Federal University of Pernambuco

Graduation in Computer Science

Informatics Center
2009.1

ALYSSON FEITOZA SANTOS

AUTOMATIC SIGNATURE GENERATION

Diploma Thesis

Advisor: Djamel Sadok (jamel@cin.ufpe.br)
Co-advisor: Stênio Fernandes (stenio@gprt.ufpe.br)

Recife, PE
2009

Federal University of Pernambuco

Graduation in Computer Science

Informatics Center

2009.1

AUTOMATIC SIGNATURE GENERATION

Diploma Thesis

Diploma Thesis presented to the Informatics Center of the Federal University of Pernambuco by Alysson Feitoza Santos, advised by Prof. PhD. Djamel Sadok, as a requirement to obtain the Bachelor of Science degree in Computer Science.

Advisor: Djamel Sadok (jamel@cin.ufpe.br)
Co-advisor: Stênio Fernandes (stenio@gprt.ufpe.br)

Recife, PE
2009

APPROVAL SHEET

ALYSSON FEITOZA SANTOS

AUTOMATIC SIGNATURE GENERATION

Thesis approved on June 29th, 2009.

Thesis Committee

Prof. Djamel Fawzi Hadj Sadok, PhD – UFPE
(Advisor)

Prof. Nelson Souto Rosa, PhD – UFPE
(Examiner)

To the people that have always believed in me and that participate, directly or not, to my success: My family, friends, girlfriend, classmates, professors and work colleagues. These last 4 and half years were the best time of my life, till so.

"My foothold is tenoned and mortised in granite, I laugh at what you call dissolution, and I know the amplitude of time."

Walt Whitman

ACKNOWLEDGEMENTS

At first, I would like to remark that the following acknowledgments are not only for those who have helped me on this thesis, but for all that have been with me during my academic years. It is also important to remark that those acknowledgements are not in order of importance.

I'd like to thank my father Aloysio Santos and my mother Maria de Lourdes Santos, for all the support, love and dedication toward myself through all these years. Not less important than my parents, I also would like to thank my sister Daniely Santos who have always believed in me.

I cannot forget to sincerely thank my advisor Djamel Sadok, for the all support within this thesis and in GPRT. Not less important, I would like to thank Judith Kelner, for the previous year of support and acknowledgment at GPRT. I also want to thank my Project Manager and Co-Advisor Stênio Fernandes, who always clarified my ideas, giving me the necessary technical support for this thesis, besides motivating me to make a differentiated work.

To all those who have been with me on this undergraduate journey at the Informatics Center (even those that have quit the course but remained my friends), for all nights working (sometimes just chatting) together, discussions, parties, meetings and unforgettable times: Brá, Fanf and Shiru (longtime friends, since kindergarten times), Faga, Diru, Lui, Pit, Fifu, Borba, Andrezão, Kila, Bei, Biru, Paaty, Carol, André Paraíba, Bruno Apebão, Zu, Inó (Whose Father married my parents. Yes, he used to be a priest.), Denisson, Javael, Caio Peter, Mário (Mário who?), Tiago, Crazy João, João Paulo, Jera and Espanta.

I also would like to thank my girlfriend Helga (Namoradinha) for all the kindness and patience with me, besides all the support and love.

Additionally, I would like to thank all my colleagues at GPRT, for all friendship, advices and support provided: Arthur Callado (the oracle), Rafael Antonello, Ana Cristina, Thiago Xeroso (the man), Thiago Rocam, Leo, Manu, Nadia, Feitosão, Bruno, Ademir, Chico, Digão, Rover, Rodrigo Germano, Hachid, Kalil and Josias.

Please forgive me if I forgot someone.

ABSTRACT

The identification and classification of the traffic that is traversing a network link is a very important task performed by Internet Service Providers (ISP) administrators, in order to have a better comprehension of the applications being used by their users and to analyze how their network is prepared to provide a quality service to their customers. Among all the mechanisms used to classify network traffic, the Deep Packet Inspection (DPI) is the most used technique. However it is a mandatory requirement that the ISP DPI system must have a good set of application protocol signatures, in order to have an accurate traffic classification and an acceptable completeness level. Building this set of signatures is a very time consuming task and demands a high expertise, in order to make them very accurate and specific for each application.

This thesis presents a set of tools capable to ease the job of construction, test and characterization of signatures for DPI systems. Three tools were built for this purpose: (i) a signature generator, capable of identifying common substrings used by an application protocol and merging them into signatures; (ii) a pattern generator, which will help the user characterize all the aspects of the generated signature; (iii) and a traffic generator that is responsible for creating totally configurable per application class flows, whose packet payload will carry signatures that can be detected by DPI systems.

Keywords: Deep Packet Inspection, Signature Generation, Traffic Analysis, Computer Networks, Traffic Generation.

RESUMO

Identificar e classificar o tráfego que passa através de um enlace de rede é uma tarefa extremamente importante realizada pelos administradores de provedores de serviços de Internet, a fim de obter uma melhor compreensão acerca das aplicações mais frequentemente utilizadas pelos seus usuários e de analisar o quanto a sua rede está preparada para prover serviços de qualidade aos seus clientes. Dentre os mecanismos utilizados para classificar o trafégo de uma rede, a técnica de *Deep Packet Inspection* (DPI) é a mais utilizada. Entretanto é necessário que o sistema de DPI dos provedores de serviço tenha um bom conjunto de assinaturas para conseguir classificar o trafégo de forma precisa e com um nível aceitável de completude. Construir tais assinaturas é uma tarefa que exige bastante tempo e conhecimento especialista, a fim de torná-las extremamente precisas e específicas para cada aplicação.

Este trabalho apresenta um conjunto de ferramentas capazes de minimizar o trabalho de construir, testar e caracterizar assinaturas usadas por sistemas de DPI. Três ferramentas foram construídas com este propósito: (i) um gerador de assinaturas, capaz de identificar as cadeias de caracteres que são mais utilizadas pelos protocolos das aplicações e transformá-las em assinaturas; (ii) um gerador de padrões, o qual irá ajudar o usuário a listar todos os aspectos das assinaturas geradas; e, por fim, (iii) um gerador de tráfego responsável por gerar fluxos por classe de aplicação de forma totalmente configurável pelo usuário, os quais terão pacotes contendo, em sua carga útil, assinaturas que podem ser identificadas por sistemas de DPI.

Palavras-Chave: Análise de Tráfego, *Deep Packet Inspection*, Redes de Computadores, Geração de Tráfego, Geração de Assinaturas.

TABLE OF CONTENT

LIST OF FIGURES

List of Tables

1 INTRODUCTION

Internet Service Providers (ISP) and network administrators want to identify the kind of traffic that is passing through their backbone, in order to provide a better service to their customers, by offering them a high quality of service, and to plan and manage their infrastructure. Classifying network traffic according to its applications is a very important matter for a broad range of network areas, including traffic shaping, differentiated services, QoS improvement and billing. These are very important concerns to ISPs.

Many studies have been conducted in the traffic classification area, leading to the development of many classification methods. Some of them, like port based classification are no longer effective enough, due to dynamic port usage by some applications, especially P2P, and to traffic tunneling, for example, over HTTP. Other techniques used to classify network traffic are Machine Learning algorithms, application communication behavior analysis and payload based methods.

The Payload based methods are very effective and precise at traffic identification. They consist of inspecting the entire payload of a packet, seeking signatures and patterns which can point to a unique application. Once the payload matches a signature, we are able to classify the packet and unravel which application does it belongs to. This method is also known as Deep packet inspection (DPI) and is a very time consuming and a CPU-intensive activity, consuming over 90% of the processor power in the whole system [13], even when it is performed in off-line mode (on a previously captured trace file). Due to their high accuracy, DPI systems are broadly used both in the scientific community and by private companies. A very famous DPI system is an Intrusion Detection and Prevention System (IDS/IPS) called Snort [8] which can detect what applications are passing through the ISP backbone and also identify and block incoming attacks or unwanted operations, such as port scanning.

A DPI system must seek signatures within the payload of the packets to accurately classify a flow. Such approach is not new to the Internet worm research community [6] (e.g., intrusion detection systems); however, their focus is limited to identifying the security threatening traffic only. This work is focused on the application identification among innocuous traffic which widens the target traffic to be identified. It is also worth mentioning that the characteristics of worm and system invasion attempts signatures are very different from common user application signatures. Worm-like signatures can be very unspecific once the purpose is only to block a current attack, not to identify what kind of attack is passing on the wire. A single hexadecimal value can characterize a buffer overflow that can be used on an enormous variety of different attacks. Application traffic signatures must be unique and very specific, once the ISP managers want to know exactly what application it is, in order to either block, provide a differentiated service experience to the end user or to charge a different price for the application usage.

The use of regular expressions to describe application signatures is becoming very common in flow classification. A flow is identified by a 5-tuple: source IP address, source port, destination IP address, destination port, and transport protocol. This classification method can provide a very efficient and accurate matching mechanism. However, the regular expression matching process requires an enormous computational power, which is not scalable for online traffic identification. The way that the regular expression is constructed has a direct impact on the flow classification and on the total matching performance. Despite this fact, several DPI systems use regular expressions to represent application signatures. The already mentioned Snort intrusion detection/prevention system (IDS/IPS) has more than 1000 application signatures and offers the user the possibility to insert new regular expressions on demand.

In order to achieve a high precision on traffic identification with DPI systems, the signatures used on the matching process must be well constructed and very application specific, to avoid false positives, i.e., misclassifications. A long time is spent in the manual extraction of application patterns by a network specialist, which requires a preceding knowledge of the application protocol being studied or an empirical packet payload inspection for pattern recognition. Both activities are very time consuming and susceptible to failures which make the signature maintainability and update process a very difficult task. Sometimes this process may become even harder, by the fact that many protocols specifications are proprietary, i.e., not available to the community.

Automatically generating an application signature is a very important task to help ISP managers update and maintain their traffic analyzer system signature database, improving accuracy and performance. They can build signatures more efficiently without needing to manually inspect a captured packet trace file. This thesis will describe an application signature generator that can point out common patterns that appear frequently on a given set of pre-captured flows stored on a trace file. As will be outlined, the intention is not to generate signatures in real time, but offline, with a set of flows that belong specifically to the target application. This thesis is going to describe the system architecture and algorithm, the options that can be used to tune the tool, and the parameters that must be adjusted for each application to increase the signature relevance. It is worth mentioning that this tool only creates substrings that can uniquely identify an application. This work does not automatically generate regular expressions, but the substrings generated can ease the regular expression construction process, since the ISP manager will have a clue on which set of substrings he should pay more attention to when building a regular expression for a specific application.

Another contribution of this work is the creation of a pattern generator that will support the elaboration of histograms on three aspects that can help on the signature matching process: packet size distribution, the offset in which the signature is found on the packet payload, and in which packet of the flow the signature occurs. These three histograms can be created for each signature generated by our tool, following its format only. An extra feature of the pattern generator is the possibility to divide the histograms per transport layer protocol, once the signature can be carried out by UDP and/or TCP packets.

With all those information in addition to the signatures, a traffic analyzer can be used to classify flows by application in online mode. But what happens if the application to which the signatures are being created never appears in the ISP backbone at the present moment? There is dynamicity in the network applications and it may have considerable changes from one network to another. One application may be very popular in other WANs but not on the specifically WAN being analyzed. Even though not popular or having a remote probability to be present, some application signatures would be desired to make part of the signature database. Besides, the ISP manager may need to deploy his new generated DPI signatures on a totally controlled environment without affecting the rest of the network, and perform the matching process at wire speed, controlling the traffic generated by each application class, such as P2P file sharing, video streaming or web. How would an ISP administrator test online a new generated signature set?

In face of the high speed of current networks, the most viable direction in terms of cost to test modern LANs is by emulation: inject synthetic traffic, measuring their responses, and analyzing the results. It is clearly necessary to generate traffic that is very similar to the actual, in terms of either high speed and statistical properties.

To test signatures within a real environment, this diploma thesis proposes a traffic generator which will create real network packets possessing detectable payload information, according to an application packet size distribution, also having application signatures that respect the packet order that carries the signature on the flow and the signature offset within the packet payload. Such signature information regarding order and offset is provided by the automatic signature generation process. The traffic generator is a fully configurable tool that can generate traffic for a set of specified signatures, even if they belong to different applications. The user can inform the volume of traffic that should be created for each application class, and which fraction of the total volume each individual application will represent.

This tool can also be used in stress tests of network equipments, such as a router or switch, or can also be used to check if a firewall or IDS/IPS is working properly. The traffic generator must be able to generate traffic at wire speed, in order to measure how much efficiency a traffic analyzer may perform online, thus, the performance of the data generation and transference plays an important role on its construction.

The remainder of this thesis is organized as follows:

- Section 2: presents the underlying background needed for the thesis understanding, covering Regular Expressions, Deep Packet Inspection, packet transmission mechanism on Linux-based systems and Linux timing functions;
- Section 3: presents the related work that has been found in literature;
- Section 4: the proposed project for the thesis is presented in this section. The complete signature generation algorithm and its features will be outlined. It will also describe all the options and parameters that must be set to generate the signatures, as well as their impact on the generation

process. The pattern generator tool and a description of the traffic generator architecture is presented in this section either, explaining the initial parameters and configuration, as well as the methodology created to evaluate it;

- Section 5: this section shows the signatures generated for sample applications and the evaluation results obtained by the traffic and pattern generator tools;
- Section 6: this section opens up a space for discussion on some important points that were observed;
- Section 7: finally, the main concluding remarks of the thesis are presented in this section. Besides, some possible future works.

2 BACKGROUND

This Section contains a brief description of Deep Packet Inspection (DPI) systems, the format of the signatures that are generally used by DPI systems, Regular expressions, Linux socket programming API, packet transmission, and resolution and control of Linux timers, in order to provide the essential background needed for the remainder of this thesis.

2.1 Deep Packet Inspection

Deep Packet Inspection (DPI) systems, also called complete packet inspection and information extraction, focus on analyzing, indentifying and classifying the traffic that is traversing the network, by examining the content of the packet payload as it passes through probe point on the network. It can be used as a packet filter to drop frames which may carry viruses, spams, intrusion attempts, worms, or may either work as an application identifier to perform network traffic shaping. Those systems accomplish these tasks by comparing part of the packet data against strings, which represent signatures of the content to be scanned. DPI systems can classify data flows to make flow control shaping, or can scan every packet of a data link, demanding more computational power. Some DPI systems also enable advanced security functions, being very effective against buffer overflow attacks, denial of service (Dos) attacks, shell code insertion, SQL injection, and other sophisticated intrusions, working as a wire speed stateful firewall.

Besides the security concerns that an ISP manager must have, DPI systems provide a very accurate form of identifying applications used by the subscribers. The DPI approach is more effective than the simple port-application comparison, once this technique became very limited, due to protocol tunneling to overcome firewalls barriers. The DPI systems have a good classification accuracy. ISPs have been recently relying on such solutions. Throughout the global Internet, ISPs use DPI systems to perform different kinds of services, like:

1. Lawful Intercept: ISPs are commonly required by various governments around the world to perform some kind of lawful intercept when needed by law enforcement agencies;

2. Policy definition and enforcement: service providers obligated by the service level agreement with their customers may use DPI to detect copyright infringements, illegal materials, and unfair use of bandwidth. In some countries, these kind of inspection are only allowed with a government or court warranty. Policies can be defined in a way to allow or block connection to or from certain IP addresses, certain protocols or even heuristics that identify certain applications;
3. Content based filtering and charging: ISPs can block, filter or even charge customers that are using certain kind of applications (maybe undesirable ones) within their network. Examples include blocking Voice over IP (VoIP) and P2P applications;
4. Quality of Service (QoS): some applications like VoIP and Video over Demand (VoD) require low latency and high priority against common traffic, such as web browsing.

2.2 Regular Expressions

Regular Expressions (RegEx) is a set of strings that represent a pattern used to match a certain string of characters. They provide an enormous expressiveness without needing to express the desired patterns one by one. They are widely used in computer science, from compilers and programming languages to text editors. RegExes are commonly represented as state machines and can be implemented as Deterministic Finite Automata (DFA) state machines or as Non-Deterministic Finite Automata (NFA). These state machines can grow exponentially, demanding a large amount of memory and computational power during the matching process. DPI systems usually rely on RegEx to perform their inspection process. Examples of two RegEx signatures of P2P applications used to classify network flows can be found on table 2.1.

Table 2.1 - Regular Expression used to identify network applications.

Application	**Signature**
Emule	(^(\xe3|\xc5).(\x01|\x54|\x93|\xa4|\x4c|\x85|\x86|\x87|\x92|\x60|\x55|\x47|\x40)

BitTorrent	(\x13)BitTorrent Protocol(\x23)

2.3 Signature formats

A signature may be a portion of the packet payload data that is static and distinguishable for applications, which can be described as a sequence of ASCII characters or hex values. There exist many other types of signature formats, as will be presented later in this section. Due to the complexity of application protocols, the signatures that will represent them must be very generic and expressive in order to provide good classification accuracy. Signatures based on regular expressions are powerful, leading to accurate matches, although it suffers performance problems, as they can take a long time processing in software-like pattern matching solutions. The performance depends also on the expression length. Besides it is complex to be built in hardware, because they can be very difficult to construct, even though they are commonly used on DPI systems to identify normal applications or malware-like traffic. Newsome et al. has discussed three types of worm signatures [5]. They are conjunction signatures, token-subsequence signatures and Bayes signatures:

- Conjunction signatures: consists of an unordered set of tokens that identify data flows. A flow matches a conjunction signature if all tokes in the set are found in its payload in any order;
- Token-subsequence signatures: the signature is an ordered set of tokens. A flow matches a token subsequence signature if the flow contains the sequence of substrings in the signature with the same ordering. Signatures of this type can easily be expressed as regular expressions. Given the same set of tokens, a token subsequence signature will be more specific than a conjunction signature, because of the former ordering constraint;
- Bayes signatures: consists of a set of tokens, each of which is associated with a score, and an overall threshold. In contrast with the previous types, Bayes signatures provide a probabilistic matching. Given a flow, the probability that the flow belongs to the application is computed using the scores of the substrings present in the flow. If the resulting probability is over the threshold, the flow is classified as the application type.

To Identify application protocols, the signatures must be very specific, to identify a flow uniquely. Discover that a flow might carry a worm-like signature will be sufficient for a DPI system that is focused on system invasion prevention, without need to point out which worm it was. Application protocols may have similarities that will lead a classification system to behave inappropriately.

2.4 Packet Transmission Mechanism

This section contemplates an explanation on raw packet transmission, and details on how to transmit packets within a Linux-based system, which is important for a better understanding of the problems that are encountered in the construction of a typical traffic generator. Raw socket is a software interface through which an application can have full control of the packets that are being sent or received in the network. In the following section, the raw packet journey inside the Linux Kernel is detailed.

2.4.1. Sockets

Packet sockets are used to receive or send raw packets at the device driver, i.e. OSI Layer 2 level. They allow a programmer to implement protocol modules in user space on the top of the physical layer.

When a programmer opens a socket by invoking the `socket` system call for the packet transmission, the kernel internally calls `__sock_create`, which will create a new `sock` structure (the kernel representation of sockets), and register a handler for the specified protocol.

The socket function has the following interface: `socket(int family, int type, int protocol)`, where:

- `family`: refers to the protocol family that the user will handle within the socket, e.g. `PF_INET` for sockets based on IPv4 protocol, `PF_PACKET` for sockets which want to communicate directly with the link layer;
- `type`: is the type of the socket communication semantics, examples of common types are:
  - `SOCK_STREAM`: provides ordered, reliable, two-way, connection-based byte streams, commonly used with TCP connections;

    - `SOCK_DGRAM`: datagram support, commonly used by UDP connections. The user can define headers of OSI Layers 3 and 4;
    - `SOCK_RAW`: allow users to have access to lower-level communication protocols. This kind of socket can define the headers from OSI Layer 4 to the link level. Packets created with this type of sockets are passed to the device driver without any changes.
- `protocol`: The protocol of the packets, e.g. `ETH_P_IP` for IP, `ETH_P_ALL` for all types of packets.

Internally, these sockets are basically represented as a queue which stores pointers to packets inside the kernel and each of these queues is responsible for a class of packets.

The `PF_PACKET`, for instance, handles all the packets that comes in or will be go out of on the Network Interface Card (NIC) and skips all the processing that is usually performed by the TCP/IP protocol stack inside the kernel. It allows applications to access the network driver directly, so a programmer can implement network protocols in the user space.

The `PF_INET` socket only deals with IP packets, but there are other types of sockets that only deal with TCP, UDP, and other sort of packets, at the Linux protocol stack. Raw sockets are not compatible with PF_INET. Figure 1 illustrates the path of a packet into the Linux protocol stack. The `dev_queue_xmit` will be better explained in the next Section. On Linux systems, only super users are allowed to open socket connections.

The link level header is available at an independent physical layer address structure called sockaddr_ll. On raw packet transmissions, the programmer uses this structure to inform the physical layer addresses.

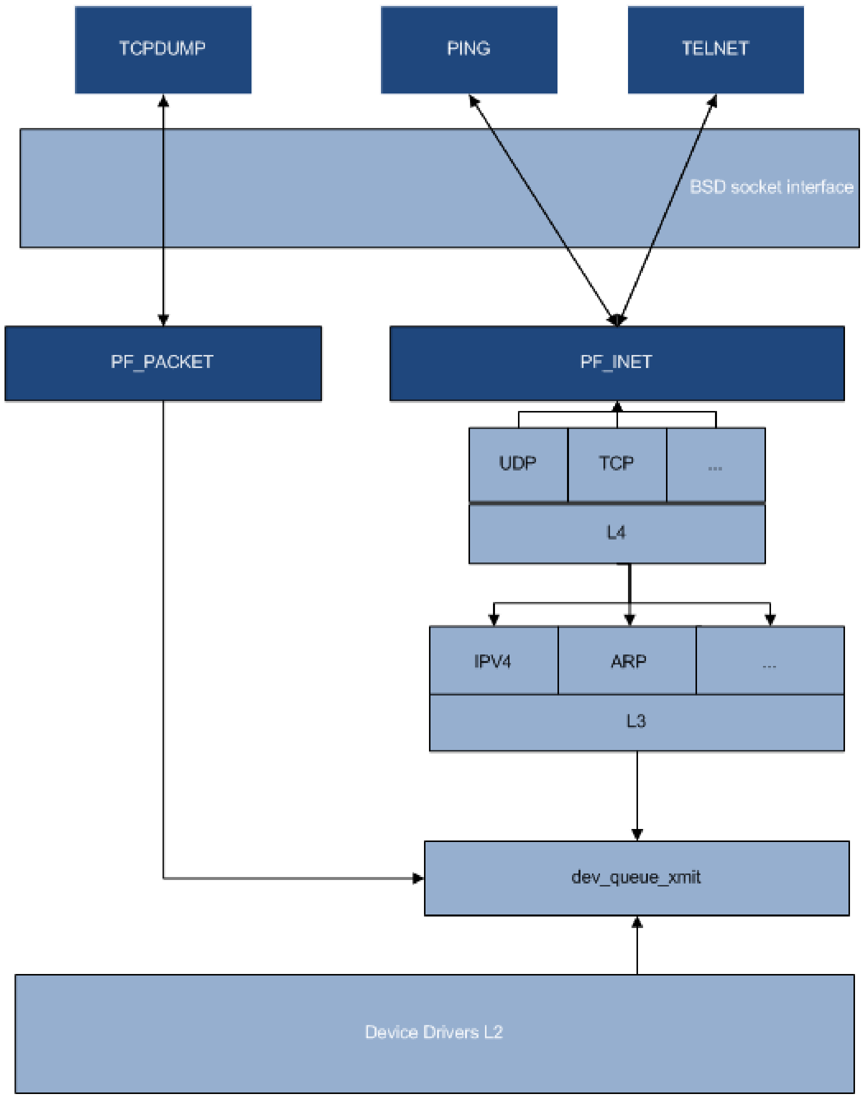


Figure 1 - Linux socket's path

2.5 Packet Transmission with PF_PACKET sockets

Whenever an application needs to send a raw packet through the network, within a PF_PACKET socket, it is done by invoking the write() or send() Linux system calls, which will makes Linux trap to the kernel and allocates a `sk_buff` (short for socket buffer) structure, which is the packet representation inside the kernel, and fetches the data from the user space into the new `sk_buff`. It then invokes `dev_queue_xmit`, passing the `sk_buff` as a parameter, which will queue a buffer for transmission to a network driver inside the kernel and will transfer frames from this kernel queue to the driver outgoing-frame buffer. When `dev_queue_xmit` is called, all the information required to the frame transmission, such as the outgoing device, the next hop, and its link layer address is ready. If a programmer is working with PF_PACKET sockets, his application must guarantee all these features. The `sk_buff` contains the outgoing device reference, a pointer to the beginning of the packet payload and its length. The main tasks of `dev_queue_xmit` are as follows: (i) checking whether the frame is composed of fragments and if the device can handle them through scatter/gather DMA operations, combining the fragments if the device is incapable of doing so; (ii) verify if the OSI layer 4 checksum is computed; (iii) and select which frame will be transmitted, i.e. copied to the network drive egress buffer. Regarding that some devices, like the loopback device, do not have a queue to store the packets, whenever a frame is passed to `dev_queue_xmit` it is delivered to the NIC buffer. The `dev_queue_xmit` function puts a pointer to a `sk_buff` inside the CPU queue for egress packets (there is one queue for each CPU), and then raises a software interrupt (`softirq`), and returns the congestion level of the queue to the caller. Please note that in the event that this queue is full, the new outcoming packets are discarded. The function that processes the `softirqs` is known as `do_softirq.` It checks a bit mask and calls the appropriate handling routine, which represents the set bit. For transmission, the interrupt of interest is the `NET_TX_SOFTIRQ`, which can be raised in two cases:

- When the transmission is enable on the device and it is assured that the frames waiting to be sent are actually sent when all the needed conditions are met, i.e., when the device has enough memory to handle it.
- When the device driver asks the `net_tx_action` softirqd to take care of it.

The `net_tx_action` handle routine, which will treat the IRQ raised, deallocate all the buffers that have been added at the device driver by dequeuing the first packet from the current CPU queue and will try to grab the lock of the device egress queue to transmit the frames. The handler `net_tx_action` schedules a device for transmission if it could not grab the lock of the device egress queue (making it unable to transmit), by calling the `netif_schedule` function, preparing it to send frames later. The `net_tx_action` handler routine is also called in two situations, which are when there are devices waiting to transmit and when it is time to do housekeeping onto the buffers that will not be used anymore by the kernel.

Almost all devices use a queue to schedule egress traffic and the kernel can use queuing disciplines to arrange the frames in a more efficient order for transmission. A detailed discussion about traffic control and its queuing disciplines is out of the scope.

2.6 Linux Timers

Countless computerized activities are driven by timing measurements. Schedule actions to be performed somewhere in the future is a necessity that every operating system has. A mechanism to deal with activities scheduled to run at some relatively precise time must be built. Any microprocessor that wishes to support an operating system must have a programmable interval timer that periodically interrupts the processor. This periodic interrupt is known as a system clock tick and it works as a system's regent, orchestrating its activities.

Linux has a very simple view of what time is; it measures time in clock ticks since the system booted. Timing measurements are performed by several hardware circuits based on fixed-frequency oscillators and counters. All system times are based on this measurement, which is known as jiffies after the globally available variable of the same name. It has two types of system timers, both of them are queue routines to be called at some system time but they are slightly different in their implementations. The Figure 2 shows both mechanisms.

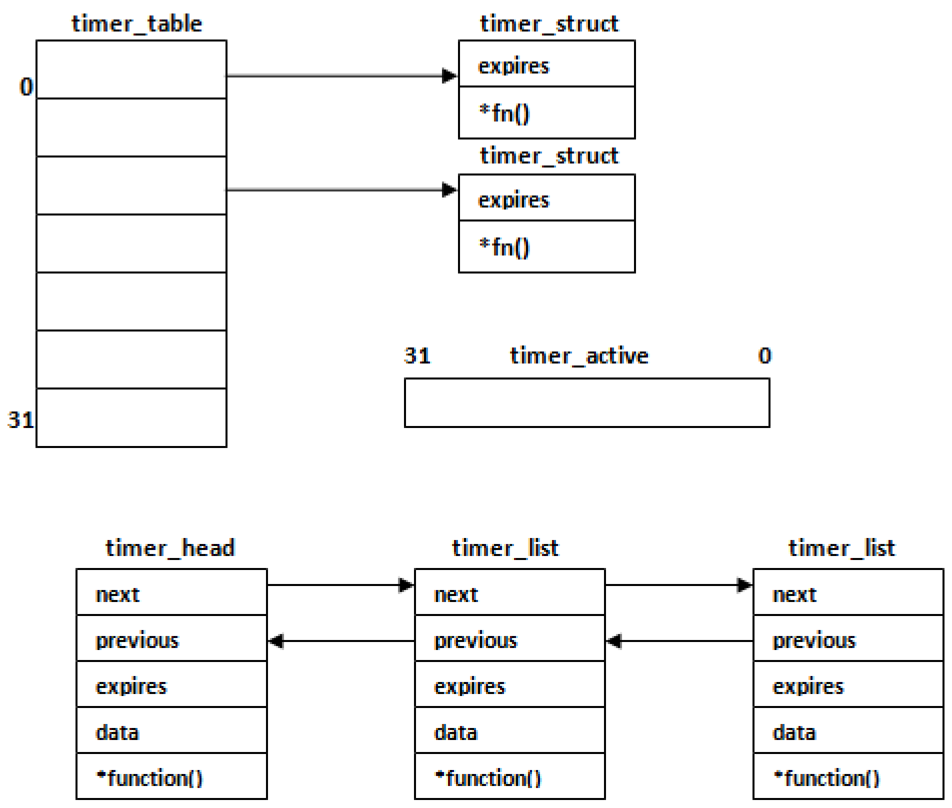


Figure 2 - Linux timers

The Linux used to have an old timer mechanism that had a static array of 32 pointers to timer_struct data structures and a mask of active timers, represented by timer_active.

Where the timers should go in the timer table is defined statically. Entries are added into this table mostly at system initialization time. The newer Linux mechanism uses a linked list of timer_list data structures held in ascending expiry time order.

Both methods use time in jiffies as an expiration time, so that a timer that attempt to run in five seconds would have to convert it to units of jiffies and add that to the current system time to get the system time in jiffies when the timer should expire. Every system clock ticks the timer bottom half handler and marks it as active, so that when the scheduler next runs, the timer queues get processed. The timer bottom half handler processes both types of system timer. For the old system timers the timer_active bit mask is check for bits that are set.

If the time for an active timer has expired, its timer routine is called and its active bit is cleared. For new system timers, the entries in the linked list of timer_list data structures are checked.

Every expired timer is removed from the list and its routine is called. The new timer mechanism has the advantage of being able to pass an argument to the timer routine.

2.7 `Sleep` and `Nanosleep` time functions

The `sleep` function makes a running thread or process became inactive for a certain period of time specified in seconds. When the elapsed time arrives, an interrupt is raised for the sleeping process. The `nanosleep` function has the same functionality as sleep, but it has a higher timer resolution: it receives a time struct as input expressed in nanoseconds.

The current implementation of `nanosleep` is based on the normal kernel timer mechanism, which has a resolution of 1/*HZ* s (i.e, 10 ms on Linux/i386 and 1 ms on Linux/Alpha). Therefore, `nanosleep` pauses for at least the specified time, however it can take up to 10 ms longer than specified until the process becomes runnable again.

As some applications require much more precise pauses (e.g., in order to control some time-critical hardware), `nanosleep` is also capable of short high-precision pauses. If the process is scheduled under a real-time policy like *SCHED_FIFO* (first-in, first-out) or *SCHED_RR* (round-robin), then pauses of up to 2 ms will be performed as busy waits with microsecond precision.

The `nanosleep` function shall return a value of 0 on success and -1 on failure or if interrupted. This latter case is different from `sleep` which returns 0 if the requested time has elapsed, or the number of seconds left to sleep.

3 RELATED WORK

Researchers have done a lot of work in classifying network traffic according to the applications. A naive internet application traffic identification technique such as port-based identifications does not guarantee accuracy and precision anymore due to the diversity of today's internet traffic. Besides, a lot of applications use tunnels, such as HTTP tunnels, and dynamic port allocation to circumvent port-based blocking by firewalls. In addition to the native port based method, there are some new methods including machine learning algorithms based on flow statistical information [2], application communication pattern based methods [9] and payload based methods [4].

The machine learning techniques classify network traffic using flows information and traffic characteristics, such as average size of packets, inter arrival time between packets, protocol usage, etc. McGregor et al. [10] used machine learning techniques for clustering flows, in order to achieve a high precision of the classification, but they found that some applications could not be well classified with this approach. ACAS [3] uses the first N bytes payload as the input to train a machine learning model and uses it to classify flows. However, using the first N bytes payload directly may introduce too much noise during the training phase, leading to poor classification accuracy.

Mingjiang Ye et al. [9] introduced a behavioral method to classify P2P traffic, by studying the characteristics of P2P applications. They recognize a P2P flow by checking pieces of data being downloaded and uploaded. If a flow has the same data block on an upload and on a download flow, inside a parameterized time interval, the associated flows are classified as P2P. Their approach proved to be very effective on a wide range of P2P applications, such as on many BitTorrent [14] clients.

Moore et al. used application signatures to classify network traffic in [4]. They used the signatures in two ways: searching them in single packets and searching them in the first 1KB payload of flows. They have pointed that matching signatures in first 1KB payload is enough for a good classification, which could help to generate signatures more quickly and with less memory usage. They have not discussed how to generate signatures. Sen *et al*. have generated some P2P signatures manually by analyzing their application-layer protocol [11] through available documentation and packet-level traces. P2P applications was mostly focused because of its high traffic usage and complexity, besides their proprietary protocols, which difficult their deployment. Sen et al. signatures were very precise and well constructed, but were somehow limited. They created signatures that could be found only in the beginning of the packets, in order to accelerate and ease the classification phase, which can be very efficient it terms of matching speed but can lead to a huge amount of unclassified traffic. They also paid more attention to the P2P download flows, instead of the signaling ones. The reason was that signaling flows correspond to less traffic volume, than download flows. The amount of time spent on the generation phase was not minimal and must be remade if new protocol versions appear in a near future. Their methods used the common strings appearing on the applications flows to generate signatures.

Automatic signature generation tools have been studied recently in worm detection area [5][6][7]. A worm is a self-replicate program: it remotely exploits software vulnerabilities on a victim host, in a way that the victim becomes infected, and begins to remotely infect other victims.

The EarlyBird worm signature generator system [7], generates overlapping fixed-length content block over each byte offset for each sample flow, and extracts the content blocks appearing many time as a whole signature. By sifting through network traffic for content strings that are both frequently repeated and widely dispersed, they could automatically identify new worms and their precise signature. Their signature generation was made online on a single network sensor. They stated that an optimal algorithm could work as follows: For each packet passing through the network, the content is extracted and all substrings are processed and indexed into a prevalence table that is incremented every time the substring is found. This table implements a histogram of all observed substrings. This table keeps two lists, containing IP addresses that are searched and updated each time a substring counter is incremented. Sorting this table on the substring count and the size of the address lists will produce the set of likely worm traffic. Substrings that are not widely dispersed are discarded, leaving only the worm-like content. To scale to high-speed links, the EarlyBird algorithm uses a different approach from the optimal one proposed, but in essence it works quite similar.

The Autograph system [6] generates worm-like signatures by selecting TCP suspicious flows (a flow that might carry some threat) from an online monitor and sending them to a distribute machine, which will store that flows on disk. Autograph reassembles the payload of a given flow, making it become a single contiguous block of data. Them it divides the payload of the flows into variable-length substrings using a Content-based Payload Partitioning schema based on Rabin fingerprints. The number of generated substrings is reduced, although short common substrings can be missed using the partition schema. After the payload partition phase, Autograph measures the frequency, with which a substring occurs across all suspicious flow’s payload, and proposes the most frequent ones to be signatures candidates. Autograph incorporates a blacklisting technique into signature generation. An administrator may configure Autograph with a blacklist of disallowed signatures, in an effort to prevent the system from generating signatures that will cause false positives. The blacklist is simply a set of strings. Any signature Autograph selects that is a substring of an entry in the blacklist is discarded.

Besides the worm signature generators, mentioned previously, there are some works aiming at application signature generation. Byung-Chul et al. [2] have studied the application signature generation problem. They propose the LASER algorithm which tries to find the longest common subsequence among samples without any prior knowledge of the application protocol. The longest common subsequence extracted from sample flows is chose to be the application signature. To improve the system performance, the LASER algorithm only considers the flow’s first N packets to generate the signatures. It also groups the packets by its size, since large packets are not likely to carry the same kind of information as the small ones. The algorithm compares two samples to get the longest common subsequence between them, and then compares it with other samples iteratively to refine it. Their paper showed that it is a challenge to generate signatures when there is a lot of noise on the samples, for example, common substrings that repeatedly appear on the sample flows, but do not belong to the application protocol. It is worth mentioning that they generated signatures in form of substrings, instead of regular expressions.

Mingjiang Ye et al. [1] built a tool called AutoSig, to generate application signature automatically, by combining the techniques used on the Autograph and on EarlyBird systems. Their algorithm consists on dividing the payload of a set of flows in short substrings called shingles, extract the ones that constantly appear in a certain amount of flows and merge them if they are neighbors or overlap. As the LASER algorithm, they also generated signatures in form of substrings, not as regular expressions. Even though their signatures led to a very good flow classification, they needed to calibrate some commonality threshold parameters used by the algorithm manually, in order to achieve a perfect signature, which reduces the automation of the process.

Many traffic generators have been proposed on the literature, but none have focused on generating synthetic packets with detectable payload. Traffic generators are mainly implemented as user-space applications, like RUDE [17] and MGEN [18].

RUDE is able to instantiate a number of simultaneous patterns of traffic, but provides a non-extensible script language. RUDE’s architecture does not provide any explicit support for extensible interfaces and is not suitable to work at high rates. RUDE also includes a traffic collector called CRUDE, which can be used to receive and log the traffic created.

MGEN provides both a command line and a Graphical User Interface (GUI) for traffic generation on user-space. It supports different UNIX-based OS’s, but its accuracy is limited because of the system timers used (as presented on Section 2, on Linux kernels the normal timer resolution is only 10ms).

KUTE [16] (formerly known as UDPGen) is an UDP traffic generator designed to achieve high performance over Gigabit-Ethernet. It is a Linux 2.6 kernel module that operates directly on the network device driver, avoiding moving data from user space to kernel space, which causes a high delay on packet transmission. The user can configure the source and destination IP addresses, source and destination ports, packet rate, packet length, duration of the flow, packet’s payload, Time to Live (TTL), Type of Service (ToS), and whether UDP checksums and IP identification field should be used.

The Internet Traffic Generator (ITG) [19] focuses on reproducing TCP and UDP traffic and replicating samples of stochastic processes for inter-departure time and packet size. It is comparable in performance with RUDE and MGEN, although it provides more traffic patterns. The Browny and Robust Traffic Engine (BRUTE [20]) is a PC-based traffic generator which takes advantages of the capabilities of Linux

Kernel 2.4 – 2.6, to generate traffic at high rates. In a gigabit Ethernet scenario, on a commodity machine running Linux, the highest throughput achieved with BRUTE was 1.09 Mfps (Mega Frame (packets) per seconds). BRUTE was designed as a user space application, aiming to obtain a high flexibility at the expense of a slight increase of latency. It is easily extensible through optimized functions for implementing additional traffic sources.

There are many others traffic generators and traffic simulators, which are much more complex once they implement real traffic behaviors, on the open source community available for free downloads on the internet, ranging from very efficient generators to more simple ones. In addition, there are many others traffic generators implemented as hardware platforms, like the very efficient Agilent [25] traffic generators. However none has a focus on generating traffic for DPI systems testing, with configurable traffic profile on the application level, transmitting packets with detectable and real payload. The traffic generator proposed by this thesis is very specific for this goal, although it can be used for network nodes stress tests, and is totally implemented for commodity platforms, requiring only a reliable NIC to achieve a good performance.

4 PROJECT DESCRIPTION

The present thesis aims to build three tools that together ease the work of an ISP administrator or DPI manufacturer, to construct signatures, for traffic application identification and profiling, and to test the generated signatures online. It is not on the scope of these tools to help on worm-like or other kinds of malware signatures construction. The traffic generator can be used as a network stresser, to test the capabilities of a switch or router, although it does not provide support for that. It focuses solely on generating controlled and detectable traffic from one point to another on the same local area network.

During the next sections, this thesis will describe how the signature, pattern and traffic generator work. All the tools run on Ubuntu 8.10 machines, with kernel 2.6.

4.1 The Signature Generator (SigGen)

This Section aims at explaining how the signature generator tool (SigGen) works. It also includes all the improvements proposed by this work, in order to generate more accurate signatures and to decrease the amount of false positives signatures. A simple command-lined user interface is also explained to simplify the tool usage.

4.1.1. Signature Definition

First of all it is important to mention that the SigGen will generate signatures, for application protocols, in format of substrings, as LASER [2] and AutoSig [1]. Like the systems, mentioned previously, SigGen will not create regular expressions. Its work is to point the most relevant substring found on the given set of sampled flows, in order of relevance, combining substrings if they appear on the same set of flows. The generated substrings will represent uniquely an application and can be used for traffic classification. The signatures are generated as a sequence of hexadecimal values, if it is not printable, or as sequence of ASCII values, if it is printable, just as the samples signatures extracted from [1] showed in table 1.

Table 4.1 - Sample AutoSig Signatures. \0x20 means a blank space in hex

Application	Signature
FTP	[220\0x20];[220\0x20][FTP\0x20server\0x20]; [221\0x20Goodbye];
HTTP	[HTTP/1.]; [GET\0x20/][HTTP/1.];

As presented on Table 4.1, the signatures may be a single string, like "[HTTP/1.]" or it can be the combination of two strings, like the FTP signature "[220\0x20][FTP\0x20server\0x20]". If the signature is a combination of strings, means that it is necessary to find all those strings on the same flow. Notice that these substrings can be directly transformed into a regular expression, although it may not be efficient. If a specialist wants to write a regular expression with these signatures, a manual packet trace analysis is still required, but at least he will be aware of the most important substring to consider.

4.1.2. The Generation Algorithm

SigGen is mostly based on the algorithm proposed by Mingjiang Ye et al. [1] with some improvements to generate the signatures and to reduce the number of false positives. The AutoSig has 3 distinct phases, namely the substring generation, aggregation and signature construction. Figure 3 illustrates the AutoSig signature generation process.

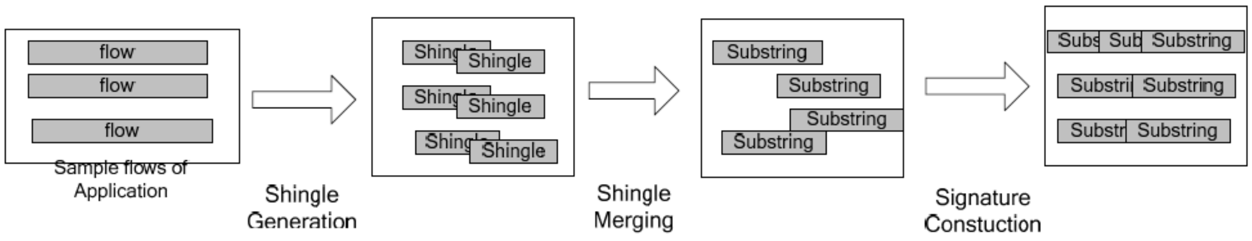


Figure 3 - AutoSig signature generation phases

SigGen follows the same AutoSig three phases, doing an extra verification on the end of the second phase to exclude signatures previously marked by the blacklist. As the AutoGraph system, SigGen allows the user to specify strings that he already knows that are not specific enough for that application, like an “HTTP/1.1” pattern. This feature is called blacklisting a substring, and will be further explained on this thesis. Another extra feature that SigGen includes is the possibility to split the flows into control and data flows; in order to decrease the noise caused by packets with different purpose, mixed up on the same flows set. The input of the algorithm is a set of sample flows which belongs uniquely to a certain known application and gives as output signatures capable of identifying such application. It is out of the scope of this thesis to explain how the sample flows were captured. It is worth mentioning that a process sniffer was used to capture only the traffic that was generated by a certain application, dumping it to a file. All the flows were captured on a machine connected to the GPRT (Network and Telecommunication Research Group) local area network, which is part of the UFPE (Federal University of Pernambuco) network, which is connected to the Brazilian RNP (National Research Network).

The sample flows are organized on a hash table, where they are going to be stored. Even though hash tables offers a better access time on elements searching, most of the signature generation time will be spent on sequential iterations through all flows. The reason why a hash table was chose is to ease the flow’s payload mount phase. A hash function is calculated for each packet of the trace file and the payload of the first 10 packets is mounted sequentially and stored on a correspondent flow structure. Besides Moore et al. [4], other studies proved that by analyzing only the first packets of a flow it is enough to classified such flow [12]. If the sample flows are a very large set, the algorithm may require a large amount of memory, since during all phases the data (from the first 10 packets) of the sampled flows will be kept on memory. Tables showing the memory required by different sized traces will be presented on Section 5. If the user wants to divide the sample flow into control and data, he must inform a packet size limit. It is recommended that the user runs the PatternGen (which will be explained later on this thesis) to see graphically the packet size distribution of the application packets. Data flows tend to be larger than control ones. SigGen divides the flows according to the packet size informed; mounting the large sized ones separately from the little sized and then processes all phases for both data and control flows, generating different substrings.

The first phase on the generation process, called shingles generation, consists of dividing the payload of each flow into small and equal lengthened pieces of data, called by Mingjiang Ye et al. shingles. A common substring can produce a lot of common shingles. For example, ‘BitTorrent’ is a common substring in BitTorrent protocol. When shingle size is 5, there are 6 common shingles from the common substring, which are: ‘BitTo’, ‘itTor’, ‘tTorr’, ‘Torre’, ‘orren’ and ‘rrent’. A flow payload data division into shingles is represented by Figure 4.

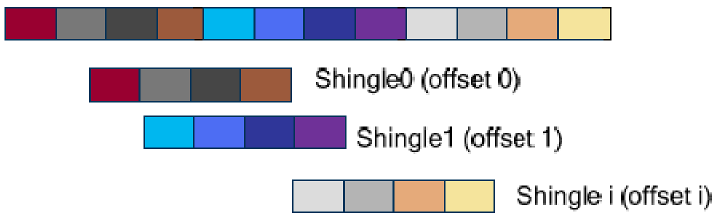


Figure 4 - Shingles Division

The data pieces that appear frequently in more than a certain amount of flows will be extracted and kept on a separate common shingles table that maps each shingle with a set of unique flows containing it. Figure 5 represents this common shingles table.

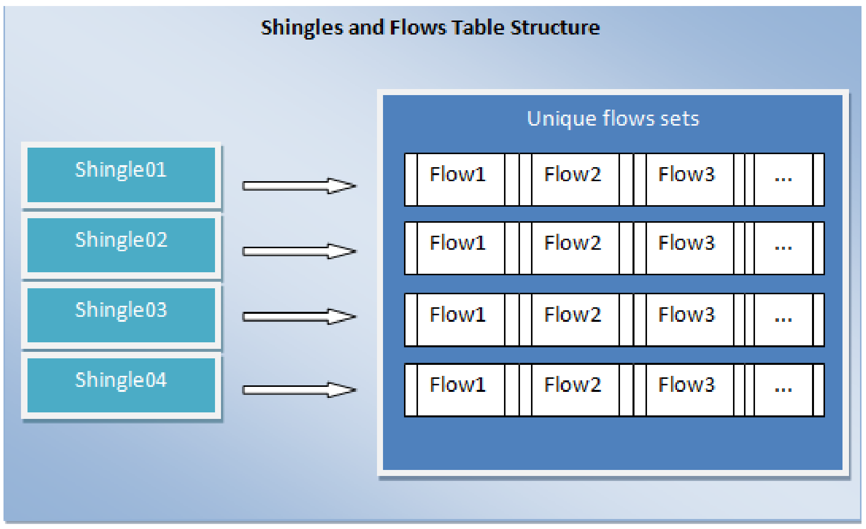


Figure 5 - Shingles and unique flows table

One can notice that the reason why flows were chosen is because the signatures will be used on flows identification, not on packets identification. Extracting the shingles that appears commonly on a certain amount of packets will lead to poor traffic classification, once it may happen that a certain shingle appear more than once on a single flow, but do not appear constantly on the rest of the samples. As stated by Mingjiang Ye et al., application signatures usually tend to be smaller than worm like signature. To circumvent that, small shingle sized are extracted from the flows, being the length four the most adequate, as showed by Mingjiang Ye et al. For precision reasons and to ease the comparisons and implementation, the ‘00’ hex decimals characters were extracted from the flows payload. There also exist some regular expressions matching libraries that do not recognize ‘00’ hex values.

After extracting common shingles and storing them on a separate table that stores the unique flows in which the shingles occur, SigGen needs to merge the encountered shingles to transform them in larger substrings. The substring aggregation phase consists of merging the most common pieces of data found, if they are overlapped or adjacent, and test if these combined substrings appear in a certain parameterized amount of flows. SigGen tries to merge each shingle from the common shingle table without concerning to order. The process is done sequentially. If it reaches the end, and a merging has occurred, it rewinds to the beginning and continues, till no merge happen. SigGen implements the adaptive merge algorithm proposed by Mingjiang Ye et al. Before explicit the adaptive algorithm, let’s discuss about the most simple substring merge algorithm, the greedy merging.

This algorithm consists in simply merging two substrings if they are overlapped or adjacent, and then this merged substring can be further merged with others substrings. The algorithm does not take into account the relevance of this combined substring on the sample flows set. For example: while generate substrings for a famous Chinese IP2PTV application called tvants, a generator can separate two common shingles to merge, “TVANTS” and “ SHARE”. If it checks on the data stored from the sample flows, it will discover that there are many occurrences of the substring “TVANTS SHARE”, proving that this two shingles are adjacent. But the string “TVANTS” by itself, appears in much more flows than the combined one. So the greedy algorithm would produce a substring that recognizes fewer flows.

The adaptive merging algorithm focus on generating substrings that will classify as many unique flows as possible. Using the most common shingle table as input the merge score for shingle X and shingle Y is given by the formula illustrated by Figure 6.

$$sim(x,y) = \frac{|flows(xy)|}{|flows(x) \cup flows(y)|}$$

Figure 6 - Calculating shingles merging score

Flows(x) represents all the unique flows mapped by the most common shingle table, flows(xy) means all the unique flows containing the combination of shingle x and y (if they overlaps the combination will be a proper merging of them, if not, it is assumed that they are neighbors and a simple concatenation is made) and flows(x) U flows(y) means all the unique flows that contains both shingle X or shingle Y. To merge two strings SigGen compares the value returned by the formula with a parameterized value S, ranging from 0.0 to 1.0, which can be given by SigGen as an input by the user. If the user does not inform the S, 0.8 is the default value used. If the calculated value returned by the formula is equal or greater than S, the two shingles are merged. Figure 7 extracted from the Mingjiang Ye et al. work, illustrates an example of its use. Regard that AutoSig uses 0.9 as the default S value.

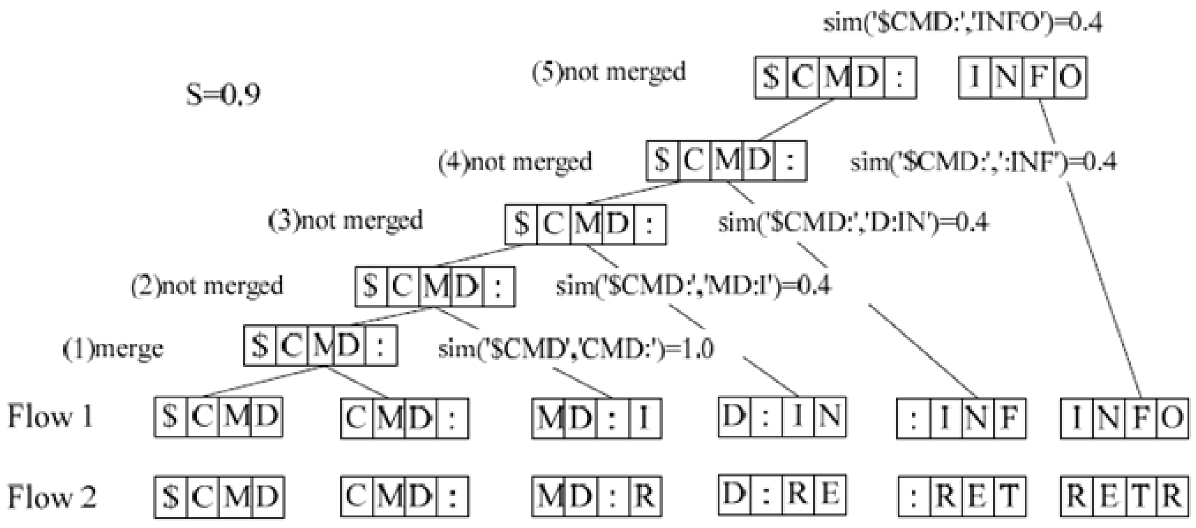


Figure 7 - Adaptive merging algorithm

After trying to merge all the shingles, SigGen, checks the resulting substrings with a blacklist configured by the user. The blacklist consists solely by strings that shall not be considered if pointed by the substring merging phase. These blacklisted strings tend to be ones that do not identify uniquely the application protocol begin analyzed. SigGen simply discard the string that are equal to a blacklisted one or contains a blacklisted as a substring. In this same step, SigGen also discard substrings that are contained on others, being or not on the blacklist.

On the signature construction phase, the substrings are arranged on a sibling tree like structure (Figure 8), in which a parent has a list of children, all of them brothers, in order to combine two substrings on the same signature.

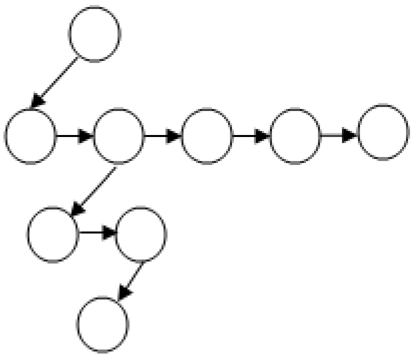
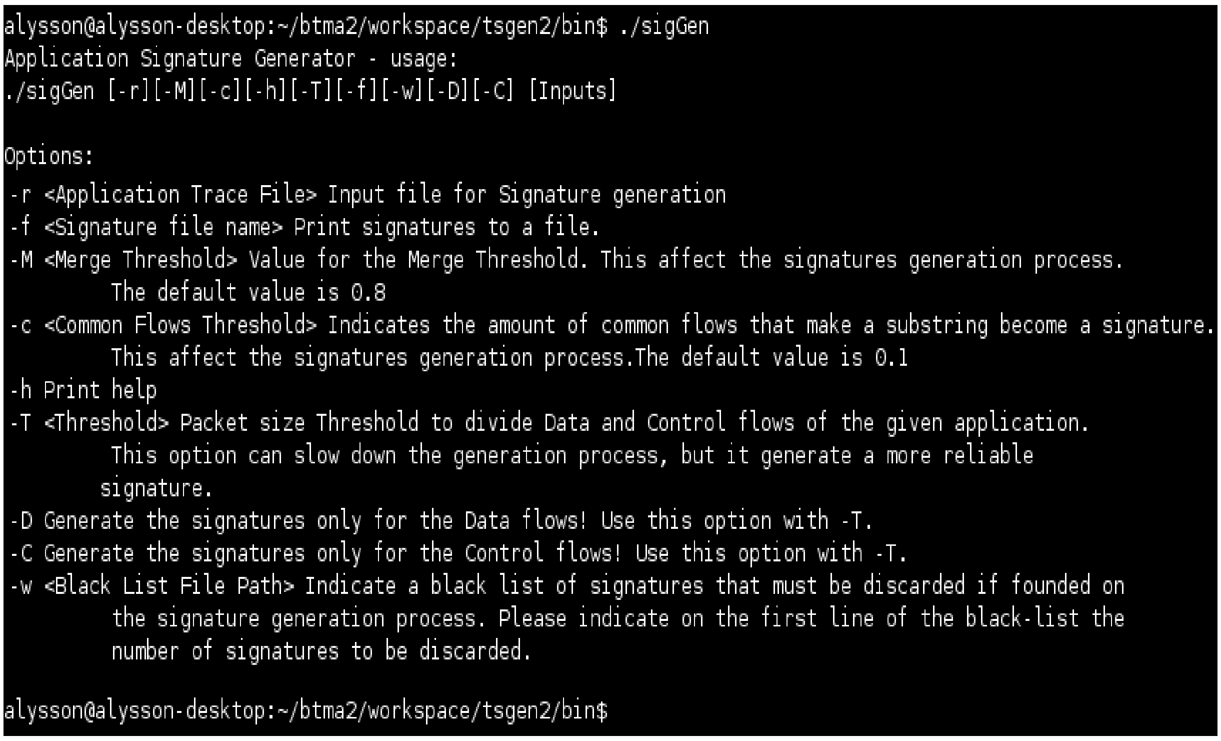


Figure 8 - Sibling Tree

Each node of the tree represents a substring and the root node is the empty string. If the flow set containing a substring Y is a subset of the flow set of another substring X, the node representing Y will become a children of the X node's, if not they will become brothers. It means that the set of flows of X contains not only the substring X but some of them also contain the substring Y. After constructing the tree, a path from the root to any leaf will become a signature. The main difference against the AutoSig signature construction phase is that they do not use sibling trees. As mentioned before, the tool cannot generate signatures totally autonomously. After the generation phase, a human is needed to manually refine the generated signatures by changing the parameters values to create more refined signatures. Figure 9 shows the command line help of the SigGen tool. Options –M and –c are the adaptive merge score and the percentage of flows that a given shingle most appears to be added on the common shingles table respectively. Their default values are 0.8 and 0.1. The user also has the possibility to generate a signature file that can be used by both the PatternGen and the TrafficGen tools.



```
alysson@alysson-desktop:~/btma2/workspace/tsgen2/bin$ ./sigGen
Application Signature Generator - usage:
./sigGen [-r][-M][-c][-h][-T][-f][-w][-D][-C] [Inputs]

Options:
 -r <Application Trace File> Input file for Signature generation
 -f <Signature file name> Print signatures to a file.
 -M <Merge Threshold> Value for the Merge Threshold. This affect the signatures generation process.
         The default value is 0.8
 -c <Common Flows Threshold> Indicates the amount of common flows that make a substring become a signature.
         This affect the signatures generation process.The default value is 0.1
 -h Print help
 -T <Threshold> Packet size Threshold to divide Data and Control flows of the given application.
         This option can slow down the generation process, but it generate a more reliable
        signature.
 -D Generate the signatures only for the Data flows! Use this option with -T.
 -C Generate the signatures only for the Control flows! Use this option with -T.
 -w <Black List File Path> Indicate a black list of signatures that must be discarded if founded on
         the signature generation process. Please indicate on the first line of the black-list the
         number of signatures to be discarded.

alysson@alysson-desktop:~/btma2/workspace/tsgen2/bin$
```

Figure 9 - SigGen command line help

4.2 The Pattern Generator (PatternGen)

This Section aims on describe the PatternGen tool, which will generate a series of histograms that may help the use of the SigGen tool and to construct the traffic generator (TrafficGen) configuration file.

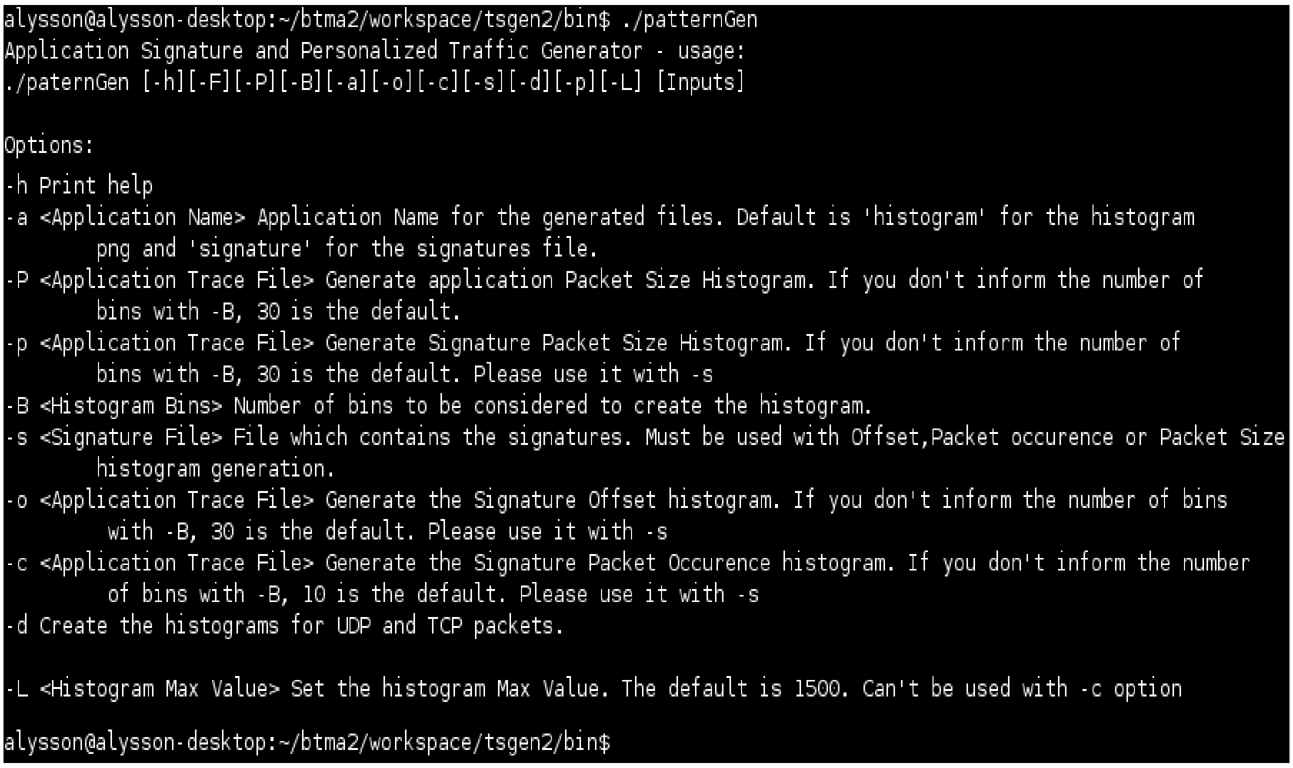
3.2.1 The Patterns Generated

The histograms generated by the PatternGen tool are listed below:

- Application packet size histogram: This histogram represents the packet size distribution of a given application.
- Signature packet size histogram: Instead of create a unique distribution packet size histogram for the entire application, it can generate a distribution packet size for each signature. The signatures can be informed as an input file and can be generated by the SigGen tool or can be handmade.
- Signature offset histogram: PatternGen can generate a histogram containing the offsets of the payloads that carry a given signature. This feature might be useful while building regular expressions.

- Signature Flow Packet Occurrence: This feature generates a histogram that represents in which packet of a certain flow a given signature appears. For example, it can tell that the signature “BitTorrent” usually appears with more frequency on the 5th and 6th packet of the sample flows.

The number of bins and the upper limit on the X axis of the histogram can be informed by the user. It is also possible to generate different histograms for UDP and TCP packets separately. Figure 10 illustrates the patternGen command line help.



```
alysson@alysson-desktop:~/btma2/workspace/tsgen2/bin$ ./patternGen
Application Signature and Personalized Traffic Generator - usage:
./paternGen [-h][-F][-P][-B][-a][-o][-c][-s][-d][-p][-L] [Inputs]

Options:
-h Print help
-a <Application Name> Application Name for the generated files. Default is 'histogram' for the histogram
        png and 'signature' for the signatures file.
-P <Application Trace File> Generate application Packet Size Histogram. If you don't inform the number of
        bins with -B, 30 is the default.
-p <Application Trace File> Generate Signature Packet Size Histogram. If you don't inform the number of
        bins with -B, 30 is the default. Please use it with -s
-B <Histogram Bins> Number of bins to be considered to create the histogram.
-s <Signature File> File which contains the signatures. Must be used with Offset,Packet occurence or Packet Size
        histogram generation.
-o <Application Trace File> Generate the Signature Offset histogram. If you don't inform the number of bins
         with -B, 30 is the default. Please use it with -s
-c <Application Trace File> Generate the Signature Packet Occurence histogram. If you don't inform the number
         of bins with -B, 10 is the default. Please use it with -s
-d Create the histograms for UDP and TCP packets.

-L <Histogram Max Value> Set the histogram Max Value. The default is 1500. Can't be used with -c option

alysson@alysson-desktop:~/btma2/workspace/tsgen2/bin$
```

Figure 10 - PatterGen command line help

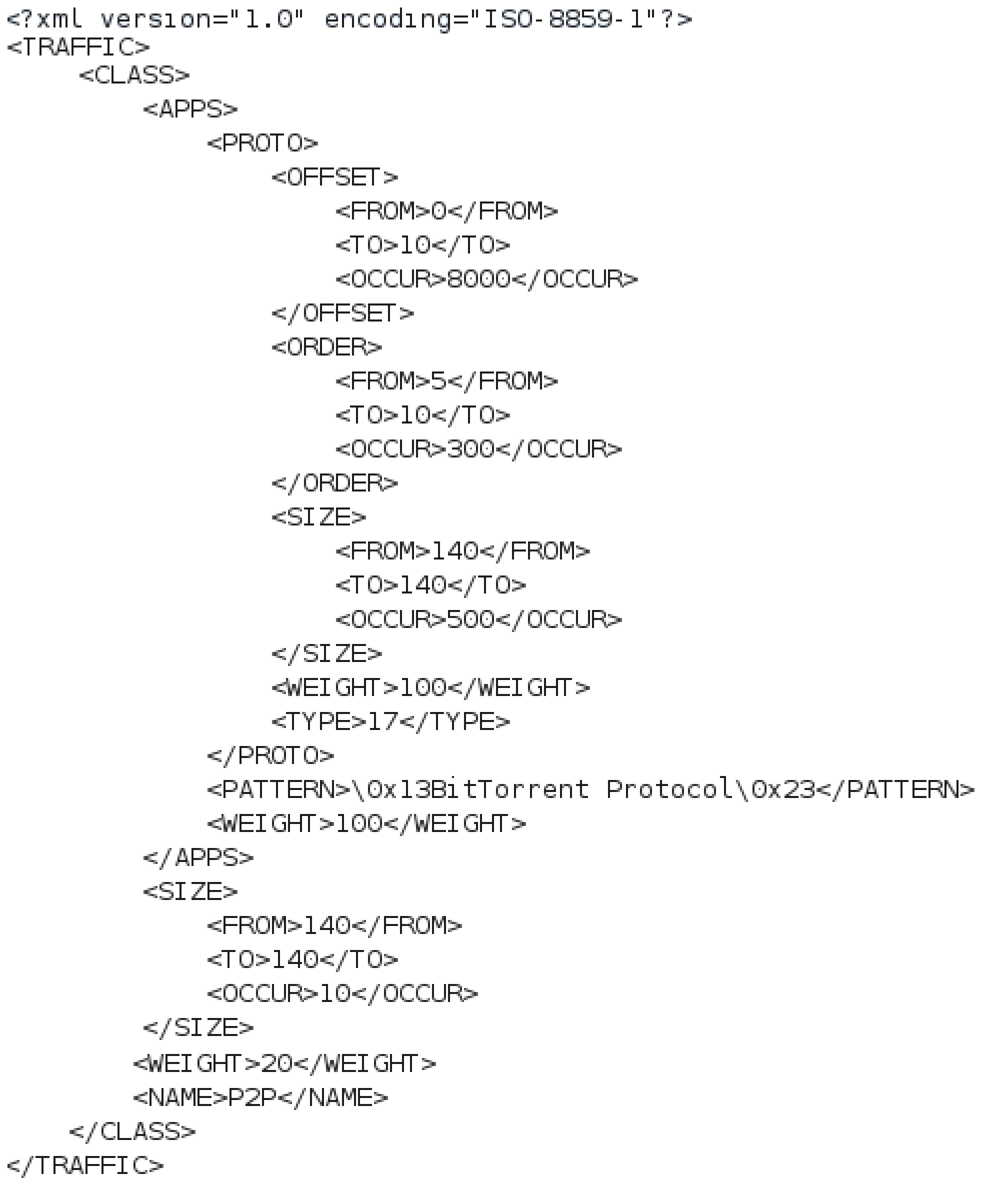
4.3 The Traffic Generator (TrafficGen)

After creating the application signatures with SigGen and include then on his DPI system, an ISP administrator may want to verify if they are working properly. If his DPI works online, the administrator can use the TrafficGen to create detectable traffic between two nodes on a Local Area Network. TrafficGen generates flows following a configuration file, in which the user can set the amount of traffic for a certain application class, configuring the traffic profile generated. For example, the traffic can be 80% constituted by P2P file sharing applications and 20% of normal Web flows. Each application class can have different percentages of per-application traffic. Performance and memory required to run the TrafficGen will be shown on

Section 5. It is not the focus of the tool to be a real traffic simulator, like the discrete event network simulator ns-2 [23] environment. The goal of TrafficGen is only to produce configurable and detectable application flows for DPI systems classification tests.

3.3.1 The Configuration File

In order to create a totally controlled traffic, following a packet size distribution, a payload format, and a traffic application profile, the user must configure a XML that will be parsed and interpreted by TrafficGen. Figure 11 brings an example of a valid XML accepted by TrafficGen.



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<TRAFFIC>
     <CLASS>
          <APPS>
               <PROTO>
                    <OFFSET>
                         <FROM>0</FROM>
                         <TO>10</TO>
                         <OCCUR>8000</OCCUR>
                    </OFFSET>
                    <ORDER>
                         <FROM>5</FROM>
                         <TO>10</TO>
                         <OCCUR>300</OCCUR>
                    </ORDER>
                    <SIZE>
                         <FROM>140</FROM>
                         <TO>140</TO>
                         <OCCUR>500</OCCUR>
                    </SIZE>
                    <WEIGHT>100</WEIGHT>
                    <TYPE>17</TYPE>
               </PROTO>
               <PATTERN>\0x13BitTorrent Protocol\0x23</PATTERN>
               <WEIGHT>100</WEIGHT>
          </APPS>
          <SIZE>
               <FROM>140</FROM>
               <TO>140</TO>
               <OCCUR>10</OCCUR>
          </SIZE>
         <WEIGHT>20</WEIGHT>
         <NAME>P2P</NAME>
     </CLASS>
</TRAFFIC>
```

Figure 11 - Xml configuration file

The tag “CLASS” specify each application class that will be considered on the traffic generation. Each XML can have as many classes as the user wants. Section 5 has some comparisons between the number of application classes and the TrafficGen performance. The “WEIGHT” tag inside “CLASS” represents the percentage of the traffic that the class will have and the tag “NAME” only show its name, for indexing purposes. A class can have as many “SIZE” tag as it needs. Each “SIZE” tag represents a bin of the IP packet size histogram distribution that this application class follows. “SIZE” tag will be further explained later. Every class contains one or more “APPS” tags, which describes a certain application that belongs to that class. Figure 11 has the description of a P2P class that will be responsible for 20% of the generated traffic, containing only one application; in this case it is the file sharing application BitTorrent.

The “APPS” tag will contain only one “PATTERN” tag that will describe the application’s signature to be placed on the generated packets’ payload. The pattern can be the same signature generated by SigGen tool. If the user wants to add more than one signature per application, he must be aware of the amount of traffic that each signature will represent on the total belonging to that application. The “WEIGHT” tag represents the contribution that this application will has on the total traffic of the application class. The “PROTO” tag means the UDP or the TCP transport Layer protocols that the application might use. A single “APPS” tag can only have two “PROTO” tags, TCP and UDP. Each “PROTO” may have as many “ORDER”, “OFFSET” and “SIZE” tags as the user wants. The “ORDER” tag describes each bin of the flow packet order histogram in which the signature is found. “SIZE” represents the same as it did inside the “CLASS” tag, but instead of representing the entire application class packet size distribution, it describes only the distribution of a certain application that uses “PROTO” as its transport protocol. If the “SIZE” inside a “PROTO” tag is greater than the “SIZE” of a “CLASS” tag, the protocol size will be the one chose, otherwise the class one will. “OFFSET” represents where the signature pattern will be placed inside the packet’s payload, i.e., in which offset of the payload the pattern will start. This three tags have the tags “FROM”, “TO” and “OCCUR”, and they represents the same for all. “FROM” means the upper bound of the bin, “TO” means the lower, and “OCCUR” means the occurrences on that bin. TrafficGen uses the medium value of the bounds to represent the bin. If the application uses both TCP and UDP as its transport protocol, the tag “WEIGHT” sets the amount of traffic that each protocol will represent on the traffic of that application.

3.3.2 TrafficGen Architecture

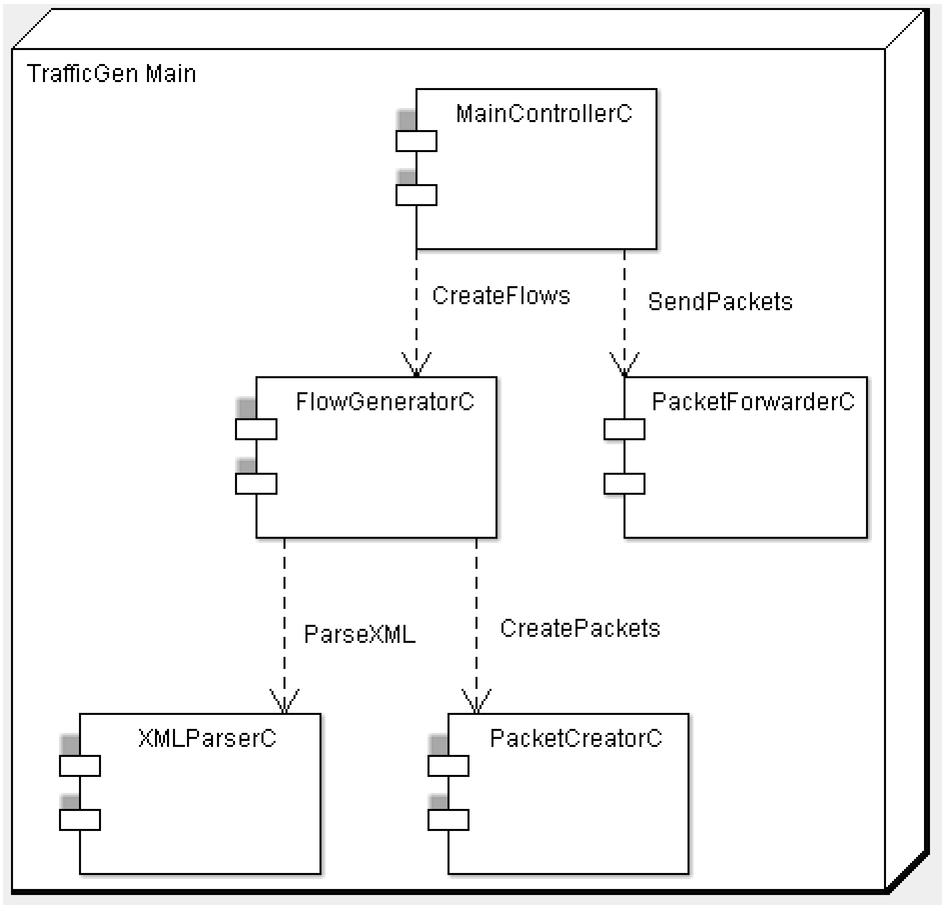


Figure 12 - TrafficGen architecture

The TrafficGen architecture consists of five components. Figure 12 shows a big picture of the architecture. A dashed line from one component to another means that it uses the other features to implement its goals. Figure 13 shows the components interaction as a sequence diagram.

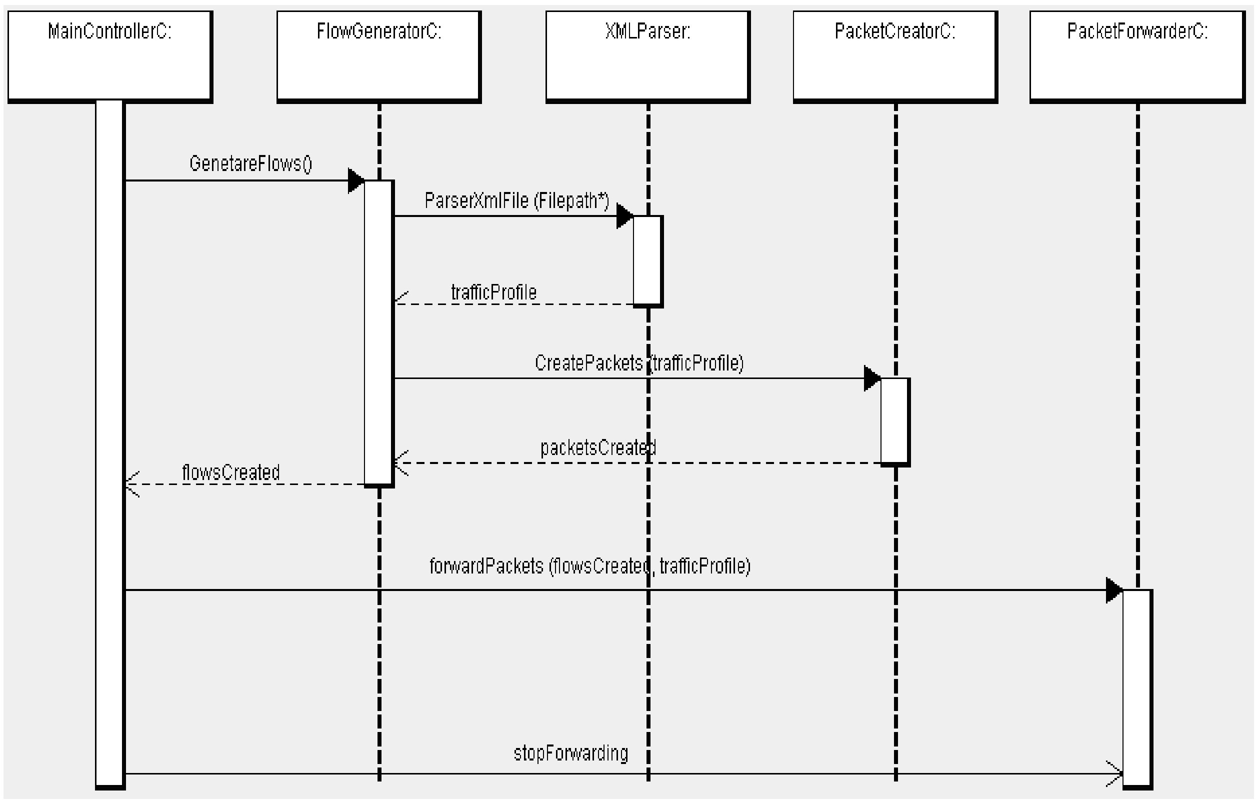


Figure 13 - TrafficGen Components Sequence Diagram

The MainControllerC controls all the TrafficGen execution. This component is responsible for initiate the global variables, parse the command line options, initialize a PF_PACKET socket (see Section 2 for more details) with the egress interface, and set the TrafficGen process's priority and initialize the thread that will be responsible for parse the XML file, create the flows, create the packets for each flow and send the traffic. It is also possible for TrafficGen to work multithreaded as show Figure 14. On multithreaded executions, each thread will have its own instance of each component, to avoid complex synchronizations, and its own socket with the interface.

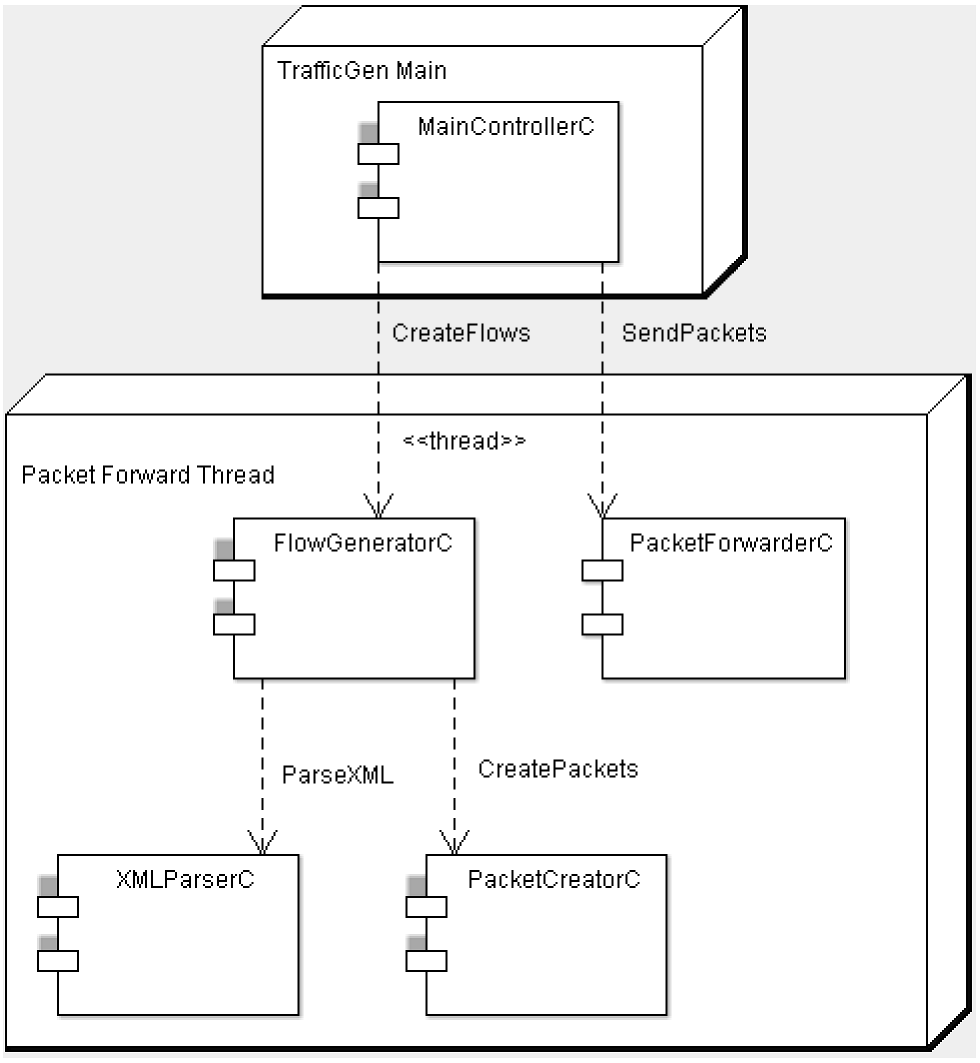
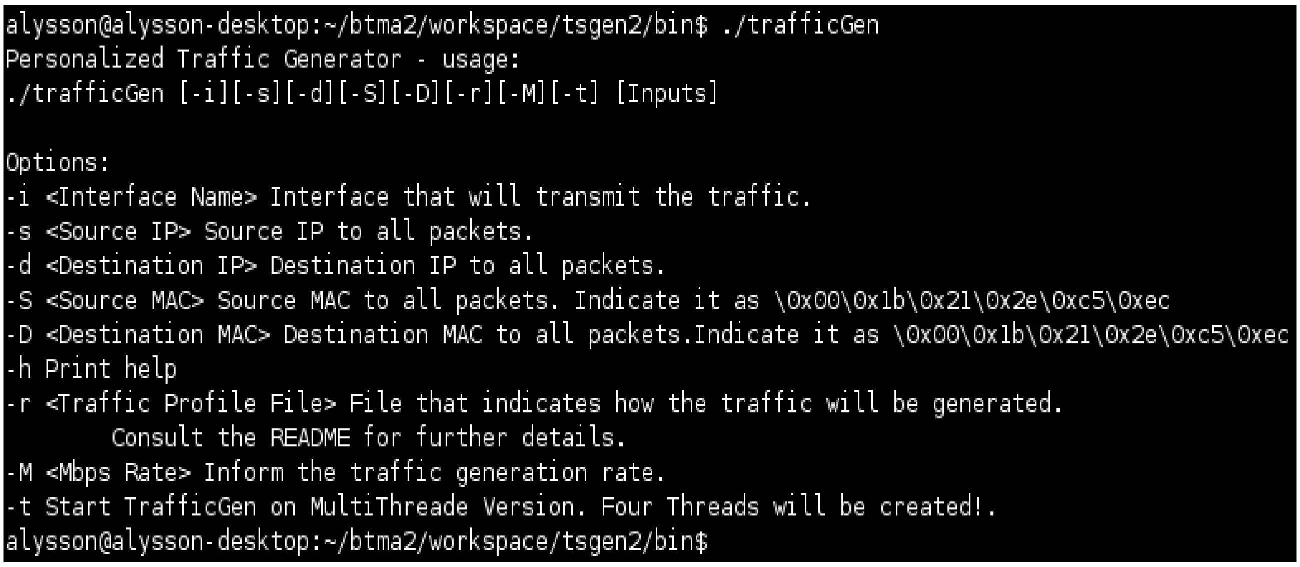


Figure 14 - Multithreaded Architecture

The FlowGeneratorC is responsible for creating the flows. Before begins the packets transmission, TrafficGen creates a pool of flows, to avoid consuming the CPU while transmitting the packets. It creates for each application a UDP and a TCP flow containing 150 packets. If the application only uses TCP as its transport protocol, the UDP flow will never be sent nor will packets be created for it. FlowGeneratorC uses the XMLParserC component to parse the xml configuration file and store it on hash tables, mapping a given application class to its respective weight, each single application to its weight and so on. After the parsing, FlowGeneratorC creates the packets using the PacketCreatorC component for each flow, following the traffic profile configured. To decide which application packet will be sent, which size the packet will have, and so on, PacketCreatorC generates a random number, ranging from 0 to 1, and choose how the packets will look like according the weights of the xml file. For example, if PacketCreatorC generates 0.7

randomly to choose which size a packet will have, the bin representing 70% of the selected application packet size, according to the xml file, will be chose. PacketCreatorC also includes the Ethernet (TrafficGen only works with Ethernet frames), IP and TCP or UDP headers on the packet. All packets belonging to a flow will have the same source and destination ip addresses and the same source and destination ports numbers.

PacketForwarderC will execute right after the flows creation. It is responsible for sending the packets through PF_PACKET sockets, and will loop forever till the user terminates the process by typing CTRL+C, making the MainControllerC stop the forward routine. If TrafficGen is running with multithreads, each thread will have a different socket with the same interface. Currently TrafficGen only transmit packets through a single NIC. If all the packets of a certain flow have been sent, TrafficGen starts sending the flow again from the beginning. Figure 15 shows TrafficGen command line help. TrafficGen allows the user to inform the source and destination IPs to be placed on the packets, as well as the source and destination MAC address.

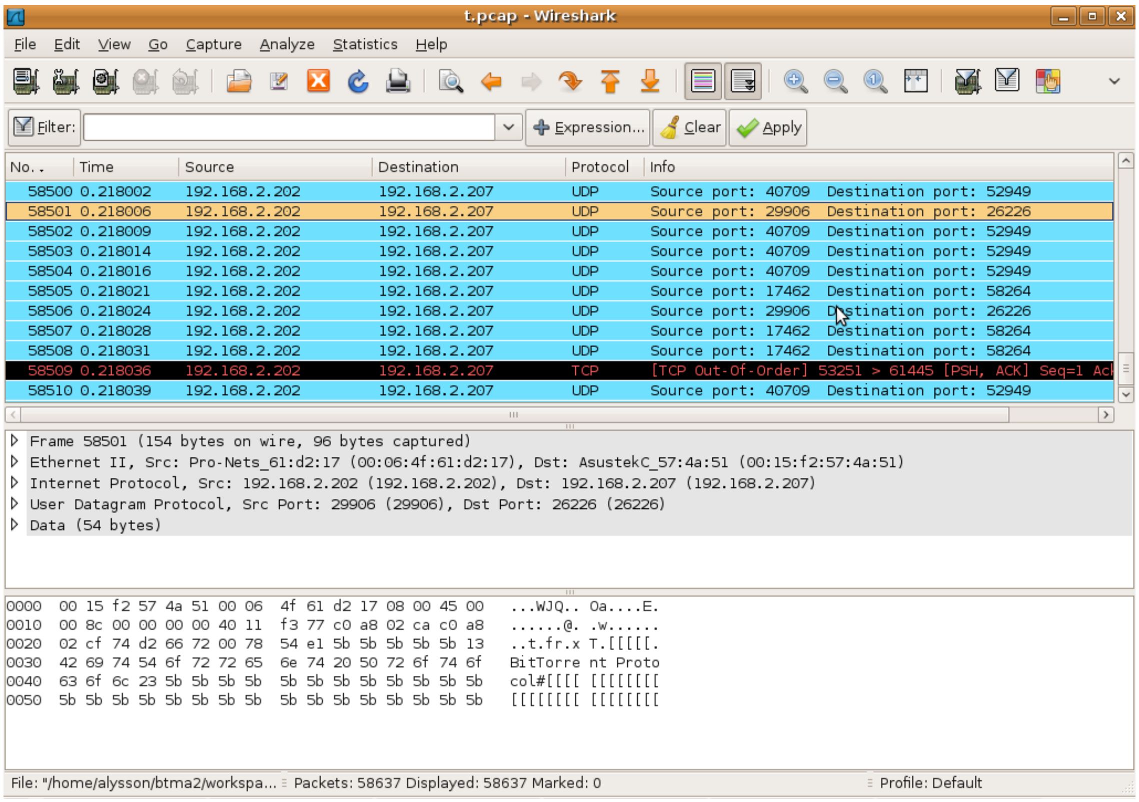


```
alysson@alysson-desktop:~/btma2/workspace/tsgen2/bin$ ./trafficGen
Personalized Traffic Generator - usage:
./trafficGen [-i][-s][-d][-S][-D][-r][-M][-t] [Inputs]

Options:
-i <Interface Name> Interface that will transmit the traffic.
-s <Source IP> Source IP to all packets.
-d <Destination IP> Destination IP to all packets.
-S <Source MAC> Source MAC to all packets. Indicate it as \0x00\0x1b\0x21\0x2e\0xc5\0xec
-D <Destination MAC> Destination MAC to all packets.Indicate it as \0x00\0x1b\0x21\0x2e\0xc5\0xec
-h Print help
-r <Traffic Profile File> File that indicates how the traffic will be generated.
        Consult the README for further details.
-M <Mbps Rate> Inform the traffic generation rate.
-t Start TrafficGen on MultiThreade Version. Four Threads will be created!.
alysson@alysson-desktop:~/btma2/workspace/tsgen2/bin$
```

Figure 15 - TrafficGen Command Line help

Figure 16 presents an example of a packet generated by TrafficGen on the interface of a very popular network protocol analyzer (also known as a traffic sniffer) called wireshark [24]. As is shown, TrafficGen places the signature on the exact offset of the payload, indicated on the XML configuration file, and fills the rest of the payload with random characters.



```
No. .  Time      Source          Destination     Protocol Info
 58500 0.218002  192.168.2.202   192.168.2.207   UDP      Source port: 40709  Destination port: 52949
 58501 0.218006  192.168.2.202   192.168.2.207   UDP      Source port: 29906  Destination port: 26226
 58502 0.218009  192.168.2.202   192.168.2.207   UDP      Source port: 40709  Destination port: 52949
 58503 0.218014  192.168.2.202   192.168.2.207   UDP      Source port: 40709  Destination port: 52949
 58504 0.218016  192.168.2.202   192.168.2.207   UDP      Source port: 40709  Destination port: 52949
 58505 0.218021  192.168.2.202   192.168.2.207   UDP      Source port: 17462  Destination port: 58264
 58506 0.218024  192.168.2.202   192.168.2.207   UDP      Source port: 29906  Destination port: 26226
 58507 0.218028  192.168.2.202   192.168.2.207   UDP      Source port: 17462  Destination port: 58264
 58508 0.218031  192.168.2.202   192.168.2.207   UDP      Source port: 17462  Destination port: 58264
 58509 0.218036  192.168.2.202   192.168.2.207   TCP      [TCP Out-Of-Order] 53251 > 61445 [PSH, ACK] Seq=1 Ack
 58510 0.218039  192.168.2.202   192.168.2.207   UDP      Source port: 40709  Destination port: 52949

Frame 58501 (154 bytes on wire, 96 bytes captured)
Ethernet II, Src: Pro-Nets_61:d2:17 (00:06:4f:61:d2:17), Dst: AsustekC_57:4a:51 (00:15:f2:57:4a:51)
Internet Protocol, Src: 192.168.2.202 (192.168.2.202), Dst: 192.168.2.207 (192.168.2.207)
User Datagram Protocol, Src Port: 29906 (29906), Dst Port: 26226 (26226)
Data (54 bytes)

0000  00 15 f2 57 4a 51 00 06  4f 61 d2 17 08 00 45 00   ...WJQ.. Oa....E.
0010  00 8c 00 00 00 00 40 11  f3 77 c0 a8 02 ca c0 a8   ......@. .w......
0020  02 cf 74 d2 66 72 00 78  54 e1 5b 5b 5b 5b 5b 13   ..t.fr.x T.[[[[[.
0030  42 69 74 54 6f 72 72 65  6e 74 20 50 72 6f 74 6f   BitTorre nt Proto
0040  63 6f 6c 23 5b 5b 5b 5b  5b 5b 5b 5b 5b 5b 5b 5b   col#[[[[ [[[[[[[[
0050  5b 5b 5b 5b 5b 5b 5b 5b  5b 5b 5b 5b 5b 5b 5b 5b   [[[[[[[[ [[[[[[[[

File: "/home/alysson/btma2/workspa... Packets: 58637 Displayed: 58637 Marked: 0    Profile: Default
```

Figure 16 - TrafficGen Generated Packet on Wireshark

3.3.3 Controlling the generation rate

Generate traffic on controlled rates are important to organize a network testbed environment. Different machines generating packets on random rates are not well suited for large scale tests, speacially on DPI system online tests. TrafficGen is capable of control the packet forwarding rate in Mbits/s, through an input value given by the user.

To implement the rate control, TrafficGen sums all the bits sent by the NIC. Assuming that the user chose 500Mbits/s as its preferable rate, TrafficGen will divide a second into ten 100 ms (microseconds) bins. Dividing 500 by 10, TrafficGen discover that it will need to send 50 Mbits during 100ms. If the summing of the sent bits reaches 50 Mbits, or more, and 100ms has not been elapsed, TrafficGen sleeps for the rest of the time, till 100ms, using the Linux nanosleep function. Nanosleep was chose due to its nanosecond timing resolution. If TrafficGen sends 50 Mbits in more than 100ms, it does not sleeps and continue sending packets on the machine maximum speed.

The decision to divide the second into ten 100ms bins was made to avoid a high burstiness on the first portion a second of the generation traffic, which could make a switch or a router discard packets. If TrafficGen do not divide the second, on the first microseconds of a second the NIC was going to transmit packets on the maximum speed that it could and would sleep till one second elapsed, generating burst on the beginning of a second. Reducing the bursts on windows of 100ms, distribute the rates on a more uniform manner, decreasing the loss probability. TrafficGen only permits speed control on its single thread version. The multithread version generates packets at the NIC full speed.

5 EVALUATION AND RESULTS

In this section, the results of the evaluations performed on each tool are presented. For each tool are presented different results. For SigGen, are presented some signatures from famous P2P applications such as BitTorrent, Ares, and Emule and from a web streaming application such as Sopcast. It was also generated signatures for IP2PTV applications such as Tvants and Tvuplayer. Both are very famous Chinese TV broadcast applications. Besides the signatures, this Section presents the memory requirements for different sized trace files, the influence that the parameters have on the generation process and the precision of the signatures on flow classification.

For PatternGen, are presented some histogram generated by the tool for each option that it provides. To evaluate TrafficGen, some performance tests were needed. The maximum throughput (in Mbps) that the tool could perform for different numbers of applications as well as the memory required on each test.

5.1 Evaluating the Signature Generator (SigGen)

In order to provide a proof-of-concept that SigGen is capable of automatically generate signatures for applications protocols, this section will present some of the patterns founded by SigGen. Table 5.1 shows the packet trace captured by the already mentioned process sniffer.

Table 5.1 - Trace File captured for SigGen

Application	Trace File size	Number of Flows
BitTorrent	1.1 GB	32404
Ares	14 MB	3097
Emule	115 MB	5890
Sopcast	337 MB	1724
Tvants	162 MB	1256
Tvuplayer	504 MB	4482

SigGen were evaluated while generating signatures for all the applications on Table 5.1 in all aspects. Considering different values of the merge score parameter and different types of common substring selection threshold, the relevance of these parameters on the number of substrings generated will be presented as follows. Figure 17 shows different values of the common substring selection threshold the number of substring generated, without dividing the flows into data and control. Remember that this threshold ranges from 0 to 1. For the tests performed on Figure 17 the merge score parameter had the same value 0.4 for tests performed for each application.

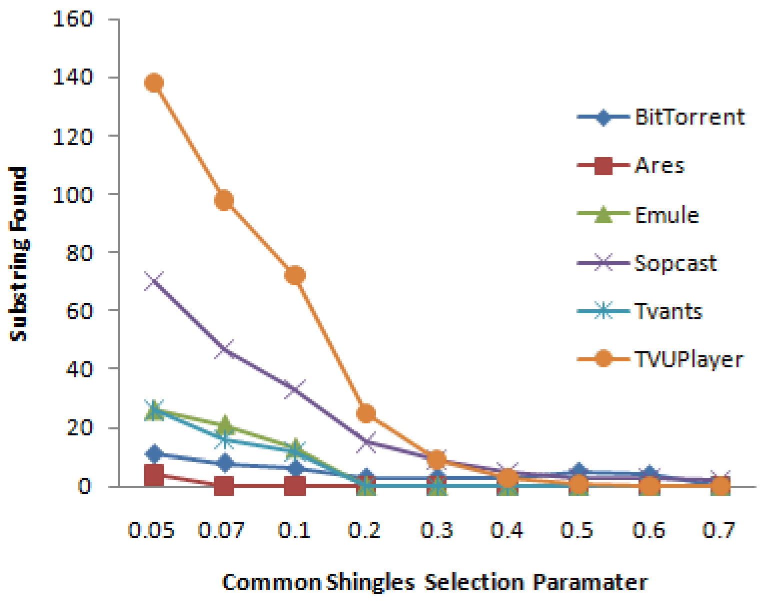


Figure 17 - Common Shingle Selection Parameter Variation

As expected, the amount of substring generated with values close to 0 is bigger than with values close to 1. This happens because it is very difficult to have patterns that are present in more than 70% of the flows, but this may vary from application to application. The user must set this parameter empirically for each application trace file, trying to find signatures that occur in the maximum numbers of flows as possible.

Figure 18 shows the relationship between the numbers of generated substring with the merge score parameter. As the common substring selection threshold, the merge score also ranges from 0 to 1.

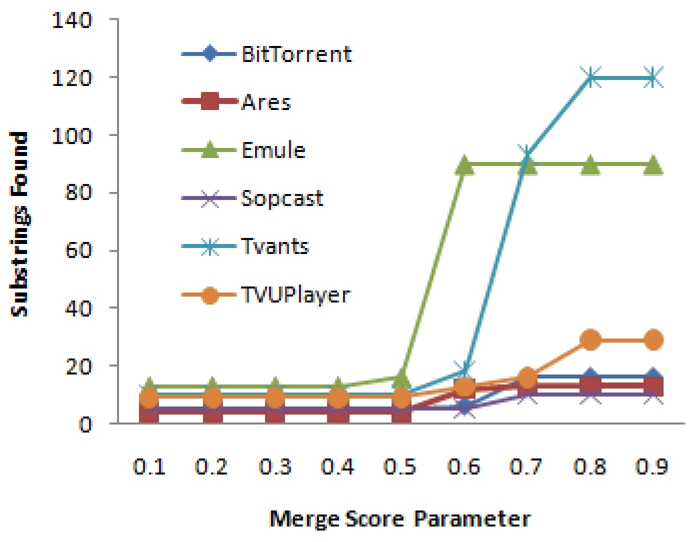


Figure 18 - Merge Score Parameter Variation

One can infer from Figure 18 that depending on the application trace file, lower merge score values tends to generate fewer substrings than higher ones. This is explained by the fact that for lower merge score values, the resultant merged substring must appear in fewer flows, making SigGen merge the substrings more frequently. With higher merge score values, SigGen may not merge the strings correctly, creating the signatures as shorter sequences. Regard that if it is used lower values of merge score, SigGen will work similar to the greedy merge algorithm, generating unspecific and generic signatures. It is important to understand that these results were obtained with the trace files illustrated on Table 5.1. For different traces, the results probably will not be similar.

Table 5.2 - SigGen memory usage

Application	RAM Memory Required
BitTorrent	1,92 GB
Ares	44 MB
Emule	171 MB
Sopcast	451 MB
Tvants	94 MB
Tvuplayer	1,055 GB

Table 5.2 show the average amount of RAM memory required to generate the signatures from the applications protocols. Large trace file requires more memory

than small ones because the flows payload must be kept on the memory during all signature generation process. A copy of the flows must also be stored to compose the common shingles and unique flows table, although this table will only have the basic flow attributes (same protocol, source and destination IP addresses and port numbers), without needing to store the flows payload. If many common shingles are selected, and there are many flows on the trace file, SigGen will need GBytes of RAM memory. If the machine supports memory swapping, SigGen will work properly even with large trace files, although it may lead to a poor performance, due to the necessity of page swapping by the OS memory management process, and may compromise the use of the machine by others concurrent processes. The biggest tested file was 1 GB, as can be seen on Table 5.1.

Table 5.3 - Signatures Generated

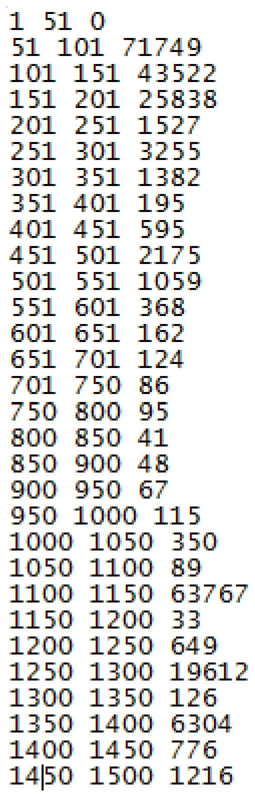
Application	Signature
BitTorrent	[\0x13BitTorrent Protocol\0x20];
Ares	[\0x08\0x01\0x02\0x01\0x06];
Emule	[0x02\0x01\0x01\0x18http://emule-project.net\0x03\0x01\0x11];
Sopcast	[\0x05(\0x05\0x02];[\0x13\0x88\0x02\0x02];
Tvants	[TVANTS SHARE\0x08\0x06\0x11\0x01PV\0x11];
Tvuplayer	[\0xFF\0xFF\0xFF\0xFF\0x13\0x88\0x02\0x02];

Table 5.3 shows some signatures generated by SigGen for the traces presented on Table 5.1. Table 5.3 only has the most relevant signature extracted from the sample flows with SigGen, i.e., the patterns that appeared on more sample flows as possible. Some signatures were generated by using the blacklist and the flow division feature. Many signatures pointed by SigGen may lead to poor classification, biasing to false-positives. This may happen because SigGen can encounter patterns that are only common on the used trace file. For example, on the BitTorrent tests, SigGen pointed the hash value used to identify the file being transferred as a common string. Although it indeed was presented on many flows, it will not be a good pattern to classify flows with a DPI system, because it is too

specific for the trace collected. To circumvent these problems, the user can use the blacklist feature, to discard signatures that he knows that can lead to false-positives on future classifications.

5.2 Evaluating the Pattern Generator (PatternGen)

To evaluate the PatternGen, this Section will expose the histogram that can be generated with it. PatternGen generates the histogram on two different types of files; a text file, representing each bin of the histogram and its occurrence, and a PNG graph file generated with the Linux GNU Scientific Library (GSL) [22].



```
1 51 0
51 101 71749
101 151 43522
151 201 25838
201 251 1527
251 301 3255
301 351 1382
351 401 195
401 451 595
451 501 2175
501 551 1059
551 601 368
601 651 162
651 701 124
701 750 86
750 800 95
800 850 41
850 900 48
900 950 67
950 1000 115
1000 1050 350
1050 1100 89
1100 1150 63767
1150 1200 33
1200 1250 649
1250 1300 19612
1300 1350 126
1350 1400 6304
1400 1450 776
1450 1500 1216
```

Figure 19 - Tvants Packet Size distribution File

Figure 19 and Figure 20 illustrate a histogram text file containing all the bins of the histogram and an application packet size histogram graph generated by PatternGen for Tvants application trace file, mentioned on Table 5.1, respectively.

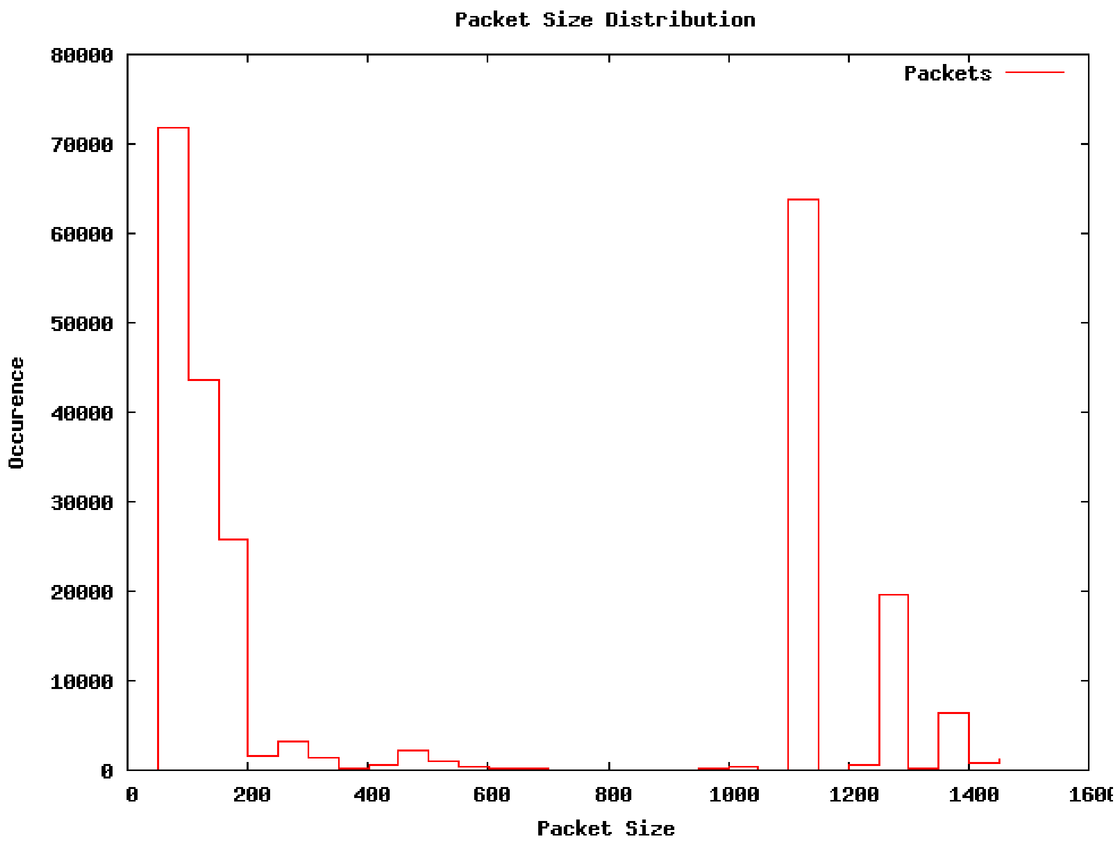


Figure 20 - Tvants packet size distribution graph

PatternGen creates the histogram File because the user may use it to configure the TrafficGen XML profile if the bins values. The graph is generated to make possible for the user to visualize all the modes of the histogram, to choose one bin as the Data-Control flows division limit, before execute SigGen. On Figure 20 graph, 800 lengthened packets could be the application Data-Control flows division limit, as this histogram seems to be bimodal.

The user can generate a packet size histogram for each transport layer protocol, UDP and TCP, once many applications uses TCP for control traffic and UDP for data transfer. Figure 21 and Figure 22 illustrate the same tvants distribution for UDP and TCP protocol, respectively.

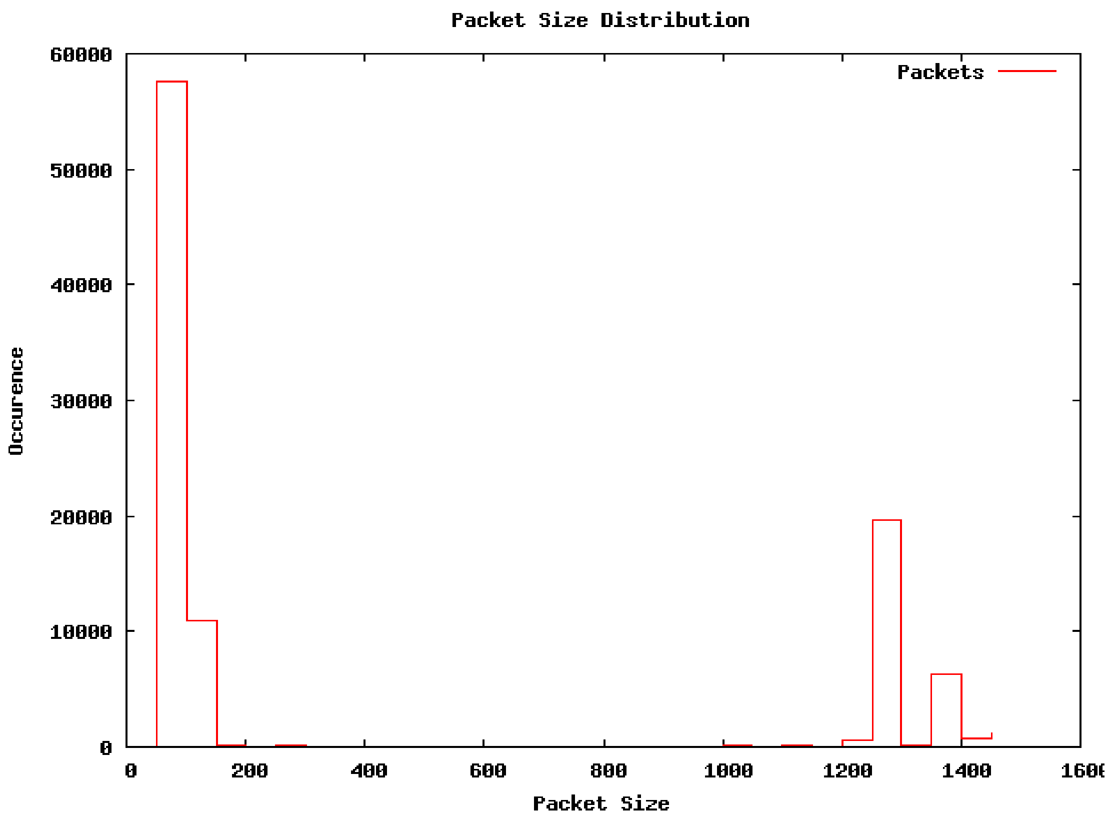


Figure 21 - Tvants TCP packet size distribution graph

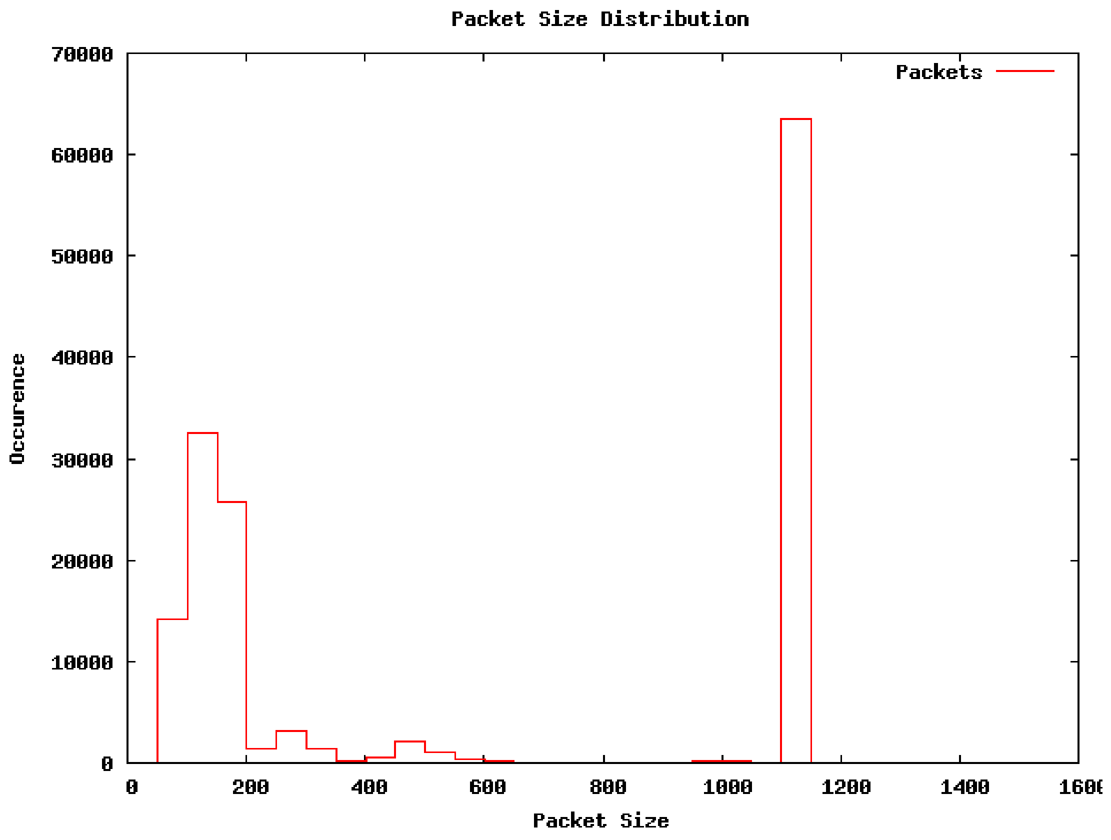


Figure 22 - Tvants UDP packet size distribution graph

As expected, Figure 21 and 22 kept the bimodal characteristic of the distribution. By looking at the Figures, it is noticeable that the TCP traffic are mostly composed by small sized packets, probably with control responsibilities, and the UDP are used to transport large data files. These insights are very valuable to SigGen, once the user can generate signatures to identify solely control or data flows.

The others histograms generated by PatternGen are similar to those presented on this Section. Another interesting option that PatternGen offers to the user is the possibility to visualize a histogram containing the exact position that the generated signature appears on the packets' flows payload. Figure 23 shows a graph of the signature "TVANTS SHARE\0x08\0x06\0x11\0x01PV\0x11", from tvants application, generated by SigGen.

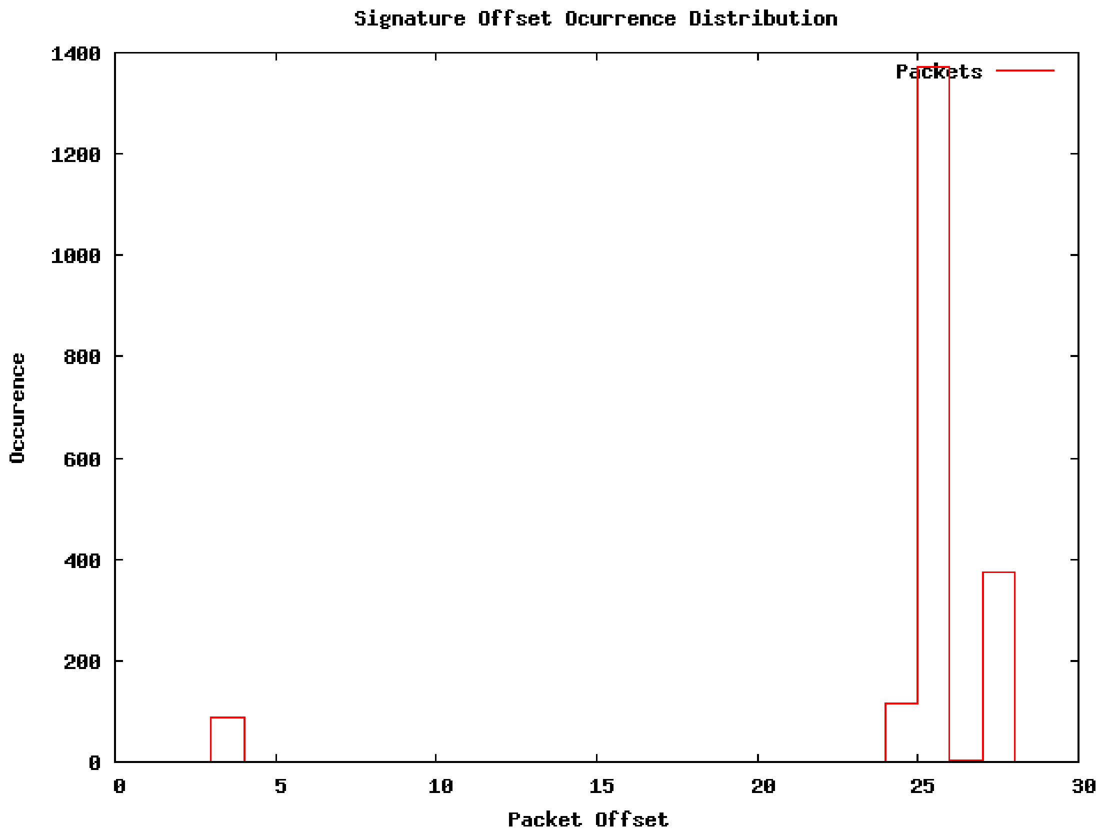


Figure 23 - Tvants signature offset distribution

By Figure 23 one can see that the signature occur more often on the 25th byte of the packets' payload.

5.3 Evaluating the Traffic Generator (TrafficGen)

TrafficGen, as a traffic generator tool, will be evaluated by different aspects. One aspect is the total transmission throughput, measured in Mbps (Mega bits per second) and in Mpkts/sec (Mega packets per second), once it is important to quantify its generating capabilities. A comparison between the single thread and multithread versions will be presented, as well as the total RAM memory required and the CPU usage rates. Another important aspect is the scalability of the tool, while generating traffics for different numbers of class and per applications classes. A graph containing its measurements will be provided on this Section. The machine used for the traffic generation is described on Table 5.4.

Table 5.4 – Traffic Generator Machine Configuration

Machine	Processor	RAM Memory	Administrative NIC	Traffic Generator/Receiver NIC	HD	Operating System
M	Intel Xeon X3210 Quad-core	4GB DDR	Onboard Gigabit	Offboard, 3Com Gigabit	3x 500GB Sata HDs	Linux, 2.6

4.3.1 TrafficGen Throughput

Figure 24 and 25 shows the transmission rates achieved by TrafficGen single thread and multithread version, respectively. The machine generated traffic through 20 min each.

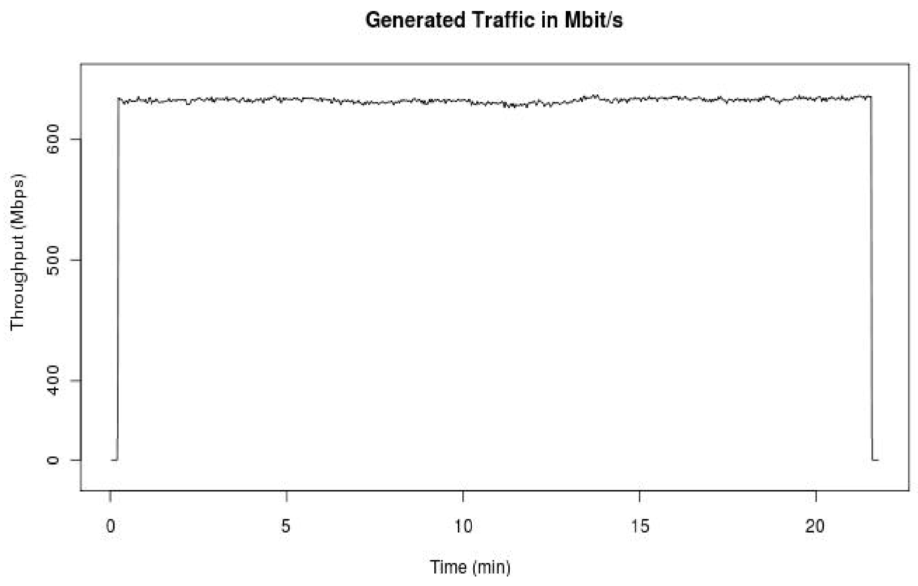


Figure 24 - TrafficGen Single Thread Version

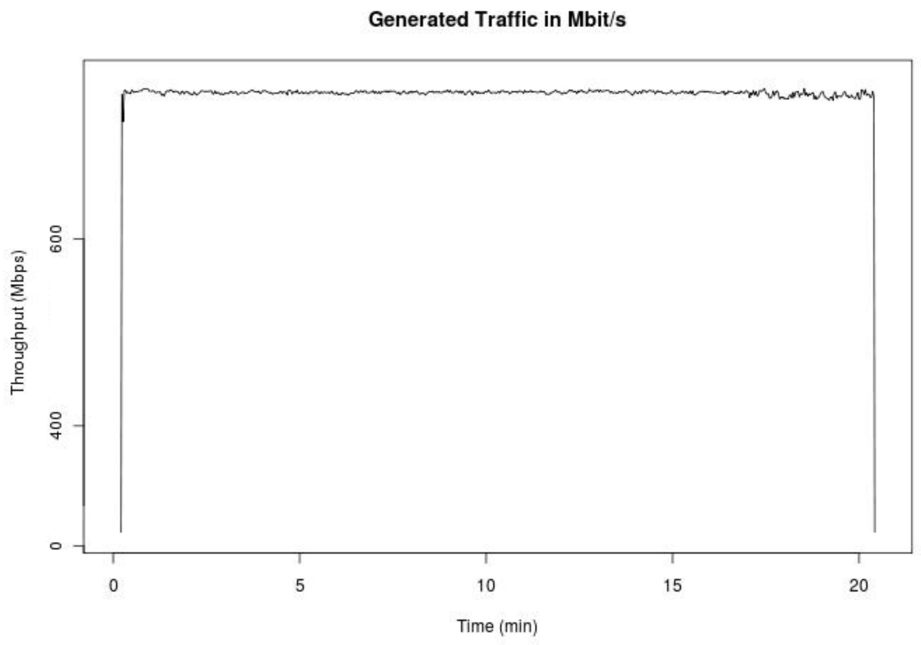


Figure 25 - TrafficGen Multithread Version

On its single thread version, TrafficGen reached an average throughput of 630 Mbits/s, transmitting 0.51 Mpkts/sec creating 1200 different flows from three different applications. It was needed to verify its performance on packets per second to see if it could generate high rates even with small sized packets. The packets generated on these tests were 112 Bytes sized from IP to application OSI layer, excluding the Ethernet frame portion. The transmission throughput generated by the multithread version was not much different than the single thread, reaching 750 Mbits/sec on average, sending 0.61 Mpkts/sec, creating 4800 different flows from three different applications. The same IP lengthened packets were generated. Although the throughput have not improved much on the multithread version, in terms of packets per second it could create 0.1 Mega extra packets per second, which is a very considerable improvement, making the multithread version more suitable for send small sized packets.

Table 5.5 shows the memory consumed and the CPU used by each TrafficGen version. Regard that the single thread version is capable of control the generation rates, while the multithread works on the machine and NIC maximum speed, although the throughput tests were conducted without rate controlling. The 390% CPU usage is the sum of the four machine cores.

Table 5.5 - TrafficGen memory and CPU requirements

Version	CPU usage	RAM memory used
Single Thread	100% - one core	278 MB
Multi Thread	390% - four cores	1,04 GB

4.3.2 TrafficGen Scalability

It is important to test TrafficGen on different scenarios to see how it behaves and scales. To tests its scalability, while executing with different kinds of traffic profiles, tests with 10, 20, 30, 40 and 50 different applications were made. The most relevant scale attributes considered on each test where the memory required, the total throughput in Mbits/sec and in Mpkts/sec. All the tests were performed with the Single thread version running at its full capacity, without the rate control option, and the configuration XML files with the same packet size, to not add a difference on the total traffic outputted in Mbits/s between the tests. Additionally, the tests were executed for 20 min each.

Table 5.6 shows the different amount of memory required for each TrafficGen execution.

Table 5.6 - Scalability Memory Requirements

Number of Applications	RAM memory required
10	661 MB
20	1,27 GB
30	1,89 GB
40	2,26 GB
50	2,95 GB

As expected, when more applications are added to the traffic profile XML, more memory TrafficGen will consume. This happens because it is necessary to keep all the flows, and its packets, on memory during the execution. Hence, more applications implicates on more memory usage.

Table 5.7 shows the average throughput in Mbits/sec and in Mpkts/sec for each different test.

Table 5.7 - Scale Throughput

Number of Applications	Throughput in Mbits/s	Throughput in Mpkts/s
10	590	0.455
20	582	0.440
30	578	0.422
40	572	0.384
50	556	0.358

As expected, by adding more applications to the configuration file the throughput in Mbits/sec and in Mpkts/s produced by TrafficGen reduces. This happens because TrafficGen must decide, by generating random numbers, which packet from which application will be sent on the socket. With more applications, more complex this decision will be. Regard that this result were achieved on the machine described by Table 5.4.

6 DISCUSSION

In the previous section a set of tools aiming at ease the work of an ISP administrator of construct, maintain and update signatures for its DPI systems were described. SigGen generates new signatures, pointing out the most common patterns that appear on a set of pre-captured sample flows, PatternGen creates histograms and helps to characterize the generated signatures and finally TrafficGen creates real point to point traffic flows, with packets containing detectable signatures by DPI systems.

Although the tools aid the ISP administrator on signatures construction and deployment, a human intervention is still needed, due to the specificities that each application has in face of the others, turning the parameters calibration a difficult task. Besides that, to generate signatures capable of classify as many flows as possible, within a high precision, it is important that a human check the signatures found and evaluate them, discarding false-positives and noise patterns. Finally, if a DPI works with regular expressions, further analysis may be required to turn the substrings into RegEx. SigGen is also requires a lot of memory to run its experiments, which can make its performance less efficient due to the necessity of pages swapping. SigGen also does not shows the volume in bytes involved on the flows that carry a certain generated signature. This are a very important statistic, because a signature can appear in many flows, but these flows may not represent a lot of traffic in bytes. Additionally, the sample flows used as input can impact directly on the generated patterns. The tools presented here plays an important role on the signature generation process, in which they can reduce the time spent on the most critical task, which is the most common and relevant substrings identification, besides the possibility of test the new signatures on real traffic simulations.

Additionally, it was not evaluated on this thesis how TrafficGen works on real online DPI tests, in order to check how it is respecting the configuration XML file, due to the lack of a DPI system that could recognize the generated signatures. Build one was out of the scope of this thesis. Further tests must be performed, in order to evaluate if its precision is accurate and if it can be largely used by DPI systems manufactures on online tests. Only offline tests, to check if the packets were being correctly created, during its construction were performed.

7 CONCLUDING REMARKS AND FUTURE WORKS

This thesis presented a set of tools that might be used by DPI system maintenance personnel to construct, update and test the signatures used on traffic classification. The time spent to perform this task will be reduced, and the necessity of extra consultancy from a protocol expert will be discarded. Besides reduce the signature construction time, the tools aid the user to build more efficient and accurate signatures, making its DPI system more reliable on traffic classification.

The signature generator, SigGen, proved to be very effective on signatures creation, although it uses too much memory and must have its parameters calibrated manually, which make it less automatically, but ease the signatures construction job. It is better to use SigGen with large trace flows file, although it may require more memory, it will generate more accurate and specific signatures. To reduce the noise on the generated signatures, it is recommended to divide the flows into Data and Control, as well as the blacklist feature, to discard probably false-positives patterns. A further study on its memory usage is required on future works, besides an extended analysis on the viability of an automatic parameter calibration to avoid the human intervention dependence. Inform the amount of bytes the flows that carry a specific signature is also a good contribution to be further included. More tests with the signatures precision shall also be included.

PatternGen usage is very straight-forward and must be used as an extra tool on signature characterization, to analyze the exact position of the signatures on the packets' payload and to know exactly in which flow's packet it appears, besides to have a further analysis on the protocol that carries the signatures most commonly and how is the average packet size. These are very important characteristics to build more efficient DPI system signatures.

Finally, the traffic generator TrafficGen is a tool that provides to the user the possibility of test its new signatures online, by generating point to point configured traffic, controlling the generation rates, the traffic profile per application class and per specific application. TrafficGen can also be used for network stress and DPI performance tests. Test TrafficGen online on a real DPI system that can identify the signatures being sent is an important aspect to be analyzed on future works.

REFERENCES

[1] Mingjiang Ye, Jianping Wu, Ke Xu, "AutoSig-Automatically Generating Signatures for Applications", Proc. Of IEEE 9th International Conference on Computer and Information Technology (ICTI), Xiamen, China, October 11 – 14 , 2009 .

[2] Byung-Chul Park, Young J. Won, Myung-Sup Kim, and James Won-Ki Hong. 'Towards Automated Application Signature Generation for Traffic Identification,' Proc. of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2008), Salvador, Brazil, April 2008, pp. 160-167.

[3] Haffner, P., Sen, S., Spatscheck, O., and Wang, D. 2005. ACAS: automated construction of application signatures. In Proceedings of the 2005 ACM SIGCOMM Workshop on Mining Network Data (Philadelphia, Pennsylvania, USA, August 26 - 26, 2005). MineNet '05. ACM, New York, NY, 197-202.

[4] A. Moore and K. Papagiannaki, 'Toward the Accurate Identification of Network Applications', Passive and Active Measurements Workshop, Boston, MA, USA, March 31, April 1, 2005.

[5] Newsome, J., Karp, B., and Song, D. 2005. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In Proceedings of the 2005 IEEE Symposium on Security and Privacy (May 08 - 11, 2005). SP. IEEE Computer Society, Washington, DC, 226-241.

[6] Kim H, Karp B. Autograph: Toward automatic distributed worm signature detection. In: Proc. of the USENIX Security Symp. Diego, 2004. 271-286.

[7] Singh, S., Estan, C., Varghese, G., and Savage, S. 2004. Automated worm fingerprinting. In Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation – Volume 6 (San Francisco, CA, December 06 - 08, 2004). USENIX Association, Berkeley, CA, 4-4.

[8] www.snort.org – Accessed on March 1, 2009.

[9] Mingjiang Ye, Jianping Wu, Ke Xu, Dah Ming Chiu, 'Identify P2P Traffic by Inspecting Data Transfer Behaviour', In Proceedings of IFIP Networking, Aachen, Germany, May 11 – 15, 2009;

[10] A. McGregor, M. Hall, P. Lorier, and J. Brunskill. Flow Clustering Using Machine Learning Techniques. In PAM, 2004.

[11] S. Sen O. Spatscheck and D. Wang, "Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures," Proc. Of ACM WWW'04, 2004.

[12] Bernaille, L., Teixeira, R., Akodkenou, I., Soule, A., and Salamatian, K. 2006. “Traffic classification on the fly,” *SIGCOMM Comput. Commun. Rev.* 36, 2 (Apr. 2006), 23-26.

[13] Yu, F., Chen, Z., Diao, Y., Lakshman, T. V. and Katz, R. H., “Fast and memory-efficient regular expression matching for deep packet inspection,” *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*. ANCS '06. ACM, New York, NY, 93-102, 2006.

[14] http://www.bittorrent.com/ – Accessed on March 12, 2009.

[15] Antichi G., Di Pietro A., Ficara D., Giordano S., Procissi G., Vitucci F., "Design of a High Performance Traffic Generator on Network Processor", DSD08 - Euromicro, Sept. 03, 2008, Parma Italy

[16] S. Zander, D. Kennedy, e G. Armitage. KUTE – A High Performance Kernel-based UDP Traffic Engine. Technical Report 050118A, January 2005, Swinburne University of Technology, Melbourne Australia.

[17] http://rude.sourceforge.net – Accessed on April 23, 2009.

[18] http://mgen.pf.itd.nrl.navy.mil/ – Accessed on April 23, 2009.

[19] A. Botta, A. Dainotti, A. Pescapè, “Multi-protocol and multi-platform traffic generation and measurement”. INFOCOM 2007 DEMO Session, May 2007, Anchorage (Alaska, USA).

[20] N. Bonelli, S. Giordano, G. Procissi, R. Secchi, “Brute: A high performance and extensible traffic generator,” in Proceedings of SPECTS05, July 24-28, 2005, Philadelphia, USA.

[21] Benvenuti, Christian, “Understanding Linux Network Internals”, U.S.A.: O’Reilly, December 2005.

[22] http://www.gnu.org/software/gsl/ - Accessed on June 10, 2009.

[23] http://www.isi.edu/nsnam/ns/ - Accessed on June 16, 2009.

[24] http://www.wireshark.org - Accessed on June 17, 2009.

[25] http://www.home.agilent.com/ - Accessed on June 03, 2009.