-directional flow. This implies that the SPID algorithm can (with various success) be used also to identify protocols that are transported or tunnelled within protocols such as UDP, HTTP, NetBIOS, DCE RPC, ISO 8073 or even SSL.

4.1 Fingerprint Data Format

All fingerprints are represented in the form of probability distributions. This means that the data for each fingerprint is represented by two arrays (vectors) of discrete bins; one array of counter bins and one array of probability bins. These two arrays are in this report referred to as being a counter vector and a probability vector.

The values of the counter vectors are non-negative integer values, where the value at each index represents the number of times the observations (analysed packets) have triggered that particular index.

The probability vectors are a normalised version of the counter vectors, where the values of the probability vectors are values between 0.0 and 1.0. The sum of all values in every probability vector is always equal to 1.0.

Index	0	…	79	80	81	82	83	84	85	…	255
Counter vector	1263	…	715	935	296	919	1056	1845	643	…	1434
Probability vect.	0.006	…	0.003	0.004	0.001	0.004	0.005	0.009	0.003	…	0.007

Figure 2 Fingerprint data example displaying byte frequency for HTTP

[2] A 5-tuple is a set of: source IP, source port, destination IP, destination port and transport protocol
[3] A bi-directional flow consists of the data sent in both directions for a specific 5-tuple

The data format used in the examples of this report uses vectors of length 256 for the counter and probability vectors; an implementation of the SPID algorithm can, however, use any length for these vectors.

4.2 Protocol Attribute Meters

The SPID algorithm makes use of several different protocol attribute metering techniques in order to convert packet-level data into the previously described fingerprint data format. These "attribute meters" are designed to make use of network flow data – such as packet size distribution and packet direction distribution – as well as application level data – such as byte frequencies, reoccurring byte-sequences and offsets for common byte-values.

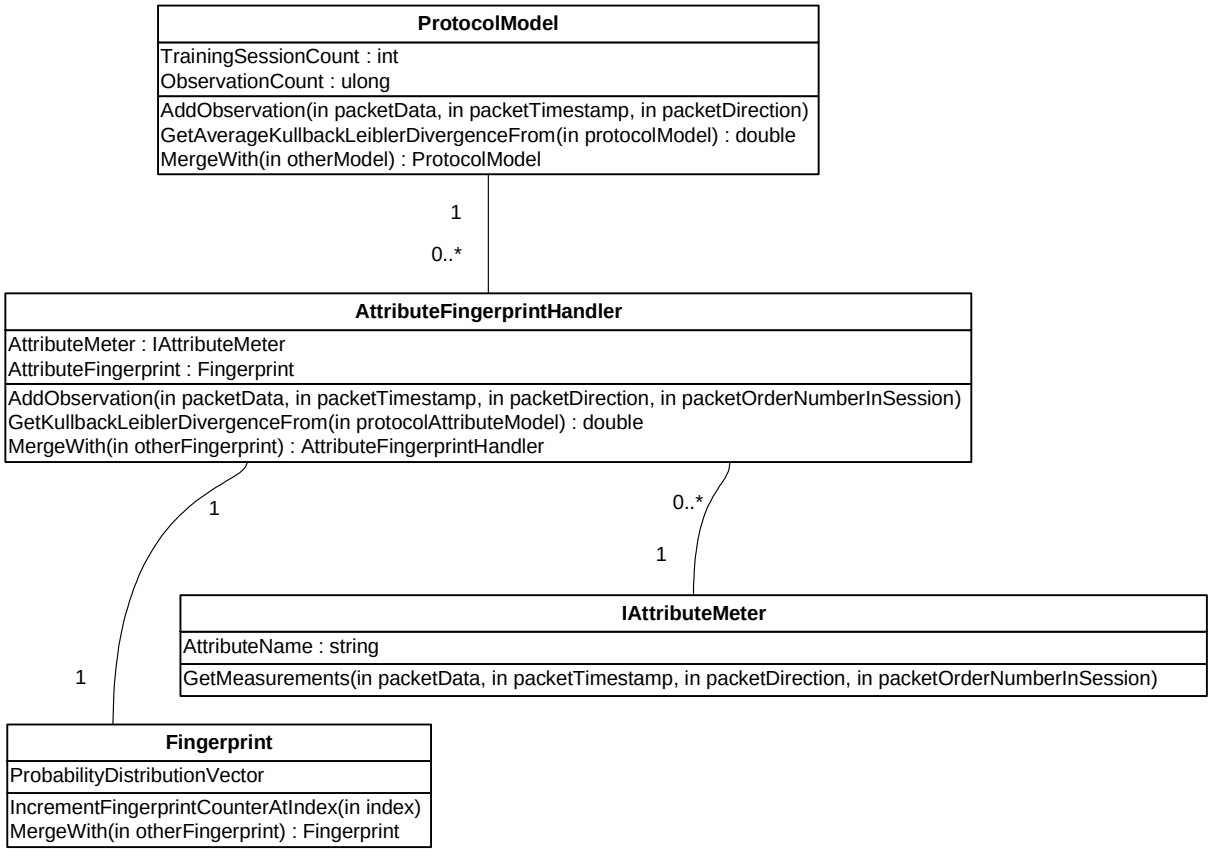

Figure 3 SPID algorithm UML class diagram

Every attribute meter holds a function called **`GetMeasurements`**, which takes the packet application data, packet timestamp (in order to calculate packet inter-arrival time), packet order number in session and packet direction (client to server or vice versa) as input and returns a set of integer values. The returned integer values represent the indices, in the associated fingerprint counter vector, that should be incremented.

```
int[] GetMeasurements(byte[] packetData, DateTime packetTimestamp,
PacketDirection packetDirection, int packetOrderNumberInSession){…}
```

A simple attribute meter can for example have a **`GetMeasurements`** function that returns a set of integers representing the individual byte values in the application layer of a packet. A simple HTTP GET request such as "**`GET / HTTP/1.1`**"[4], which is fed to such a metering

[4] The double carriage return line feed required to form a valid HTTP request are omitted to simplify the example

function, would for example generate the following return values: 71, 69, 84, 32, 47, 32, 72, 84, 84, 80, 47, 49, 46 and 49[5].

Utilising information about byte positions (offsets) in the application layer data can generate very good classification attributes for identifying protocols. One such offset-and-value-aware attribute metering solution is to have the **`GetMeasurements`** function return a set of integers that represent the offsets and values (1 or 0) of individual bits in the application layer data. The following example code shows how such an attribute metering function, which looks at the first 128 bits (16 bytes) of the application data, could be implemented:

```
int[] GetMeasurements(byte[] packetData, DateTime packetTimestamp,
PacketDirection packetDirection, int packetOrderNumberInSession){
  BitArray applicationDataBits=new BitArray(packetData);
  List<int> measurements=new List<int>();
  for(int i=0; i<128; i++){
    if(applicationDataBits[i]==false)
      measurements.Add(2*i);
    else
      measurements.Add(2*i+1);
  }
  return measurements.ToArray();
}
```

The example code provided above is somewhat similar to the "Discrete byte encoding" described by Haffner et al. in "ACAS: Automated Construction of Application Signatures", with the exception of bytes being replaced by bits and vice versa.

Another simple attribute measurement method is to have the **`GetMeasurements`** function return a value based on the packet size, i.e. small packet sizes will generate low values and large packet sizes will generate high values.

Attributes metering functions can also combine several properties into a composite attribute meters. Examples of composite attributes meters are:

- packet direction + packet order number in session + packet size
- packet direction + packet order number in session + packet inter-arrival time
- packet direction + byte-value frequency
- packet direction + packet order number in session + byte offset-value data
- packet order number in session + byte offset-value data
- packet order number in session + byte-value frequency + packet size

More than 20 other innovative, and sometimes rather complex, attribute metering functions will be made available as part of the open source proof-of-concept code for the SPID algorithm.

4.3 Generating Protocol Models for Observed Sessions

The defined attribute measurement functions can be used in order to generate fingerprints for individual network packets. The SPID algorithm is however designed to identify protocols used in network sessions, which is why a set of attribute fingerprints (one fingerprint per attribute meter) shall be assigned to each session rather than to every packet. Every fingerprint and its related attribute meter are paired together in an object called "attribute fingerprint handler". The attribute fingerprint handler does also provide several important support func-

[5] ASCII representations: 71=0x47='G', 69=0x45='E', 84=0x54='T' etc.

tions needed to, for example, compare fingerprints. Each attribute fingerprint handler is associated with a network session. The set of attribute fingerprint handlers assigned to the same session are contained within an object called "protocol model".

The protocol model object is created upon the establishment of a session (after the three-way handshake in the case of a TCP session). The fingerprints belonging to a protocol model are empty upon creation, which means that the fingerprints' counter vectors all have the value zero.

Every packet with payload (application layer data) is called an observation. Each such received observation shall be fed to the current session's protocol model object by using the `AddObservation` function. Upon receiving an observation the protocol model calls the `AddObservation` function of each attribute fingerprint handler. Each attribute fingerprint handlers do in turn call the `GetMeasurements` function of their associated attribute meter, which returns a set of vector indices, and then increment the fingerprint counters at these indices. Hence, when going back to the previous example with the HTTP GET command and the simple packet byte attribute meter, the counters would increment from all zeroes to:

- 3 for the counter at index 84 (since there are three T's in "`GET / HTTP/ 1.1`")
- 2 for counters at index 32, 47 and 49 (space, '/' and '1')
- 1 for counters at index 71, 69, 72, 80 and 46
- 0 for all other counters

All other attribute fingerprint handlers, belonging to the same protocol model, will also increase their counters based on the sets of indices that are returned from their respective attribute meters' `GetMeasurements` functions.

Subsequent packets in the same session will trigger the attribute fingerprint handlers of the session's protocol model to get more attribute measurements, which will cause the fingerprints' counter vector values to increment even more. A good approach is, however, to limit the attribute metering functions to only return attribute measurements for the first number of packets in a session. Doing so will decrease the time complexity of the algorithm as well as allow for better precision when the protocol needs to be identified early in a session.

4.4 Generating Protocol Models for Known Protocols

Protocol models for known protocols are generated from real network traffic, preferably stored in the "pcap" packet capture dump format. The pcap files do, however, need to be pre-classified, either manually or automatically, in order to be used as training data for the SPID algorithm.

The pre-classified training data is converted to protocol model objects (one per protocol) by generating protocol models for each session and merging the fingerprints of the same protocol and attribute type. The protocol model merger is, for every protocol, performed by adding together the counter vector values of fingerprints of the same attribute measurement type. The addition shall be performed pair wise for each vector index as shown in the following example code:

```
for(int i=0; i<FINGERPRINT_LENGTH; i++)
  mergedCounterData[i] = counterDataA[i] + counterDataB[i];
```

The more sessions that are merged together for each protocol, the more reliable the fingerprint will be. A rule of thumb for how many sessions are needed, to reliably fingerprint a protocol, is to have as many training sessions as the length of the vectors in a fingerprint (i.e. 256 for the examples provided in this report). Early evaluations of the SPID algorithm have, however, showed that reliable results can be achieved with as few training sessions as 10% of the fingerprint vector length.

4.5 Comparing Fingerprints

One of the key components of the SPID algorithm is how the fingerprints of an observed session are compared to fingerprints of know protocols. The previously described probability vectors are used as input data for the comparison function. The counter vector values of the observed session are first used in order to update the probability vectors, in case there has been a change of the counter vector values since the previous fingerprint comparison.

A probability vector ($P_{\text{attribute}}$), for a specific attribute, of an observed session is compared to a probability vector ($Q_{\text{attribute,protocol}}$) of a protocol model by using the Kullback-Leibler divergence (also known as the *relative entropy*) measurement.

$$\text{KL-Divergence}(P_{\text{attribute}} \,||\, Q_{\text{attribute,protocol}}) = \sum_i (P_{\text{attribute}}(i) * \log(P_{attribute}(i) \,/\, Q_{attribute,protocol}(i)))$$

The result of the KL divergence measure is a value that represents how much extra information, per attribute measurement, that is needed to describe the values in P (the observation probability vector) by using a code (such as a Huffman coding) that is optimised for Q (the model probability vector) instead of using a code optimised for P itself.

The best protocol model match for an observed observation P is the model which yields the smallest KL divergence for the observed session, i.e. the protocol model which most effectively (in terms of entropy) can be used to encode data with the distribution of P.

An alternative to using KL divergence is to use the cross entropy of P and Q. The cross entropy will always yield the same best protocol match as the KL divergence since the cross entropy for P and Q is equal to the Shannon entropy of P plus KL divergence of P and Q.

$$H(P,Q) = H(P) + \text{KL-Divergence}\ (P||Q)$$

One drawback of the cross entropy is that high-entropy sessions (in terms of both flow- and application data behaviour) will generate higher cross entropies than the cross entropies of low-entropy sessions. It can therefore be hard to determine if the best protocol match is good enough or if it is just a false positive. This problem is eliminated when using KL divergence since each divergence measurement is normalised with the observed session's own entropy. Hence, a global threshold value can be assigned when using KL divergence so that only protocols with divergences below the threshold can be accepted as probable protocol matches.

4.6 Comparing Protocol Models

Protocol models of observed sessions are compared to protocol models of known protocols by calculating the Kullback-Leibler divergences of the models' attribute fingerprints. The best protocol match is the one with the smallest average Kullback-Leibler divergence of the underlying attribute fingerprints. A good approach is to assign a threshold value, where Kullback-

Leibler divergence average values under the threshold are considered matches and larger divergences are not.

5 Proof-of-Concept Application

The proof-of-concept application for the SPID algorithm is written in C# using the Microsoft .NET framework. The application is designed to load two types of files; protocol model database files (in XML format) and single TCP-session capture files (in pcap format). The application automatically attempts to identify the application layer protocol when a TCP-session capture file is loaded.

The proof-of-concept application makes use of 27 different attribute meters in order to generate fingerprints. Many of these attribute meters are somewhat overlapping, so there is some potential to being able to reduce the number of attribute meters in future implementations.

5.1 Application Functionalities

A protocol model database XML-file can be loaded into the application by selecting "Import Protocol Model Database" from the File menu. After doing so a table of protocols will appear in the list view on the right of the user interface. The protocol models list also displays the number of sessions and observations that have been used, in form of pre-classified training data, to generate the fingerprints of each protocol model.

Upon loading a pcap file with a TCP session[6] the application starts creating a protocol model based on the first 100 packets in the session. The protocol model's attribute fingerprints are then compared to those of the known protocols in the database. The average Kullback-Leibler divergence, between the observed session's fingerprints and those of the known protocol models, is displayed in the divergence column of the "Session Protocol Identification" list view at the left. The protocol that best matches the observed session will automatically be selected in the drop-down list below the session protocol identification list view as displayed in the screenshot below:

[6] The loaded pcap should contain only one TCP session, preferably created through the "Follow TCP stream" functionality in Wireshark and saved with the "Displayed" packet range.

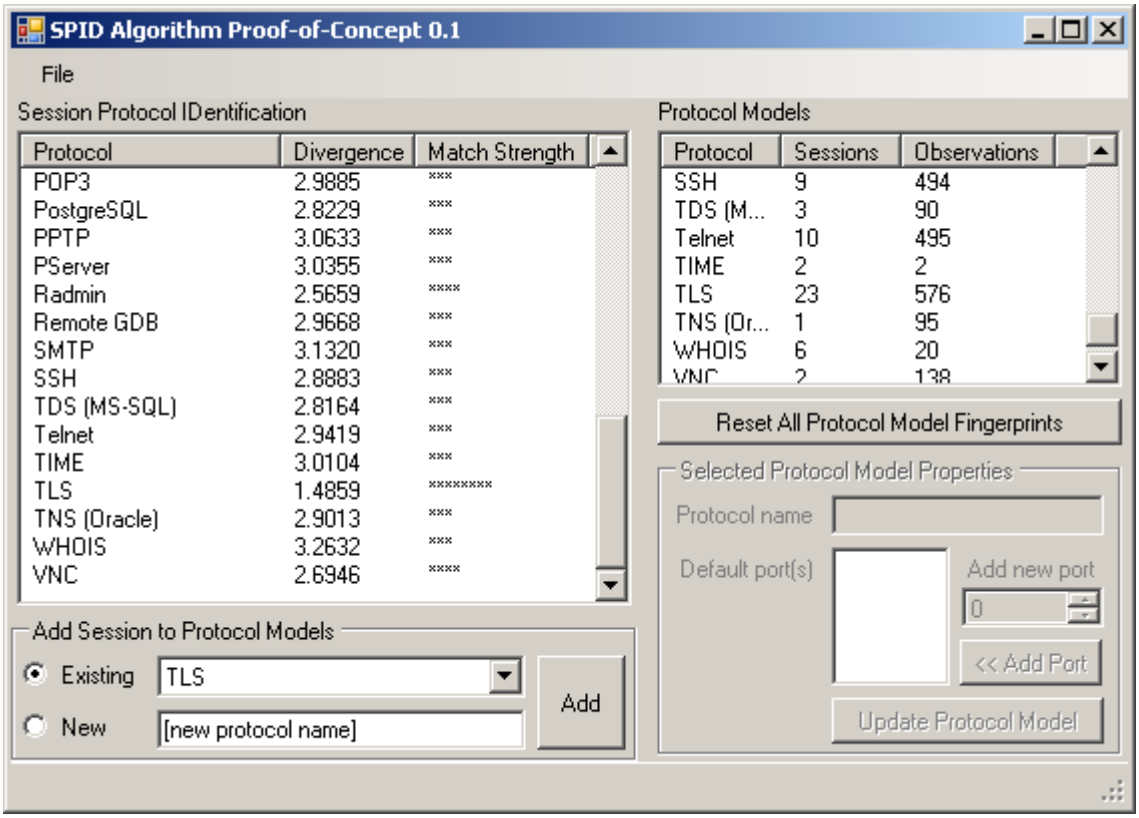

Figure 4 SPID Algorithm proof-of-concept application attempting to identify a TLS session

The observed session's protocol model can be appended to an existing protocol model by selecting the correct protocol in the drop-down list and pressing the "Add" button. A new protocol model can also be added to the protocol model database by manually assigning a new protocol name to the observed session. The application can also be used in order to generate a new protocol model database from scratch by adding new protocols to an empty protocol model database.

Different protocol model databases can also be merged into a combined protocol model database simply by importing several protocol model databases (one at a time) into the application. The merged database can then be saved to disk by choosing "Save Protocol Model Database" from the file menu.

Metadata about protocols, such as their default port numbers, can be added to the protocol model database. The port number information is not, under any circumstances, used in order to perform the protocol identification. The purpose of adding metadata about default port numbers is merely in order to allow other applications to use the database to, for example, alert upon detection of protocols leeching on well known ports of other protocols (such as P2P protocols running on TCP 80 or 443).

5.2 Protocol Model Database

The SPID proof-of-concept application will be published along with a protocol model database, which can be used to identify a limited set of protocols. The included database will only contain protocol models generated out of publicly available pcap files from sources such as Wireshark Sample Captures, OpenPacket.org and Lauras Lab Kit.

Users of the proof-of-concept application can, however, extend the included protocol model database with more traffic and more training data in order to support more protocols and attain more reliable protocol identifications.

5.3 Under the Hood

The Kullback-Leibler divergence function called `GetKullbackLeiblerDivergenceFrom()` in the AttributeFingerprintHandler class holds a small fix for the original KL-function, in order to introduce some evenly distributed noise and to avoid division by zero. This fix ensures that the KL-divergence function treats all probability vectors, in the protocol model database, as if their corresponding counter vectors were incremented by one at all indices. The probability vectors of observed sessions are treated as if their counter vector values were incremented by 1/VectorLength (i.e. 1/256).

The pcap file parser and the protocol dissectors for layer 2 to layer 4 are based on code from the open source application NetworkMiner.

A more detailed description of the inner workings of the "SPID Algorithm Proof-of-Concept" can be attained by looking at the source code, which is freely available from SourceForge at: http://sourceforge.net/projects/spid/

The SourceForge page for the SPID algorithm also holds a binary executable application, built for Microsoft Windows. The Microsoft .NET framework 2.0 does, however, need to be installed on the computer in order to properly run the SPID algorithm proof-of-concept application.

6 Evaluation

This report contains a limited amount of evaluation since there is not yet enough training and validation data available to assess the robustness of the provided protocol identification functionality.

6.1 Protocol Model Database Memory Complexity

The data rich SPID fingerprint format yields distinctly larger protocol signatures compared to, for example, regular expression based signatures. The size of a SPID protocol model database depends on three variables; number of protocols (P), number of used attribute meters[7] (A) and the length of the vectors (L). The memory complexity for the database is thereby $O(P*A*L)$. Both A and L are, however, considered being constants (with recommended values of: A=8, L=256), so the memory complexity is in fact $O(P)$. There is thereby a linear relationship between the memory complexity and the number of protocols in the database. Hence, an application that only needs a small number of protocols (in order to for example identify peer-to-peer protocols) can in a real-case scenario have a database that requires less than 100kB memory. A more comprehensive database of 100 protocols (which is about the number of protocol signatures available in Cisco's Network-Based Application Recognition as well as the L7-filter application) would require a couple of megabytes worth of memory. Using a low-entropy representation, such as XML, yields an overhead of more than twice the data size, so an XML-based protocol database file for 100 protocols would require approximately 10MB of storage.

[7] There will be one fingerprint for every possible protocol-attribute meter combination

Requiring 10MB of storage space is rather much, considering the limited amount of space available on the flash memories used by many embedded devices, but it is within the range of what can be considered acceptable. The possibility of compressing the database does, of course, also exist. A 10MB protocol database in an XML format can be expected to consume no more than 1MB of storage space after being compressed.

6.2 Operational Time and Memory Complexity

Let M be the total session fingerprints in memory and N the average number of analysed packets with payload (a.k.a. observations) in a session. The number of protocol models in the database is here considered to be constant, just as well as the number of protocol attributes used and the fingerprints' vector lengths. The packet size is also assumed to be constant, since the maximum size of transmitted packets (MTU) is around 1500 bytes on most networks[8].

Capturing packets and creating protocol model fingerprints for each session generates a memory footprint of $O(M)$, where M is the number of sessions needed in memory. The session fingerprints can, depending on the purpose of the application, be released from memory as soon as the protocol has been successfully identified.

The time complexity for capturing packets and updating a session model's fingerprints will be $O(\log M)$, to find the correct session model, for each received packet with payload. This yields a time complexity per session of $O(N \log M)$.

The time complexity for identifying the protocol used in a session by using a protocol fingerprint database is $O(P)$, where P is the number of protocols in the database. This yields a total time complexity of $O(N * (P + \log M))$ per session, if an effort to identify the used protocol is performed every time a packet with payload is received.

7 Future Work

Even though the SPID algorithm looks very promising it has yet to be benchmarked against other similar implementations to assess its potential.

7.1 Algorithm Tuning

The SPID algorithm needs to be tuned, in order to adjust parameters such as:

- Vector lengths
- Kullback-Leibler divergence threshold value
- Number of attribute meters used

One other very important tuning aspect is to choose the best combination of attribute meters, which provides the most robust protocol identification service. The best combination of attribute metering functions may vary depending on the requirements of a specific application; early identification might for example be prioritised to actively block illicit traffic, while this functionality is not very important when performing post-event analysis. My recommendation is, however, that all implementations should use the same combination of attribute meters. Doing so will make it easy to share attribute fingerprints for known protocols with others in the spirit of the open source community. I am therefore planning to do evaluations to find a good combination of attribute meters, which jointly can be used to reliably identify most protocols without requiring an immense protocol model database. I am hoping

[8] Ethernet has an MTU of 1500, IEEE 802.3 has 1492 and IEEE 802.11 uses 2272

to be able to identify a set of around eight strong and orthogonal attribute meters to replace the current set of 27 attribute meters.

7.2 Assembling Training Data

One of the keys to performing robust protocol identification is to have a good and accurate database of protocol fingerprints. I have unfortunately only a very limited set of training data, due to the lack of publicly available full-content packet capture files and because of the difficulties involved with getting permission to capture full-content traffic dumps from public networks. An important next step is therefore to collect sufficient training data, either as packet capture files or in the form of attribute measurement fingerprints, from several different sources. Any support that can be provided in this area is warmly welcomed by me!

The current plan for this project is therefore to get in contact with as many interested parties as possible, in order to accumulate a database with protocol models with enough natural variation. These parties can include, but might not be limited to, academic research institutions, network infrastructure developers and private individuals.

It is also important to get a good mix of training data from various types of networks, since both packet inter-arrival times and packet sizes might vary depending on the network's bandwidth and latency as well as the data link type.

7.3 Implementation

I am also planning to implement the SPID algorithm as a part of the NetworkMiner application, which is an open source Network Forensic Analysis Tool (NFAT) that I have developed. This way NetworkMiner will no longer have to rely on port numbers in order to select the correct application layer protocol parser for a network session.

One of my long-term goals with the SPID algorithm is to have the SPID algorithm implemented and used in network infrastructure devices and network security devices. This is also the field where I believe the SPID algorithm will be of most benefit.

8 References

Auld, Moore and Gull (2007), “Bayesian neural networks for Internet traffic classification”. IEEE Transactions on Neural Networks, 18 (1). ISSN 1045-9227

Bejtlich (2006), “Port Independent Protocol Identification”. TaoSecurity blog
http://taosecurity.blogspot.com/2006/09/port-independent-protocol.html

Bernaille, Teixeira and Salamatian (2006), “Early Application Identification”. Conference on Future Networking Technologies CONEXT 06, Lisbonne.

Cisco (2006), “Enabling Protocol Discovery”.
http://www.cisco.com/univercd/cc/td/doc/product/software/ios124/124tcg/tqos_c/part_05/qsnbar2.pdf

Cisco (2007), “Classifying Network Traffic Using NBAR”.
http://www.cisco.com/univercd/cc/td/doc/product/software/ios124/124tcg/tqos_c/part_05/qsnbar1.pdf

Crotti, Dusi, Este, Gringoli, Salgarelli (2007a), “Application Protocol Fingerprinting for Traffic Classification”. GTTI 2007.

Crotti, Dusi, Gringoli and Salgarelli (2007b), “Traffic Classification through Simple Statistical Fingerprinting”, ACM SIGCOMM Computer Communication Review, 37(1), January 2007.

Dhamankar and King (2007), “Protocol Identification via Statistical Analysis”. Black Hat USA 2007.
https://www.blackhat.com/presentations/bh-usa-07/Dhamankar_and_King/Presentation/bh-usa-07-dhamankar_and_king.pdf

Dreger, Feldmann, Mai, Paxson and Sommer (2006), “Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection”. Proceedings of USENIX Security Symposium. August 2006

Erman, Mahanti, Arlitt, Cohen, Williamson (2007), “Offline/Realtime Traffic Classification Using Semi- Supervised Learning”. SIGMETRICS 2007.

Haffner, Sen, Spatscheck, Wang (2005), ”ACAS: Automated Construction of Application Signatures”. Proceedings of the ACM SIGCOMM 2005.

John and Tafvelin (2008), “Heuristics to Classify Internet Backbone Traffic based on Connection Patterns”. IEEE ICOIN08

Juniper Networks, “IDP Application Identification Feature Demo”
http://www.juniper.net/products_and_services/intrusion_prevention_solutions/idp_application_video.html

Karagiannis, Papagiannaki and Faloutsos (2005), "BLINC: multilevel traffic classification in the dark". SIGCOMM 2005.

L7-filter, "Application Layer Packet Classifier for Linux"
http://l7-filter.sourceforge.net/

L7-filter, "L7-filter Pattern Writing HOWTO"
http://l7-filter.sourceforge.net/Pattern-HOWTO

Lee, C.S. (2008), "Basic Fl0p Signature Writing Guide".
http://www.rawpacket.org/anonymous/papers/Fl0p-Sigs-Writing.pdf

Li and Moore (2007), "A Machine Learning Approach for Efficient Traffic Classification" proc. of IEEE MASCOTS'07.

Ma, Levchenko, Kreibich, Savage and Voelker (2006), "Unexpected means of protocol inference". Internet Measurement Conference 2006

Madhukar and Williamson (2006), "A Longitudinal Study of P2P Traffic Classification". Proceedings of the 14th IEEE international Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'06)

NetScout (2008), "Sniffer Application Intelligence – Application performance analysis for networking professionals". NetScout Technology Application Note 2008-04-16.
http://www.netscout.com/docs/appnotes/NetScout_appnote_Application_Intelligence.pdf

De Montigny-Leboeuf (2005), "Flow Attributes For Use In Traffic Characterization". CRC Technical Note, CRC-TN-2005-003, December 2005.

Moore and Papagiannaki (2005), "Toward the accurate identification of network applications". Passive & Active Measurement Workshop 2005 (PAM2005)

NetworkMiner Wiki
http://networkminer.wiki.sourceforge.net/NetworkMiner

Protocolinfo, "EDonkey – Protocolinfo".
http://www.protocolinfo.org/wiki/EDonkey

SPID Algorithm Proof-of-Concept application open source project
http://sourceforge.net/projects/spid/

Zhang and Paxson (2000), "Detecting Backdoors". Proc. 9th USENIX Security Symposium, August 2000.