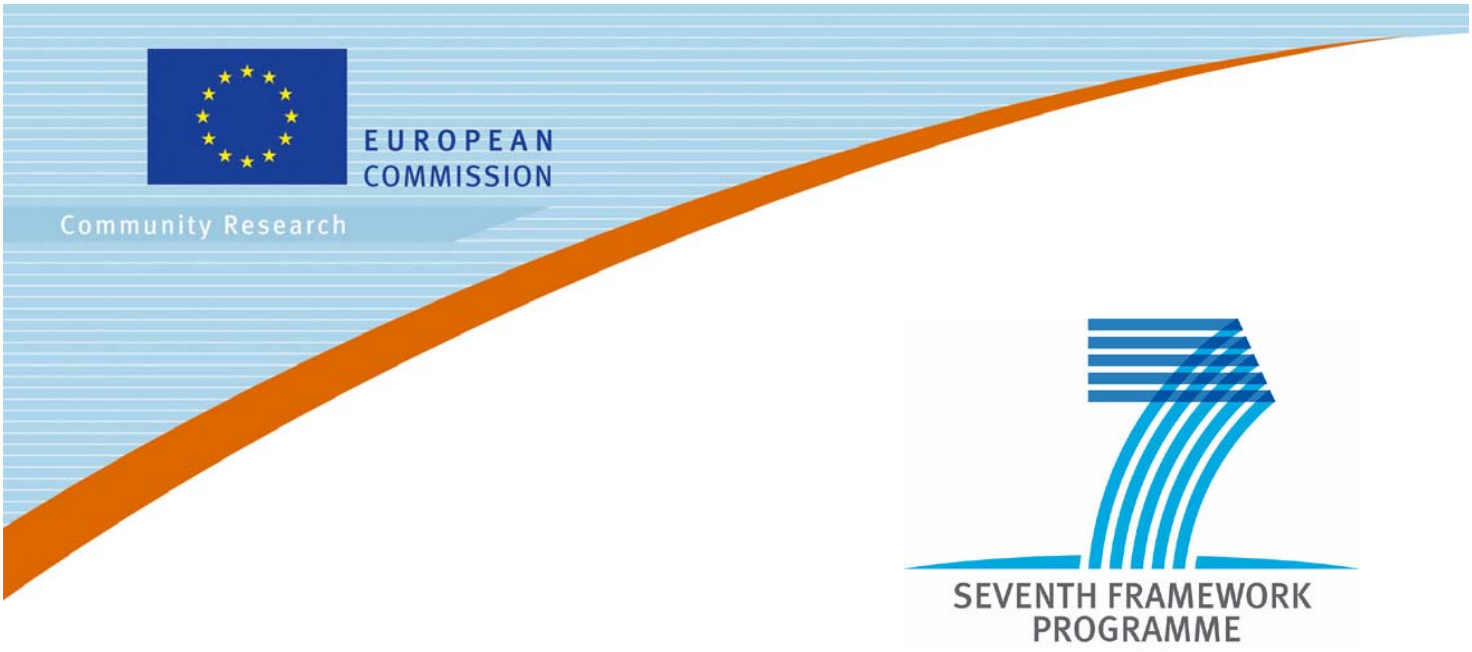

Specific Targeted REsearch Project

PRISM

D3.2.2: Preliminary monitoring applications specification and analysis

Project acronym: PRISM
Project full title: Privacy-Aware secure Monitoring
Contract No.: 215350
Project Document Number: IST-2007-215350-WP3.2-D3.2.2-R1
Project Document Date: 28/02/2009
Workpackage Contributing to the Project Document: WP3.2
Deliverable Type and Security: Public
Author(s): Felix Strohmeier, Peter Dorfinger (Salzburg Research)
Ivan Gojmerac, Esa Hyytiä (FTW)
Brian Trammell, Elisa Boschi (Hitachi)
G. Bianchi, A. Di Pietro, P. Loreti, M. Pomposini, G. Procissi, S. Teofili, F.Vitucci (CNIT)
G. Lioudakis, F. Gogoulos, A. Antonakopoulou, D. Kaklamani, I. Venieris (ICCS)
Maciej Matachowski, Sophia Khavtasi (Telscom)

Abstract:

The PRISM project introduces a novel design for network traffic monitoring applications. Traditional network monitoring applications use a "gather first, process later" operation. In this case, data protection and data reduction are static mechanisms generally applied in a one-off fashion regardless of the specific intent behind the monitoring process. Instead, PRISM moves traffic analysis and data reduction to the edge of the measurement system where possible, and replaces general techniques with specific analyses targeted toward specific tasks. This approach allows aggressive data reduction and protection for scalability as well as privacy protection, and provides technical enforcement of the proportionality principle of privacy preservation.

This deliverable is the intermediate output of the work carried out in the project work-package WP3.2. This work package explores the application of the PRISM design to real world monitoring applications, the technical monitoring processing possibilities that can be performed "on-the-fly" on a front-end device, the partition of monitoring applications into front-end and back-end parts, and advantages in terms of scalability and privacy preservation of the monitoring process as a whole. In this first phase of the project the effort has been mainly dedicated to addressing technical solutions to partition application processing to front-end, back-end, or external components consuming PRISM-protected data. This deliverable further documents some preliminary insights and approaches concerning concrete monitoring scenarios, as an anticipation of solutions that will be further explored and presented in full details in the next deliverable D.3.2.3.

Keyword list: PRISM, IST-2007-215350, Monitoring Applications, lightweight processing mechanisms, Bloom filters, embedded processing.

Table of Contents

Abbreviations

(A)IDS	Anomaly Intrusion Detection System
(A)IPS	Anomaly Intrusion Prevention System
API	Application Programming Interface
AS	Autonomous System
BE	back-end
BF	Bloom filter
CBF	Counting Bloom filter
CDP	Cisco Discovery Protocol
DPI	Deep Packet Inspection
DDOS	Distributed Denial of Service
DTF	Data Transformation Functions
DTWSL	Data Transformation Workflow Specification Language
FE	front-end
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
ID	Identifier
IDS	Intrusion Detection System
IMSI	International Mobile Subscriber Identity
IOS	Input Output System
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IPFIX	Internet Protocol Flow Information eXport
ML-CCBF	MultiLayer Compressed Counting Bloom Filter
NBT	Naive Blooming Tree
NIDS	Network Based Intrusion Detection System
OBT	Optimised Blooming Tree
OS	Operating System
PC	Personal Computer
PPC	Privacy Preserving Controller
PDP	Policy Decision Point
PDT	Permitted Data Type
PEP	Policy Enforcement Point
PRISM	PRivacy-aware Secure Monitoring
QoS	Quality of Service
RTP	Real-Time-Protocol
RTT	Round Trip Time
SIP	Session Initiation Protocol
SLA	Service License Agreement
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SPIT	Spam over Internet Telephony
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UDR	Usage Data Record
VoIP	Voice over Internet Protocol

WP	Workpackage
XML	Extensible Markup Language

List of Figures

1 Introduction

Network operators currently use a wide range of popular applications to cover their specific network monitoring needs. Ensuring the acceptance of a privacy-aware network monitoring system requires proper integration with these applications and/or implementation of their interfaces.

The monitoring applications as surveyed and described in D3.2.1 range from very specific special-purpose monitoring tools to very generic frameworks. An example of the former type is PasTmon, which focuses solely on measuring application response times; while on the other hand TSTAT produces widely varied kinds of performance statistics, as well as providing detection of the traffic patterns generated by specific end-user applications. The potential privacy risk of these general-purpose applications indicates the need for finer-grained access control, from per-application access to all traffic data to specific, per-purpose access control. PRISM provides this control.

As specified in D2.2.1, the PRISM architecture allows the flexible integration of monitoring applications to meet varying network monitoring requirements. They can be split and distributed among the front-end (FE) and the back-end (BE); components may also be external to the PRISM system itself. Depending on privacy and performance requirements, this distribution is evaluated for each different monitoring purpose and application as presented in this deliverable. The deployment of existing tools fed with only limited information, the result of data reduction and protection within the PRISM architecture, suffices for most monitoring purposes. For other purposes not possible with mere data reduction techniques, functionality and interfaces must be re-implemented within the PRISM system as a split of analysis functions between the front-end and the back-end. Therefore, here we describe the deployment of existing network monitoring applications with only limited loss of functionality, but improved privacy and performance, by combining them with the PRISM system.

The contents of the deliverable are structured as follows. Section 2 provides the general approach for the adaptation of the monitoring applications for the PRISM system. It describes the two-stage monitoring processing design and its implications in terms of tailored output of the first stage processing to the specific needs and purpose of the monitoring applications, and the consequent possibility to provide a controlled, minimisedd, and privacy-safe output devised to guarantee that the back-end monitoring application will receive only the absolutely necessary data, thus technically enforcing the proportionality principle behind privacy preservation. Section 3 details significant technical advances in front-end processing, describing various possible deployments of Bloom filters and their extensions, and thus shows that significant front-end analysis work can be provided even on devices which must operate in real time with data captured off the wire and with severe memory and state representation constraints. Section 4 covers several components which are required for the back-end implementation and defines a basic workflow specification language for the data transformation. Section 5 presents several example scenarios from the contexts of IDS, billing non-repudiation, application detection and trace publication. The deliverable is concluded by Section 6.

2 Adaptation of monitoring applications

2.1 Motivation

Every monitoring application is used for one or more monitoring purposes. This can range from detection and forensic examination of a complex network intrusion event, to the shaping of network traffic, to the measurement of traffic flows for future capacity planning, and so on. Each of these applications is in turn composed of a set of elementary monitoring tasks. In the case of network intrusion detection, for example, these tasks may include the detection of scanning behaviour indicative of reconnaissance or the matching of a specific part of a packet against a pattern in an intrusion detection signature. Likewise, in the case of traffic shaping, tasks may include generating a baseline traffic profile and measuring flow or flow set volumes against that profile.

In most cases, these elementary tasks are centrally executed by what we call a "monolithic" monitoring application. These applications follow a traditional "gather first, process later" network monitoring paradigm, where data are collected by traffic capture devices which are typically agnostic of the monitoring application itself. All the monitoring processing intelligence is delegated to the monitoring application's operation, which is fed with the captured data and is devised to extract, from such data, information which is relevant for the specific monitoring purpose.

This traditional approach to network monitoring brings about two fundamental and intrinsic flaws.

On one hand, current techniques work at the expense of the users' privacy, by allowing the monitoring activity to pry into each packet's internals. Privacy is acknowledged by European legislation as a fundamental right of the individual [EU00, EU95]. Although anonymisation mechanisms can alleviate the issue, they cannot effectively support a usage model that balances *privacy* and *utility*: they can either offer very good privacy guarantees (e.g. robust to modern traffic analysis and classification techniques, indeed extremely powerful in extracting potentially sensitive information from as little as basic and non-attributable flow statistics such is packet sizes and correlation of inter-arrival times - see e.g. [HIN02, BIS05, CRO06]), but producing in this case monitoring data that is practically useless, or they can provide good monitoring data, but at the price of ephemeral privacy protection. For instance, the reader is referred to the analysis of anonymisation mechanisms carried out in [D3.1.2], which shows how commonly employed approaches (such as statically mapping IP addresses one-to-one to an anonymised identifier) are vulnerable to de-anonymisation.

On the other hand, the traffic volumes in Internet, like many other things, tend to obey exponential growth [ELD99, COF01]. In particular, the link capacity doubles every 12 months [COF01], while the router switching capacity doubles only every 16 months (cf. [SUN02]), and the original Moore's law ([MOO65]) predicts that the number of transistors in integrated circuits doubles every two years. These observations strongly suggest that the computational burden a traffic monitoring system faces tends to increase faster than the processing capacity. Indeed, the sheer size of network traces is rapidly becoming a major obstacle to advances in network monitoring. This issue was loudly raised almost 10 years ago by one of the fathers of the Internet, Van Jacobson[1]: "*If we're keeping per-flow state, we have a scaling problem, and we'll be tracking millions of ants to track a few elephants*". Current approaches to data

[1] Van Jacobson, End-to-end Research meeting, June 2000; quote reported in C. Estan, G. Varghese, "New Directions in Traffic Measurement and Accounting", SIGCOMM 2002.

reduction face yet another conflict, opposing *trace size* to *utility*: the more effective the data reduction mechanism, the less successful the monitoring activity.

We believe that these two flaws cannot be thoroughly addressed if we remain stuck to the vision of data protection and data reduction mechanisms as static, one-for-all, mechanisms applied in an on-off fashion regardless of the specific intent behind the monitoring process. The point is that the data "necessary" for a given monitoring application does strongly depend on the goal of the monitoring task itself. Many network security applications, for example, could eliminate known-good or probably-good traffic from the set of flows subject to greater scrutiny, and perform subsequent analysis only on the latter. Similarly, performance monitoring applications might significantly restrict the analysis of the data to summarised and strongly anonymised header-related statistics (the type and amount of such an information and anonymisation mechanism being widely dependent on the considered performance monitoring application), data which would instead be completely useless for network security purposes.

2.2 Two-stage monitoring system design

PRISM proposes to face the above discussed dichotomies through the rethink of monitoring applications as composed of two coupled stages (see a wider discussion in the architecture deliverable D2.2.1). The idea is in principle very simple. Monitoring applications should be split in two parts:

1) A first, front-end, part, is devised to collect, filter, and pre-process only the data strictly necessary for performing a specific monitoring task;
2) A second, back-end, part, is devised to perform the actual monitoring task, but its operation is restricted to process such a subset of pre-filtered and/or transformed data.

Goal of the specific processing performed by the front-end monitoring stage is to tailor the output to each specific monitoring application running at the back-end. In this operation, the front-end monitoring stage should be designed to provide a controlled, minimisedd, and privacy-safe output specifically tailored to the needs of a monitoring application running at the back-end, and hence guarantee that the back-end monitoring application will receive only the absolutely necessary data, thus technically enforcing the proportionality principle behind privacy preservation.

Such a coupled two-stage approach to measurement, consisting in the split of the monitoring activity between a front-end and a back-end in a tightly integrated fashion, promises to simultaneously address these two fundamental privacy and scalability issues. The pre-processing done at the front-end stage may significantly reduce the amount of information to be delivered to the back-end stage; in the same time the selection of the specific analysis tasks to be performed on the front-end stage is not done once for all. Rather, it is customised for the specific back-end monitoring application considered, so as to ensure that this reduction causes the proper data to be retained for further processing and the applied protection mechanisms do not cause a reduced utility of the monitoring operation (the reader is referred to the deliverable D2.2.1 for a description of the underlying authorization framework necessary to enforce the above described operation).

In summary, the front-end part of the monitoring application is responsible for three classes of operation on the observed traffic:

- isolation of relevant flows from the set of all traffic flows;
- extraction of relevant information from the observed traffic and elimination of irrelevant information.

- protection of remaining information (possibly in a time-varying manner in dependence of the outcome of the front-end monitoring process – see related discussion and proposed solutions in the deliverable D3.1.2) to prevent the processing of the remaining data in a way inconsistent with the privacy of the end-users.

Note that this approach requires applications to be composed of analyses; each analysis is in turn split into cooperating front-end and back-end analysis functions.

2.3 Where and how to partition monitoring applications

The discussion above presumes that the front-end and the back-end part of a monitoring application coincide directly with the front-end and back-end *components* of the PRISM architecture.

Indeed, this is the most challenging issue from a technical point of view, and the feasibility of moving analysis to the front-end edge will be object of thorough investigation in the next section. There we investigate the limits of processing capabilities on the front-end, which is resource-constrained and must process packets as they arrive at wire speed. We show that approaches based on lightweight hash-based data structures, such as Bloom filters and elaborations there on, can perform non trivial analysis functions (such as a variation detection primitive, applicable to scan detection, see section 3.3.2).

Partition also affects system operation at the back-end. Front-loading analysis has the effect of easing privacy preservation as well as scalability improvements due to significant data reduction. The back-end components of a partitioned application are responsible for verifying that the necessary information for the monitoring application can be passed on to the monitoring application without any threat of end-user privacy violation, and for further processing the information through back-end analysis functions devised to provide a privacy-safe output as necessary. Note that these back-end analysis functions can be made extremely effective as they may access sensible data, or logged information which, if disclosed, would result in a potential privacy violation. But privacy preservation can also be achieved by providing tight control over the data passed on to an external monitoring application.

In other words, we address the above mentioned dichotomy by reducing data as early as possible and preserving the ability to operate over critical data following the principle of least access. Back-end analysis functions implemented as “embedded processing components” mediate access to potentially sensitive data. Their operation is grounded on the concept of “data transformations”, in the sense that the back-end receives data from the front-end and releases data to the monitoring application; its job is to generate the final data set from the former. This data transformations operation is tightly related to the privacy-aware access control mechanism of PRISM. In fact, access control and data transformations are specified in an integrated way, since access control in PRISM as described in [D3.1.2] described the necessary internal processing steps for the production of those data that are to be delivered to the monitoring application based on parameters such as the role of the requesting entity and the underlying purpose.

Subsequently to a monitoring application data request, the back-end executes a sequence of analysis functions that will provide the application with the necessary and proportional data. The sequence of analysis functions called by the back-end is not static, but rather dependant on the “privacy context” of the particular request. Its static part, (i.e., steps that are executed in any case) are specified by the PPC in an “offline” manner and provided to the back-end by

means of attribute certificates, while the dynamic part of the analysis functions execution is determined online by the back-end itself which evaluates the underlying conditions. The result is a formally defined sequence of executions, which takes the form of a Data Transformation Workflow Specification Language. This constitutes a proprietary workflow language for the formal specification and synchronisation of the back-end analysis functions.

2.4 Actual back-end monitoring application's adaptation

The acceptance of PRISM will strongly depend on the possibility to use existing legacy monitoring applications. It is a central requirement that PRISM is able to work with such widespread used applications. For sure developing a monitoring application from scratch taking into account privacy by using the PRISM architecture will lead to a flexible and comfortable application. A detailed example of such application can be found in Section 5.1.2. When designing the PRISM system it was also taken into account to operate legacy monitoring applications in a privacy preserving environment. The adaptation of the applications should be reduced to a minimum. Especially the end user interface should be untouched, because operators are used to it, and they are often perfectly suited for the needs of the operator. The idea to operate such application in a privacy preserving environment is simply to reduce their input to an amount where no privacy sensible information is present. The PRISM system can be used reduce, adapt and anonymise the content of packets fields. Such operations for example could be:

- Forward only packets of a specific source IP
- Anonymise the IP addresses
- Delete the payload and generate a random one
- Random inter packet arrival time
- Forward only packets where content matches a pre-defined string

The back-end will transform the information from the packets together with random values to the input format of the legacy application. Consequently there is no need for any coding within the application itself. PRISM thus allows that the legacy application can generate the results for specific monitoring purpose and ensures user privacy by processing the information sent to the application. This ensures high acceptance of the PRISM system as the needed adaptation work for an application will be reduced to a minimum.

An example can be found in Section 5.3.2 where the Skype detection engine of TSTAT is brought to a privacy preserving environment. Nowadays this application operates on the full packet stream where privacy information is present. By the usage of the PRISM system the application will only receive payload encrypted packets where the IP addresses are anonymised. Consequently no privacy information is given to the application.

In an unlikely case where it is necessary that the full packet stream is analysed for a specific monitoring purpose the application has to be brought into the PRISM system. An example therefore can be found in Section 5.1.1.

3 Partitioning functionality between FE and BE

The front-end as described in section 2.1 is devised not only to capture packets but also to provide effective and scalable means to perform packet analysis, flow isolation, extraction and flow protection functions.

Goal of this section it to address techniques and algorithms that will be implemented in the front-end in order to both i) reduce the information necessary for a monitoring application from a potentially very large volume of data, and ii) provide processing functionalities which can be used as input and guidance for per-flow cryptographic protection functions.

Concerning issue (i), we recall that reducing the amount of data to process and focusing the data processing on what is really meaningful is in itself a fundamental requirement before considering privacy issues, solely in terms of performance and scalability issues. Privacy requirements merely strengthen the need to focus data collection and processing activities on a subset of the observable data, namely that which is strictly necessary to perform a specific monitoring task.

Concerning issue (ii), we recall that, in the frame of the separate work-package WP3.1 dedicated to data protection mechanisms, we have proposed (see section 3.1 of the deliverable D3.1.2 [D3.1.2]) a novel approach based on the idea to apply cryptographic data protection mechanisms which **do depend on monitoring state information, this being in turns determined through the operation of front-end traffic analysis functions.**

In this section we discuss how to implement effective and non trivial on-the-fly packet and traffic analysis functions over the front-end data capture device, through the usage of ultra-fast, lightweight and memory-efficient data structures. The rest of this section is organised as follows. In section 3.1 we present the problems of implementing monitoring activities in the front-end together with a short review of Bloom filter data structure [BLO70], since i) they represent a promising approach for the implementation of packet processing and inspection functions directly in the front-end and ii) most of the novel analysis functions described next are based on extensions of Bloom Filters and Counting Bloom Filters. In Section 3.2 we address the issue on how to technically improve the design of Bloom filters, and therefore also some related extensions, to achieve greater efficiency and memory preservation. Three new designs (Multi-Layer Compressed Counting Bloom Filters, Blooming Trees, and Optimised Blooming Trees) are proposed. Finally in section 3.3 we present innovative analysis functions which take advantage of Bloom-based filtering techniques, and which show that non trivial analysis tasks, including deep packet inspection, rate monitoring, variation detection and continuous time scan detection, can be effectively deployed over front-end devices.

3.1 Front-end processing techniques

The development of a front-end stage that plays an active role in the overall monitoring process increases the burden on the front-end system, and therefore requires the adoption of a novel design leveraging all the capabilities of the available hardware. Indeed, in order to accomplish operations like capturing, classification, anomaly detection, flow discrimination and isolation *at wire speed*, a detailed knowledge of the hardware capabilities/bottlenecks, as well as the fine grained analysis of the available time budget for each micro-operation

involved are required. Here we explore the applicability of one particular class of dedicated hardware, a *network processor* such as the Intel IXP2350 [2].

Since most front-end analysis functions do comparatively simple operations on comparatively large amounts of data (potentially, every octet of every packet observed), we must consider the impact of memory latency on performance. Indeed, the processing unit may spend more time waiting for memory transactions to be completed than actually processing the data. Even if such latencies can be partially hidden by multithreading, they can still significantly limit maximum performance. In particular, in programmable devices, the amount of memory is hierarchically divided into four categories:

- very small but very fast local memory;
- small and fast on-chip cache (SRAM);
- larger and slower off-chip cache (SRAM);
- very large and very slow off-chip system memory (DRAM).

Time-critical operations must then in most cases involve the use of the first two categories of memories only; unfortunately, they turn out to be the most expensive and very limited in size. To give an idea of the different performance of memories, one may consider that in an IXP2350, a single read/write operation in local memory takes 2 clock cycles while an access to the large and inexpensive off-chip DRAM takes 180 clock cycles.

The attempt to devise effective solutions to be integrated into a high-performance front-end stage must then pursue the investigation of *stateless* and *memory saving* approaches in that they tightly reflect into faster operations. In this scenario, a very promising approach towards packet processing and inspection is based on Bloom filters [BLO70] and their variations. Bloom filters are compact and fast data structures for approximated set--membership query and their popularity is rapidly increasing because of their very limited memory requirements (roughly speaking, they implement the principle of `"*trading certainty for time/space*".[3]). A Bloom filter represents a set of n elements by using a bitmap of m elements. Each element of the set is mapped to k elements of the bitmap whose position is given by the result of k hash functions. To check whether an element belongs to the set one just needs to evaluate the k hash functions and verify if the corresponding bits of the bitmap are all set. Naturally, the filter allows for false positives, in that the hash functions of different elements may collide. Nevertheless, a proper choice of both the length of the bitmap and the number of hash functions minimiseds the probability of false negative. However, Bloom filters may not be used for sets whose elements may change over time, since elements cannot be properly deleted from the bitmap.

A Bloom filter may be extended by replacing the bits of the bitmap with multibit counters (or bins), resulting in a counting Bloom filter. This extension allows straightforward insertion and deletion of elements is by incrementing or decrementing the value of the counter.
The use of counting Bloom filters for statistical data processing is extremely flexible, although the fixed size of bins unnecessarily wastes memory in many cases. A significant improvement can be obtained by allowing dynamic bin sizing, bin compression, and multi-layering. These modifications appear promising for their actual application to the data processing performed by the PRISM front-end. For example, consider a set of rules to be checked by the front-end classifier: a counting Bloom filter can easily be used to represent the set. In order to verify whether a packet obeys one of the rules of the set, a simple lookup operation consists of evaluating k hash functions and checking if the values of bins are all non-zeros. If the result is

[2] see http://www.intel.com/design/network/products/npfamily/ixp2350.htm
[3] G. Varghese, *Network Algorithmics*, 2005, Morgan Kaufmann

positive one may deduce that with a small error probability the packet actually satisfies the rule and can be delivered to the second stage (back-end) for further analysis.

Several other methods and tool for fast and stateless analysis of high rate traffic have been proposed in the literature. In particular, one of the most deeply investigated problems is the approximate evaluation of traffic volume associated with the largest flows on a given link (the so-called *elephants* or *heavy hitters*). This involves accounting for packets belonging to such flows only, while discarding information associated to the other packets. To this end, one of the most popular solutions is sampled NetFlow, which simply enhances Cisco's NetFlow measurement framework by sampling packets at a rate that can be defined by the network administrator, thus significantly reducing the amount of data to be processed. Of course, flow sizes are estimated, and small flows can pass undetected using this approach.

Several enhancements to Sampled NetFlow have been proposed, such as the so-called Adaptive NetFlow scheme [VAR04] and the multistage filter [VAR02] scheme. Adaptive NetFlow overcomes many of the limitations of Sampled NetFlow, such as the impossibility of effectively counting the total number of flows. On the other hand, multistage filtering takes advantage of a Bloom filter-like data structure, composed by an array of counters selected based on a set of hash functions evaluated over each packet. When all of the counters associated to a given flow are over a selected threshold, the flow is classified as "large" and a record for it is created. A different architecture for counting the number of packets per flow has been proposed by Ramabhadran and Varghese [RAM03]. This scheme based on two layers of counters: a first layer kept in a fast cache and updated for each packet received, and a second larger layer stored in DRAM and periodically updated based on the content of the first layer.

One of the most difficult problems in traffic accounting is the necessity of keeping a very large set of counters in a large and therefore comparatively slow memory block, thus significantly affecting the performance of the whole system. The general approach to addressing this issue is to reduce the memory occupation of the set of counters. For example, counter braids [MON08] exploit the redundancy associated with the most significant bits of the counters, while space-code Bloom filters [WAN04] implement a multiple-resolution probabilistic counting scheme.

3.2 Technical advances in front-end processing

While Bloom filters provide a stateless and quickly accessible data structure for traffic analysis and classification that can profitably be implemented within the FE stage, in the presence of sets containing a very large number of items, their dimension can exceed that of the fast memory caches provided by the network processing architecture; as a consequence, such data structures should be moved into slower memory blocks, thus significantly reducing the achievable performance. In order to deal with this problem, Bloom filter compression schemes which properly exploit a multi-stage memory hierarchy are needed.

3.2.1 Multi-Layer Compressed Counting Bloom-Filters

A multi-layer compressed counting Bloom filter (ML-CCBF) [FIC08a] is a counting Bloom filter that reduces the memory requirements and the complexity of lookup. The basic idea is to explode the counting Bloom filter along another dimension, hence creating a multilayer structure, and to use Huffman coding (where a number σ is encoded by σ consecutive ones and a trailing zero), which is the minimum redundancy coding for independent and small symbols such as the bins of a counting Bloom filter. The construction of an ML-CCBF

provides a stack of bitmaps ($L_0 \ldots L_N$), where the first layer L_0 is a standard Bloom filter. The other layers are built and modified dynamically as needed. The layer L_i contains the i-th binary digit of each of the Huffman encoded counters.

Let *popcount*(u) be the number of 1s in the bitmap (0...u-1); the j-th bit of layer L_i belongs to the counter whose *popcount* on L_{i-1} is equal to j. Due to such properties, the set membership lookup can be performed by simply examining the first layer, which can be kept in fast memory. Figure 1 shows an example of a ML-CCBF. In the example, we are counting a bin c for symbol σ. The bin at layer 0 is pointed by the hash function h(σ). The number of ones before h(σ) is computed (i.e. *popcount*(*h(σ)*) = 5) and used as index for layer 1. The procedure is repeated until we find a ”0” digit (that is the end of the code). Therefore the resulting Huffman code for the counter c is 1110, which corresponds to the value 3.

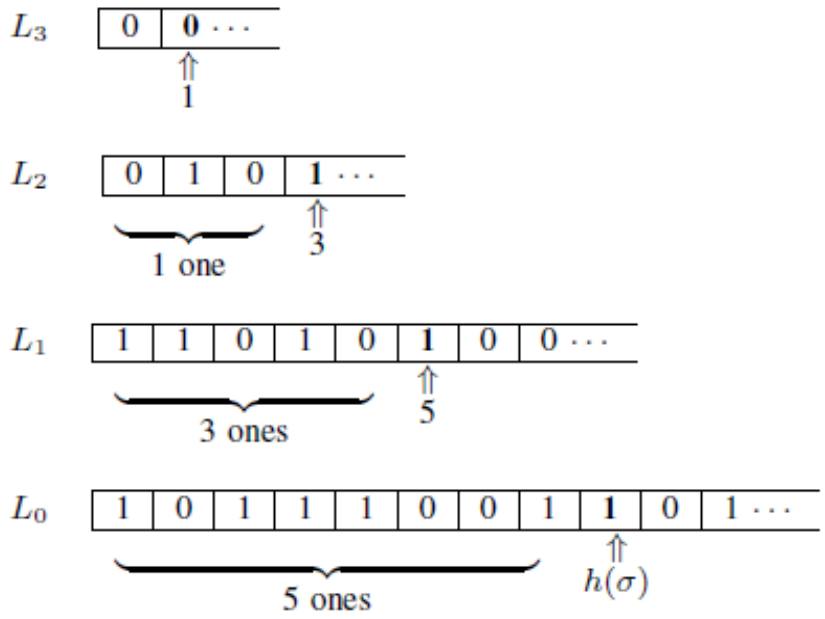

Figure 1: Example of the basic mechanism of MLCCBF

Complexity and properties

One of the most significant advantages of our algorithm is that it is an extension of a standard Bloom filter. Thus, the lookup is as simple and fast as in any Bloom filter, as we need to check only bits at layer 0. Therefore, the lookup complexity is O(1). Instead, for insertion and deletion we need to explore different layers in the structure. We refer to mi as the number of bits in layer i. The size of layer i can be obtained as:

$$m_i = m_0 P(c \geq i)$$

Since jumping one layer up requires a *popcount* on a potentially large number of bits, we divide all layers in blocks of the same bit-size D and add a table for each level. When computing *popcount*(u_j) at layer j, the first $\log2(m_j/D)$ bits of u_j are used as index to table j. Each entry of the table represents the number of ones preceding the start of the block. Thus, if W is the number of bits in a word, the actual *popcount* operation works only on less than D/W words. Therefore, the average cost of a *popcount* is 1 + D/(2W).

Both insertion and deletion procedures in an ML-CCBF require, for all k bins, the complete lookup of multiplicity (by exploring a certain amount of layers), a shift by one position and the update of the last explored table. Such an update consists simply of an increment or a decrement on a limited number of entries. Therefore the average amount of operations for insertion and deletion is given by:

$$\omega = k\{E[c](1 + D/(2W) + 2)$$

Once again, E[c] ≈ln2, thus the average sum of operations is fixed and the complexity for insertion/deletion is O(1).

Size

The use of an ML-CCBF results in a lower memory requirement:

$$S = m_0 + \sum_{i=1}^{m_0} \varphi_i + \sum_{i=1}^{n_{tab}} TS_i$$

where TS_i is the size of the table required for layer i, which requires $n_i = m_i/D$ entries of size $\log2(m_i)$, thus resulting in:

$$TS_i = n_i \log_2(m_i) = \left\lceil \frac{m_0}{D} \right\rceil P(\varphi \geq i) \log_2[m_0 P(\varphi \geq i)]$$

The average amount of required memory is therefore:

$$E[S] = m_0(1 + E[\varphi]) + TS$$

Figure 2 shows the comparison among the sizes of an ML-CCBF, a standard counting Bloom filter, and the minimum amount of bits for independent symbols (BF entropy = m*entropy), for k = 10 and m = 32768. The memory saving of our method is clear as it approaches the minimum value. Note that the optimal number of elements n = 2270 that minimises f, minimises the distance from the BF entropy as well.

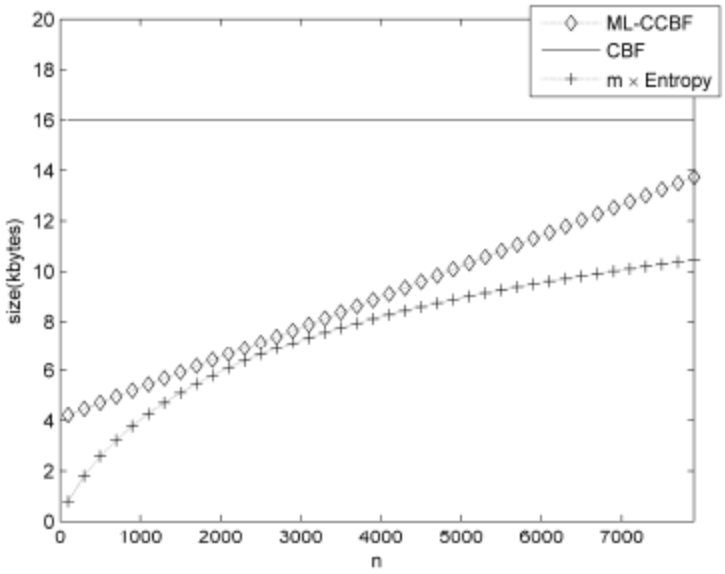

*Figure 2: Size comparison among ML-CCBF, CBF and m*Entropy*

3.2.2 Blooming trees

Blooming Trees are novel data structures similar to counting Bloom filters which allow the tuning of the false positive and overflow probabilities. The structure is allocated in different layers, thus exploiting the built-in memory hierarchy of many packet processing systems. The idea of Blooming Trees [FIC08b] is constructing a binary tree upon each element of a plain Bloom Filter, thus creating a multilayered structure where each layer represents a different depth-level of tree nodes. The aim is to achieve both low false positive probability and low memory requirements, for increased time for lookup. The latter can be mitigated by the low memory consumption enabling the deployment of the structure in faster on-chip memories.

To build a naive Blooming Tree (NBT) for n elements, L+2 layers are defined:

- a plain Bloom Filter (B_0) with k_0 hash functions hj ($j = 1 \ldots k_0$) and m bins such that $m=nk_0=\ln 2$;
- L layers ($B_1 \ldots B_L$), each composed by m_i ($i = 1 \ldots L$) blocks of 2^b bits;
- a final layer (B_{L+1}) composed by c-bits counters.

The j-th hash function hj provides a $\log_2 m + L$ b bit long output: the first group ($s_{0;j}$) of $\log_2 m$ bits is used to address the BF at layer 0, the other Lb bits are divided into L substrings ($s_{1;j} \ldots s_{L;j}$) of b bits, one for each layer.
Let *popcount*(B[u]) be the number of ones in the bitmap B[0]…B[u-1] and let us consider the simplest case b = 1 (this way, blocks become couples and substrings $s_{i;j}$ collapse into single bits). The lookup for an element σ consists of a check on k_0 elements in the BF and an exploration of the corresponding k_0 "branches" of the Blooming Tree. We jump from layer i to layer i + 1 by:

- computing a *popcount* on layer i, that gives us the index of the couple to be observed in the layer i + 1;
- checking the bit expressed by $s_{i;j}$: if $s_{i;j}$ is equal to 0, we check the first bit of the couple, otherwise the second;
- processing the bit of the couple: if it is 0, then σ is not in the set and the lookup result is NOT FOUND, otherwise the overall process must be iterated for the next layers.

An example (with $k_0 = 1$) of the lookup process is shown in Figure 3, where the tree structure of NBT is clear. For instance, let us observe the last bit of the BF, where two items collide.

The popcount (equal to 3) leads to the proper block of layer B_1. The bit s_1 of the hash is equal to 0 for both the items, so the first bit of couple is set. Then, the items present a different s_2 bit of the hash and they split: in the fourth couple of layer B_2 (as indicated by the *popcount* on B_1) both the bits are set. Therefore two different bins in B_3 count the two items.

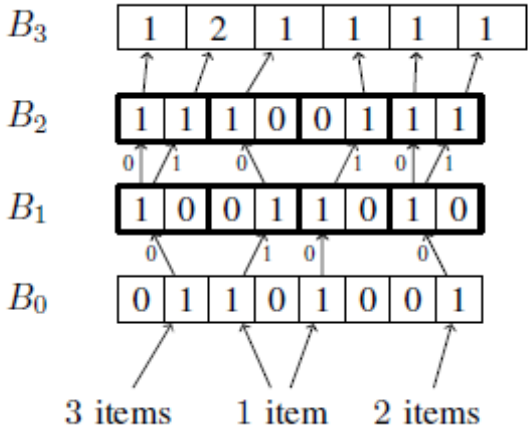

Figure 3: An example of a Naive Blooming Tree with b = 1.

3.2.3 Optimised Blooming Tree (OBT)

An optimised version of a Blooming Tree can be constructed following three observations about naive Blooming Trees:

- once there are no collisions in a certain block at layer u, there are no collisions also in the corresponding blocks in all upper layers u+1…L, but we use $2^b(L-u) + c$ bits for those blocks;
- all blocks always have at least a bit set: a block with 2^b zeros (let us call it zero-block) has no meaning;
- looking up w layers yields $f = 2^{-(k0+wb)}$.

Therefore, whenever there are no collisions in a block, a zero-block can be used to indicate this situation and stop the "branch" from growing. But we cannot stop the lookup there, since it would increase the probability of a false positive. The solution of the Optimised Blooming Tree (OBT) is to add a bitmap and an array of hash substrings for each layer. The array of substrings for layer i is composed by all the [(L-i)b]-long hash substrings that complete the hash of the "branches" that stop at layer i. In the bitmap (of m_i bits), the generic j-th bit is set if the j-th block has no collision (i.e. zero-block); this way it can be used to address the substring array (see Figure 4). The optimization can be also done at runtime.

Obviously, operational routines change. As for lookup of an element σ, whenever the x_i-th block is a zero-block, we compute y_i = *popcount*(bitmapi [x_i]) and compare the last (L-i)b bits of the hash of σ with the yi-th element in the i-th substring array. This way, the lookup becomes faster as zero-blocks are very likely to occur in any layer, thus avoiding all the steps required to jump up to layer L+1.

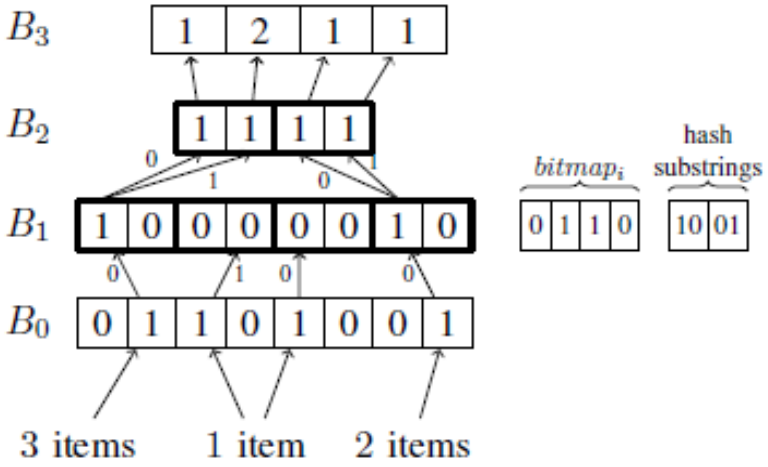

Figure 4: An example of an Optimised Blooming Tree with b = 1.

The insertion routine, however, can be slightly slower since it must also be aware of zero-blocks. If we have no collisions at layer 0, we add a zero-block, we set the corresponding bit in the bitmap as well as the corresponding substring in the substring array. Instead, if there is a collision, we have to check the colliding elements and create the corresponding branches up to the layer (let us say j) where the hash substrings differ. At layer j+1 we repeat the ordinary steps: add two zero-blocks, set the corresponding bits in the j-th bitmap and add the two hash substrings in the j-th substring array. The computational cost of deletion, in turn, is about the same of that of insertion since, again, zero-blocks require additional processing but reduce the amount of accesses to upper layer.

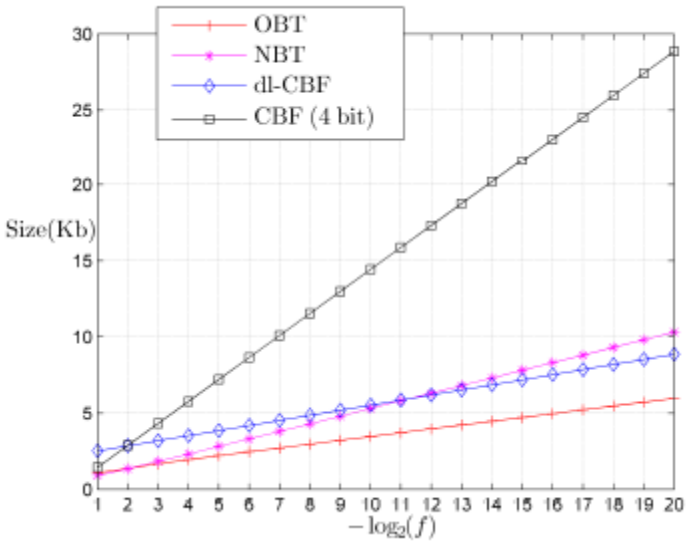

Figure 5: Size comparison for NBT, OBT, dlCBF and CBF with n = 2048.

Figure 5 shows that the overall gain in size is evident for optimised Blooming as compared to NBT and dlCBF. The d-left CBFs (dlCBFs) [BON06] are simple alternatives based on d-left hashing and fingerprints of bins. They do not rely on the principles of Bloom Filters, but they offer the same functionalities. The dlCBFs use less space, generally saving a factor of two or more for the same fraction of false positives, and the construction is very simple and practical,

much like the original Bloom Filter construction. Indeed the simplicity in constructing and maintaining data structures is maybe the greatest contribution of [BON06] as compared to previous works. Moreover, even dlCBFs have the limitation of potential counters overflow and the need for an additional fingerprint for each bin in the data structure. Figure 5 reports the size for the above mentioned structures as a function of -log2(f) (where f is the probability of false positives) for a number of elements n = 2048.

3.3 Functional advances in front-end processing

3.3.1 Bloom filters for avoiding Intrusion Detection evasion

Standard pattern matching techniques for packet inspection and network security can be evaded through TCP and IP fragmentation. Therefore, to detect such attacks, classical Intrusion Detection Systems must reassembly all packets in a flow before applying matching algorithms; keeping reassembly state for each active connection, in turn, requires a huge amount of memory and may be unfeasible for a large number of flows at high packet rate [VAR08].
The use of counting Bloom filters provides an effective solution to this issue by allowing the fast detection - at the wire speed - of evasions attempts without any need for packet reassembly. The rationale of such an approach relies on the capability of counting Bloom filters to quickly update string sets and to deal with partial signatures, as well as to effectively count the occurrences of elements.

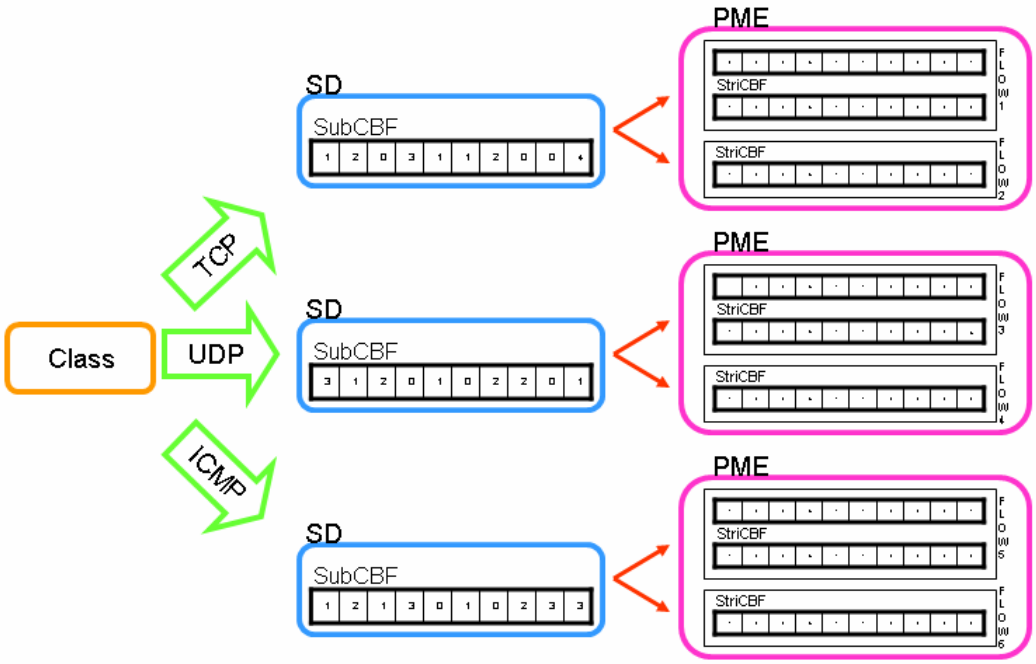

Figure 6: The anti-evasion system (SubCBF = Substring CBF, StriCBF = String CBF)

This approach splits the strings to be searched for into 3-byte substrings and creates a counting Bloom filter (hereafter referred to as a Substring CBF) to represent this set [ANT09]. Proper heuristics are used in order to account for strings which are shorter than 3 bytes. Notice that this set is easily updatable by inserting and deleting elements with no need to rebuild the overall structure. Whenever a substring is detected through the Substring CBF, a bank of further filters (hereafter referred to as String CBFs) is properly set for the specific flow: more precisely, for each string the detected substring belongs to, a filter, which accounts for the set of the remaining characters of the string, is initialised. All the packets of the flow are then

processed in search of the remaining part of the string and, whenever a match is found, the values of the corresponding String CBFs are decremented.
Whenever a String CBF is completely reset to zero, the attack is detected and the flow is either blocked or rerouted towards the back-end stage of the monitoring system. The complete picture of the system is shown in Figure 6.
Notice that the specific structure of the system itself allows either a full implementation within the front-end or a layered implementation where Substring CBFs are located in the front-end and the String CBFs in the back-end. Finally, it is worth noticing that, if implemented in the back-end, the String CBF stage might even be replaced by any other traditional pattern matching systems that operate packet reassembly.

3.3.2 Continuous time scan detection and rate control

As discussed in section 2.1, the issue of significantly reducing the amount of data to be delivered to a back-end monitoring application is a fundamental performance and scalability requirement. Privacy further enters into play when we consider that i) data reduction is by itself a form of privacy protection, and that ii) this becomes even more true when the data reduction is performed on the basis of the actual purpose of the monitoring task.

Indeed, we recall (refer to deliverable D2.1.1, section 4, for a comprehensive discussion of the underlying legal and regulatory framework) that a foundational principle behind data protection consists in the fact that the personal data that are processed must be *adequate*, *relevant* and *not excessive* in relation to the purposes for which they are collected and/or further processed. Therefore, if we can implement a technical approach which allows isolating the traffic data of monitoring interest from all observable traffic, we would technically enforce (!) said foundational regulatory principle.

The techniques described in what follows provide concrete examples on how to do this. In particular, based on innovative usage of the Bloom Filter, we have designed two mechanisms that allow front-end to perform rate measurements, to identify variations and/or repetitions of the values of fields of every protocol header. A proper combination of these two techniques is sufficient to identify the flows of interest for a large set of monitoring applications (Performance measurement, high rate flows billing, etc.).

Here we provide a complete description of the mentioned mechanisms in the frame of a specific application scenario, namely how they can be combined to isolate scanning hosts that may require further inspection by a monitoring application. The isolation of scanning behaviour is useful to identify propagating worms or attackers trying to achieve information regarding possible victims. Worm propagation processes [MOO03], in fact, scan hosts in finding targets to spread to, and many attacks are preceded by host or port scans as a reconnaissance phase. Furthermore for many monitoring applications such as content-based network intrusion detection scan traffic is considered unimportant; for these applications, separating scan traffic from non-scan traffic can improve performance by reducing irrelevant traffic to be measured. Scanning behaviour is characterised by a high degree of variation with respect to a specific parameter or set of parameters for a given network traffic flow within a given time window. For example, to detect a horizontal TCP SYN scan of a given subnet, a behavioural scanning detector would look for variation in destination IP addresses for each given source IP address. In practice for a given reference flow one (or more) specific field of a packet (e.g. the destination IP address in the previous example), is checked to determine whether it is new within the scope of the flow.

3.3.2.1 Basic scanning detection approach and limits

In prior work [BIS05] proposing a solution for fast anomaly detection based on Bloom Filter, the detector is organised into two modules as illustrated in Figure 7.

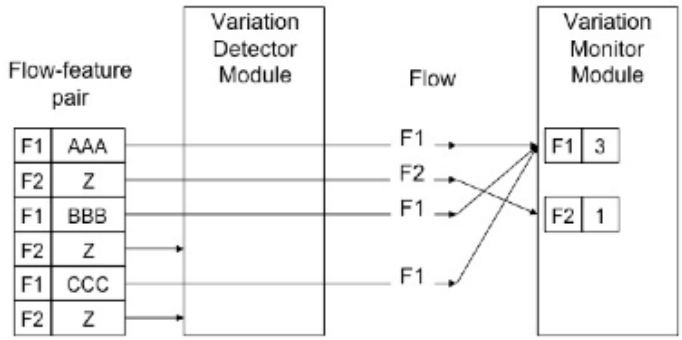

Figure 7: Fast anomaly detection Bloom Filter

The variation detector assigns each packet in the full traffic stream to a specific flow, and determines whether the specific field under monitoring has already been seen. If the flow-field pair is new, the variation monitor module notes that a new variation has occurred for the specific flow and counts it as such. The variation monitor may then signal when the variation count for any given flow overflows a predetermined threshold.

Each of these modules is implemented through an efficient data structure based upon a Bloom filter. The variation detector is an ordinary Bloom filter storing the set of flow-field pairs. The variation monitor is built around a conservative-update [ES02] counting Bloom Filter which collectively counts all possible flows, avoiding the need to track variation counts internally for each individual flow. The presented solution is limited in its simple treatment of time. Time plays a fundamental role for two reasons, and these two reasons lead to the innovative mechanisms that we propose to enhance the front-end capacity of isolating flows. The first one concerns the state kept by the variation detector. Without some mechanism periodically refreshing its filter, this module would begin with no state and gradually fill with flow-field pairs. As this happens, the effectiveness of the filter decreases as the false-positive probability increases. This opens a potential avenue of attack against the detector; an attacker could blind the module by filling up its filter with spurious flow-feature pairs before running an actual scan, which may not be detected due to interference from the high false-positive rate. We remark that the module filling and the consequent increase of the false positive rate is a process linked only with the dimension of the Bloom Filter m and with the number of elements to be stored n, and is common to all the types of Bloom Filter (i.e. Counting Bloom Filter, Spectral Bloom Filter, etc.). The second one impacts the thresholds used by the variation monitor. Given the bursty nature of network traffic in general and scan behaviour in particular, one single time window for this module is insufficient; long time windows will cause many false positives in the case of a burst in traffic, and furthermore will delay reporting of scan behaviour; short time windows will lead to false negatives as lower-frequency scan behaviour goes undetected. Therefore, an adaptive time window, which responds to the profile of measured traffic, is necessary to minimise both false-positives and false-negatives. We remark that the problematic highlighted by the scan example affect all the possible applicative scenarios where the isolation of the flows interesting for a specific monitoring application, requires taking into account the time behaviour of a flow parameter.

In order to address these issues we present i) a self-tuning mechanism for detecting variation, independent from the number of elements (the flow-feature pairs) that have to be detected but that adapts the variation detector to the actual traffic load; ii) the application of a counting Bloom filter based implementation of a token bucket [SHE97], for rate measurements. We further described how the two stages can be coupled to obtain a mechanism to isolate directly in the front-end the flow performing scanning detection.

3.3.2.2 Definitions

Hereafter we refer, in the description of the proposed mechanisms, to the following definitions of flow, feature, and variation:

Definition of Flow. A Flow is a set of packets which have some common keys or characteristics. This is formally stated as follows: if p is a packet travelling over the link, there exists a function $f_i = f(p)$ which, for each packet p, extracts or computes a bit-string f_i , which we call a flow label. As such, it is obviously not restricted to the traditional IP 5-tuple (source/destination IP address, source/destination port, and protocol. This definition of flow is equivalent to that defined by the IETF for IP flow export using IPFIX [CLA08]. Example flows conforming to this definition include: all the traffic sent from a given IP subnetwork; all the traffic for which the source port is higher than a given threshold; or all the traffic crossing a defined border with a specific IP time-to-live.

Definition of Feature. A Feature is information useful within the scope of the detection application. In particular a feature could be the number of bytes exchanged within a time windows, the source IP address, the destination port, the protocol type, etc. For our scope we will consider only features that can be extracted from individual packets. Formally, a feature is a function $g_j = g(p)$ which associates, to each packet p, a bit-string g_j, which we call a feature label. For instance, a feature can be the destination IP address or the destination port, and the feature labels g_j are the actual contents of the considered field.

Definition of Variation. A Variation occurs when a packet in a given flow presents a "new" feature label, i.e. not previously seen, within a suitable time scale, in other packets belonging to the same flow. A feature of a packet belonging to a given flow is, in fact, extracted and its value is recorded for a fixed amount of time. A feature's value different from the one already recorded for a specific flow or not recorded at all is considered a variation. Obviously the amount of time a feature's value is recorded (i.e. the suitable time scale mentioned above) is of fundamental importance for the variation identification. If a feature's value is recorded for short time (less then a RTT of a packet) Note that the time scale may be both explicitly and precisely set as a monitoring time window T during which the traffic is analysed, or can implicitly emerge, as in our approach.

3.3.2.3 Variation detector

The variation detector is composed of two Bloom filters operating in parallel. Each filter cyclically goes through two subsequent states, "Learning" and "Detecting", such that at any given time one of them is in the Learning state while the other one is in the Detecting state. At the end of the cycle, the old Learning filter becomes the new Detecting filter, and the old Detecting filter is reset to zero before becoming the new Learning filter, such that as the next cycle starts the Learning filter is always empty.

The Detecting filter determines whether a newly arriving flow-feature pair $<f_i, g_j>$ is a variation. It computes the k hash values, and matches them against the relevant bits stored in the array; if at least one bit is 0, it adds the flow-feature pair to the filter by setting all the

relevant bits of the array to 1, and advertises the flow f_i that shows a varying feature. Note that to use the described mechanisms to detect repeating flow-feature pair $<f_i, g_j>$ it is sufficient to compute the k hash values for every incoming pair, and matches them against the relevant bits stored in the array; if at least one bit is 0, the flow-feature pair is added to the filter by setting all the relevant bits of the array to 1, while if all the marched relevant bit stored in the array are set to 1, the flow label f_i is advertised.

The filter in the Learning state receives the same input $< f_i , g_j >$, and updates the filter by setting all the relevant bits to 1, with no further action. Separating the variation detector's responsibilities into two parallel Bloom filters achieves two goals. First, it allows the Detecting filter to "warm start" by being primed with flows during its time in the Learning state. Second, it allows "self-clocking" operation adaptive to the input traffic. Allowing the Detecting filter to warm start avoids a problem that would occur if a single Bloom filter were periodically reset: at each reset time, any flow-feature pair would be counted as a new variation, even if it is not. The variation monitor is self-clocking because the state transition between Learning and Detecting does not happen on a pre-specified clock; instead, it is dependent on the level of occupancy of the Learning filter, and therefore on the input traffic. Recall that, for a Bloom Filter of size m and with k hash functions, the probability that a generic bit in the array is equal to 0, given that n elements are stored in the filter is

$$\rho(n) = (1 - 1/m)^{kn} \approx e^{-nk/m}$$

Let now $B_0(n)$ be the average number of bits set to 0 given that n elements are stored in the filter:

$$B_0(n) = m\rho(n) \approx me^{-nk/m}$$

If we now consider a filter loaded with n/2 elements:

$$B_0(n/2) \approx me^{-\frac{nk}{2m}} = \sqrt{m^2 e^{-nk/m}} \approx \sqrt{mB_0(n)}$$

For a target n, the Bloom filter's false-positive probability is minimised if we set k = m/n log 2. This implies that, with an optimal k, B_0 (n) = m/2, and B_0 (n/2) = m/√2. The filters are dimensioned so that, when completely filled, they are expected to contain at most n distinct flow-feature pairs and to perform with a false- positive probability target ψ. The filter size m is then readily determined from the well-known relation ψ = 0.6185m/n, which holds for the optimal setting of the number k of hash functions. For instance a target 1% false-positive probability implies that m ≈ 10n.

A filter is kept in the Learning state until the number of 0s contained in the filter array is greater or equal than a given threshold α = m/√2. This implies that, at the end of the subsequent Detecting period, the same filter will be filled with a number of elements never significantly exceeding the target capacity for which the filter has been sized, and will operate with approximately a same number of 0/1 bits. Note that, unlike time-based systems (such as for instance [KON06]), the previous considerations hold irrespective of the traffic mix. Peaks of variations which abruptly load the variation detector are both detected by the Detection filter, and learned by the Learning filter, which implicitly acts as an estimator of the arrival traffic and a clock for the state transition of the variation detector). The usage of a fix threshold α = m/√2 to decide when we have to swap between the Learning and the Detecting filter implies that the Detecting filter will usually contain, a number of 0s lower than B_0 (n) = m/2. When the traffic is composed by a mix of new and repeating flows, in fact, the Learning

filter, since it is initially void, will see as variations also the pairs < f_i , g_j > belonging to flows already active and that are instead identified as repetition by the Detector filter. This implies that the Listening filter Lf will reach the threshold alpha when the Detector filter D_f will contain less then B_0 (n) = m/2. In order to improve the performance of the Detector filter it is possible to exploit a simple self tuning mechanism that shift the threshold alpha so that when the Listening filter become the new Detector filter, the old Detector filter contains B_0 (n) = m/2. This result can be achieved estimating the number of pairs < f_i , g_j > that the Listening filter see as variations while the Detector filter see as repeating pairs. Defining B_0 (D_f) as the number of 0s containing in the Detector filter when it is swapped, B_0 (L_f) as the number of 0s containing in the Listening filter when it is swapped and r as the number of repeating pairs and considering this value as slowly variable across following detection windows, the number of bits set to zero, when the Detecting filter is reset, is:

$$B_0(D_f) \approx me^{-(n+r)\frac{k}{m}}$$

while the number of bits set to zero in the Learning filter is:

$$B_0(L_f) \approx me^{-(n/2+r)\frac{k}{m}}$$

Defining $b_0(D_f)$ and and $[b_0(L_f)]$ respectively as $b_0(D_f) = \frac{B_0(D_f)}{m}$ and $b_0(L_f) = \frac{B_0(L_f)}{m}$.

Their ratio allows us to estimate r. In fact

$$\frac{b_0(D_f)}{[b_0(L_f)]^2} \approx \frac{e^{-(n+r)\frac{k}{m}}}{e^{-(n+2r)\frac{k}{m}}} \approx e^{r\frac{k}{m}}$$

From this equation we obtain an estimate for the number of repetition:

$$r = \frac{m}{k}\log\left(\frac{b_0(D_f)}{[b_0(L_f)]^2}\right)$$

Exploiting r we can compute a threshold alpha allowing a better exploitation of the Detector filter. After having estimated the value of r we can modify the swapping threshold alpha to better exploit the Detector filter. Since we want to obtain a number of 0s in the Detector filter equals to m/2:

$$B_0(D_f) \approx \frac{m}{2} \approx me^{-(n+r)\frac{k}{m}} \Rightarrow b_0(D_f) = \frac{1}{2} \approx e^{-(n+r)\frac{k}{m}}$$

then

$$n = \log(2)\frac{m}{k} - r$$

substituting in

$$b_0(L_f) \approx e^{-(\frac{n}{2}+r)\frac{k}{m}}$$

we obtain that the Listening filter is swapped according to e new threshold α:

$$\alpha = me^{-(\log(2)+\frac{-kr}{2m})} = \frac{m}{\sqrt{2}}e^{-\frac{kr}{2m}}$$

the first time we estimate r the threshold α is set to m/√2 and consequently we can obtain r as:

$$r = \frac{m}{k}\log\frac{b_0(D_f)}{1/2}$$

while in the following swapping windows r is computed as

$$r = \frac{m}{k}\log\frac{B_0(n)}{(\alpha/m)^2}$$

3.3.2.4 Rate Counter

Here we propose a design for the rate monitor that operates without pre-established time windows in order to support near-continuous operation. This design is based upon counting Bloom filters with one fundamental enhancement: the filter content is regularly emptied at a given target rate. Consider an ordinary counting Bloom filter with conservative update described by the number of counters m and the number of hash functions *k*. Each counter

$$b[1:m]$$

is assumed to use s bits implying a maximum counter value $S = 2^8$. Assume we want to detect all the flow presenting a specific repeating or varying rate of some characteristic. Let r be the critical long-term average rate per flow we wish to detect, and

$$W = 1/r$$

the maximum long-term average inter-arrival time between the characteristic object of the rate measurement within a flow. W is expected to be a short time interval (on the order of one second) compared with the monitoring time T. We enhance this filter as follows: At the end of each interval W, all non-zero counters in the filter are decremented by one. This has the effect of reducing characteristic count per flow by one every W seconds to a floor of zero, since the number of element object of the counting process per flow f_i is given by

$$c_i = \min(b[H_1(f_i)],...,b[H_k(f_i)]).$$

Note that this enhanced counting Bloom filter is an approximation of a common token bucket [MAS04] in which each flow f_i has at most $S - c_i$ tokens before reaching the overflow value S. The decrement at the end of each interval W is therefore analogous to assigning every flow an extra token. This structure can then be used to detect whether the rate of the characteristic for a given flow exceeds the profile of the token bucket, or generally, whether during i time intervals W more than $S+i-1$ repetition or variation (for the characteristic under observation) occur for a given flow. This technique adapts to the burstiness in contrast to static detection techniques which count variation or repetition overflows per monitoring time T. For example, assume r = 1 variation per second, so W = 1 second; and 3-bit counters, so the overflow limit

$S = 2^3 = 8$. A flow will be detected as critical if more than 8 variations occur within 1 second, more than 9 variations within 2 seconds (e.g. 5 variations per second sustained for two seconds), or more than 10 variations within 3 seconds (e.g. 4 variations per second sustained over three seconds), and so on. Note that over long time scales, any rate in excess of 1 variation per second will be detected.

3.3.2.5 Scanning Detector

The combination of the variation detector module together with the rate counter allows the front-end to isolate the flows suspicious of performing scanning activities. Whenever the variation detector detects a varying flow-feature pair it advertises the flow label f_i to the variation monitor for its accounting. The particular design of the rate counter provides the ability to detect both short bursts of variations in a timely manner and slower persistent scanning behaviour. This approach addresses the drawbacks of a relatively long constant time interval T for the variation monitor.

3.3.2.6 Performance evaluation

We simulated the performance of the basic approach outlined in Section 3.3.2 and comparing it with the proposed approach using a single Bloom filter of size m= 65536 bits for the variation detector and a counting Bloom filter with m=1024 unbounded buckets and k=4 hash functions, with m=256 bits and k=3 hash functions and then with m=1024 bits and k=4 hash functions. For sizing the Bloom Filters we set a false positive probability target psi = 0.001 and use the formula *psi = 0.6185^{m/n}* to determine n, which is the number of $<f_i, g_j>$ pairs that can be inserted in the Bloom filter respecting the probability target. We obtain m/n= 14,3 and the optimal number of hash functions k=*m/n *log 2* = 9. The variation detector filters use k=9 hash functions. Figure 8 and 9 shows the results for the case of 30,000 variations within 1000 flows. These are subdivided into two classes: ”heavy flows”, which generate a large amount of variations, and “light flows”. Figure 8 plots the total variations per flow, the number of variations output from the variation detector, and the number of variations counted by the variation monitor. The figure shows that the variation detector is the limiting factor in the detection performance of the system in this simulation, as the variation detector Bloom filter is not properly sized to accommodate 30,000 variations, causing the false-positive count to increase as the variation detector fills. Figure 9 plots the total variations per flow, the number of variations output from the variation detector, and the number of variations counted by the variation monitor.

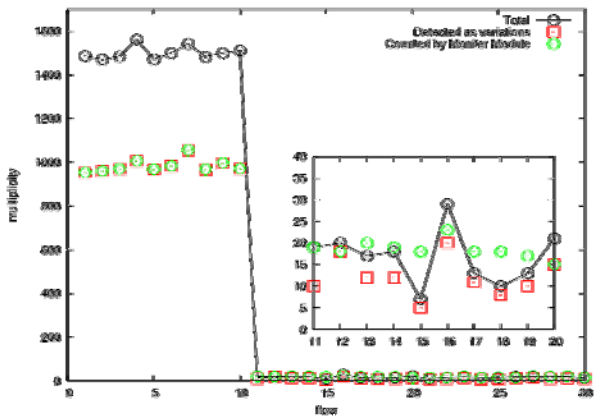

Figure 8: Basic approach

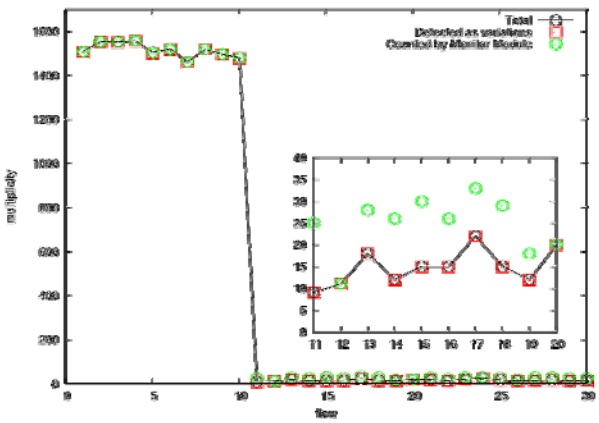

Figure 9: Proposed approach

The proposed approach, if the Bloom filters of the Variation Detector are well dimensioned, is able to detect variation with the false-positive count that is near zero.
The state change between Learning and Detecting happen when the number of zeros in the Bloom filter in Learning state is $m/\sqrt{2}=23170$ (see section 3.3.2.2 for theoretical details).As shown in Figure 10 the usage of a fix threshold causes a waste of memory in the Detecting filter. In particular when the Learning filter reaches the threshold (i.e. green line reaches a total length ratio of 0.3), the Detector filter has not reached the m/2 threshold (i.e. the red line does not reach the total length ratio of 0.5). In order to solve this problem we have proposed the usage of a self-tuning mechanism (Figure 11) that adapts the threshold to avoid memory waste (see section 3.3.2.2 for theoretical details).

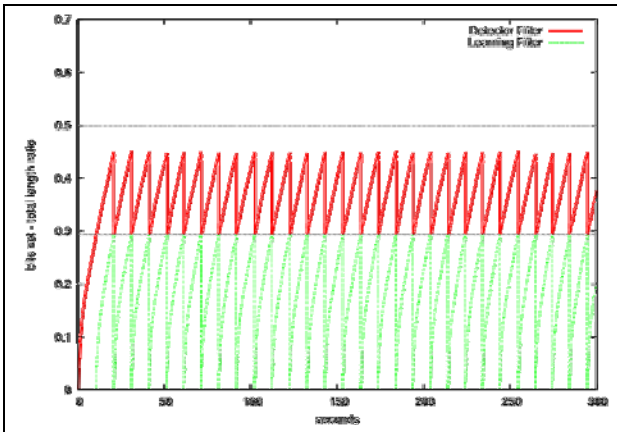

Figure 10: None adapt. threshold mem. usage

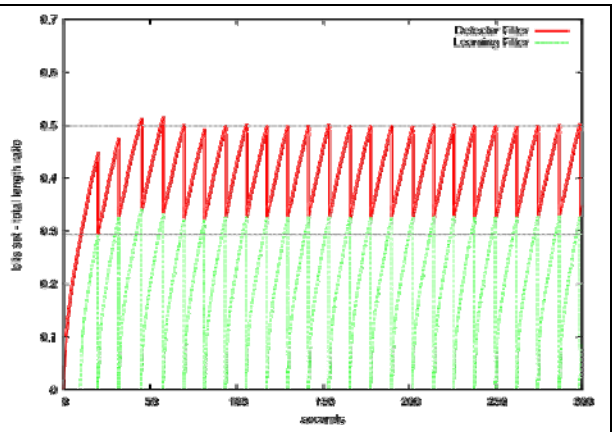

Figure 11: Adaptive threshold memory usage

4 Partitioning functionality within the BE

The PRISM back-end's monitoring services adaptation amounts to an innovative feature of the PRISM system, as far as the deliverance of data to monitoring applications is concerned. The intercessional and privacy safeguarding role of the back-end serves for the deliverance of pre-processed data, embedding functionalities that limit the risk of privacy impingement. Working in contrast to typical systems offering to monitoring applications raw packet data that are not really needed for the underlying monitoring purpose, the back-end adapts several portions of monitoring tasks in order to provide explicit results of data processing and analysis. That is, in the context of the PRISM operation, the back-end undertakes tasks, typically executed by the monitoring applications themselves, which concern the transformation of the data from their native, raw format to the data types that are actually necessary for the production of the final monitoring results.

Therefore, one of the phases of the back-end's operational flow during its mediation between the front-end and the monitoring application (i.e., the end user of the data) is the one concerning the generation of the set of appropriate and proportional data, through the execution of a corresponding set of data analysis. The back-end and its distinct components constitute the entities responsible for processing raw information derived from the front-end and transforming them accordingly, in order to provide them in a useful and a privacy preserving format at the requesting party. The computational strength and the depth of the processing capabilities of the back-end and its embedded components define the level of the aforementioned adaptation.

4.1 Back-end Embedded Processing

In order to prevent excessive disclosure of data to the monitoring application, part of the processing normally performed in the monitoring application itself, is implemented within the PRISM back-end. The Embedded Processing Components of the PRISM back-end will comprise an affluent library of software tools, for executing and performing the necessary actions during the stages of elaboration (basically data transformations ranging from simple to advanced). The two important aspects of this concept are, on the one hand, the components themselves and, on the other hand, their effective orchestration for achieving the desired result.

4.1.1 Data Transformation Functions

In order for the BE to produce the derived data types that will be delivered to the monitoring application instead of disseminating the actual raw monitoring data, it makes use of several analysis functions, the development of which constitutes the ongoing work of the workpackage WP 4.2. As anticipated, final results of the processing are handed over to the monitoring application. The data transformation functions are organised in the following libraries:

- *Anonymisation Library*: It contains a variety of anonymisation functions, such as explicitly altering fields of packets or randomizing them.
- *Aggregation Library*: It enables the production of aggregated data, incorporating functions such as mathematic calculations (e.g., sums), data clustering and generation of mean values. It can be roughly said that this family of functions implements a concept similar to that of statistical databases.

- *Protocol Headers Library*: It consists of different tools, such as *IPv4HeaderHandler* and *TCPHeaderHandler*, each one of which is applicable when specific information must be extracted from or calculated from the value of distinct fields of the respective protocol headers. Typically, the functions accept protocol headers as input, while they return with the values of specific header fields as output.

A fundamental concept behind the back-end's data anomymisation functionality is to rely on a general-purpose tool that can anonymise live or stored data on a per-field basis. That is, a data transformation strategy can define what function should be applied on each application field. In order to additionally provide flexibility in the transformation functionalities engineering, a second level of abstraction has been introduced, by means of a lower-level Application Programming Interface (API) which is exploited by the Embedded Processing Components themselves. This API provides all the elementary transformations, which are then used by the higher-level processing components for the enforcement of a data adaptation strategy.

4.1.2 Embedded Processing Components' execution organization

The Embedded Processing Components are manipulated and exploited by the Monitoring Agents (MA) of the back-end, upon receiving by a monitoring application some request for disclosure of any kind of data. Subsequently to a monitoring application request, the serving MA executes an explicit sequence of operations for generating the data types that are necessary and proportional for the satisfaction of the given request. These operations are grounded on the Embedded Processing Components.

The sequence is specified taking into consideration the incoming CertPDT and CertPDT* certificates that any monitoring application user must obtain from the PPC or an equivalent delegate and submit together with the request. More specifically, the sequence essentially reflects the set of operations that must be performed in order for the data types provided by the front-end (defined by the CertPDP*) to be transformed to the ones that should finally be disclosed to the monitoring application (defined in the CertPDP). In other words, the result of the operational sequence specification is the extraction of a certain guideline schedule that the PEP functionality of the back-end is required to follow. This schedule basically resolves the matter of eventuating in the deliverance of the requested data, given specific raw material (i.e., data originally derived from the front-end).

In that respect, a possible approach that could be followed would concern the execution by the back-end MA of certain algorithms that denote a path of transformations to be tagged along, in order to result in the requested data (accounting as the starting point the raw data retrieved from the front-end). The necessary information is contained in the PRISM Ontology, of which the back-end is aware, in terms of instances of the `DataTransformations` class that map the Embedded Components to source and target data types.

Nevertheless, as already it has been mentioned [D.3.1.2], the PPC component of the PRISM system when it comes to generating and providing certificates that denote the Permitted Data Types for a requesting entity, benefits from the execution of precise algorithms that overlap the ones mentioned above. More specifically, the PPC has already calculated and followed an explicitly defined path of data transformations, in order to map Permitted Data Type (PDT) sets with raw Permitted Data Type (PDT*) sets and to eventually generate the relative certificates. Hence, taking into consideration the stringent performance requirements of the back-end, the execution of the same algorithms for the exact same reason within the latter would be an extravagant waste of resources. For this purpose, the result of the Static Policy Decision Point (S-PDP) reasoning, taking place at the PPC, should be communicated at the back-end.

In order to be communicated to the back-end from the PPC, the calculated path of data transformations is encapsulated –as an attribute– in the CertPDT certificate. This way, the PRISM system evades the burden of a potential additional communication schema between the back-end and the PPC. Subsequent to receiving the static reasoning results of the S-PDP, the occupied MA particularly its Dynamic Policy Decision Point (D-PDP) functionality needs to be instantiated, taking into consideration the indicated workflow and enhancing it. More specifically, the validity of certain ad-hoc conditions as well as the unlikely occurrence of privacy encroachment events should be examined (for instance, the time and the date of a request should be constrained, that is certain employees of an organization should not be allowed to receive data on non-working days, or the disclosure of certain data types should be prevented when disclosure of others has been performed). The `ExclusiveCombinations` and the `Conditions` classes, combined with the `Rules` class, of the PRISM Ontology serve towards the aforementioned goal.

4.2 Data Transformation Workflow Specification Language

In this context, the need for the formal specification of the actions performed inside the back-end has emerged. Subsequently, the Data Transformation Workflow Specification Language (DTWSL) has been defined and constitutes a light proprietary workflow language in order to express the sequential dependencies among a number of tasks to execute. More precisely, the DTWSL is the workflow specification language used for the facilitation of the acknowledgement of the specified function execution sequence, by the Policy Enforcement Point functionality of the back-end MA, in order to conclude to an orchestrated and well organised exploitation of the Embedded Processing Components.
The DTWSL is a particular XML notation representing the inter-task dependencies and defines several proprietary elements and attributes that apply to the PRISM back-end operational workflow. It must be noted that the DTWSL relies on the PRISM Ontology, which is the semantic model of the system; in fact, both the data transformation functions and the different data types that PRISM deals with have their semantic representation in the PRISM Ontology. In the following, the current specification of the DTWSL is presented.

A DTWSL workflow document is always encapsulated in a `<WORKFLOW>` element. As far as the actions' conditional structure is taken into account, the following elements are defined:

- `SWITCH`: This element enables the specified workflow to support conditional behaviour. The activity consists of an ordered list of conditional branches that are considered in the order in which they appear. The conditional branches are defined by the `<CASE>` elements.
- `CASE`: Each `<CASE>` element specifies a set of actions to be performed, in the case when a condition meets. One or more XML attributes of the `<CASE>` element define the specific condition(s). As an example, subsequent to an Embedded Processing Component execution, depending on the diverse possible results of that execution, different operational activities should take place. At least one such element must be located inside a `CASE` section. When multiple conditions must be met in order for the specified group of actions to be instantiated, the `AND` and `OR` XML elements are used, depicting AND and OR relationships respectively.
- `WHILE`: This element enables the multiple executions of the enclosed parts of a workflow, according to the validity of certain conditions. At least one condition must be located inside a `WHILE` section. When multiple conditions must be met, the `AND` and `OR` XML elements are used, depicting AND and OR relationships respectively.

- `TERMINATE`: The `<TERMINATE>` element causes an operation to stop its execution.

Concerning the invocation of back-end Embedded Components, the `<INVOKE>` DTWSL element is considered. It encloses the specification of an operation, including the following XML attributes:

- `embedded_component`: it specifies the Embedded Component that will be executed.
- `source_data_type`: when the invocation concerns the transformation of a personal data type to another, the value of this attribute corresponds to the "source" personal data type, that is, the type that will be transformed. It should be mentioned that the `source_data_type` attribute points to an array of data, as it is common that the execution of an Embedded Processing Component will frequently accept as input sets of data, rather than single instances of data, as e.g., in cases of aggregation. In the case of referring to a single data instance, obviously the array of data, accounting as input, will consist of a single data record. In the case of Embedded Processing Components receiving different kinds of data types as input, multiple `source_data_type` attributes will be used.
- `target_data_type`: when the invocation concerns the transformation of a personal data type to another, the value of this attribute corresponds to the "target" personal data type, that is, the type that will result from the transformation. It should be mentioned that the `target_data_type` attribute points to an array of data, as it is common that the execution of an Embedded Processing Component will return with an output of sets of data, rather than single instances of data. In the case of resulting in a single data instance, obviously the array of data, accounting as output, will consist of a single data record. Regarding the case of Embedded Processing Components returning with different kinds of data types, multiple `target_data_type` attributes are acceptable.

Regarding the manipulation of the Embedded Components and the relative data sets, the DTWSL defines the following elements:

- `DISCLOSE`: it declares that the data of some specified types should be disclosed to the entity that has asked to retrieve them.
- `REJECT`: the request is rejected. This element is mostly part of the dynamic portion of the workflow specification, when more explicit conditions should be verified, thus such an attribute should be designed depicting potential negative authorization.

As it is has been mentioned, it is common that the Embedded Processing Components of the back-end work simultaneously with diverse sets of data types, rather than with explicit data types. Hence, the need for an expression regarding retrieval of classified data types emerges, for the facilitation of the manipulation of those distinct data types. Moreover, regarding the cases of Embedded Processing Components working with single data instances among a series of data of a single type, the requirement for a serial data extraction expression should be satisfied. In that respect, the `retrieve_data_type_from` and `get_next_element` expressions are introduced:

- `retrieve_data_type_from`: it depicts the retrieval of the specified data type set, among an array of diverse data type sets. It includes the following XML elements:

  - `data_type`: it specifies the required data type to be retrieved from a set of different data types. This attribute can point, as it has been analysed, to an array of data of the same type.
  - `from_set`: it denotes the set of data types in which the acquired data type is to be found and retrieved.

- `get_next_element`: it serves for the representation of the extraction of distinct data instances from an array of different data instances of the same type. It specifies the next element to be extracted, following a serial logic, or gets the value `NULL` when no more elements can be retrieved. In this context, the `get_next_element` expression is embracing a `data_type` XML attribute, which denotes the source set of data, under evaluation.

Figure 12 presents an example of a CertPDT certificate including a DTWSL workflow specification. It corresponds to the to the situation in which an actor falling into the role of `NetworkAdministrator` requests for the execution of the service `ISPProviderClassInfoProvision`, which provides a list of all different Internet Service Providers that can be recognised in the packet traces under evaluation, taking into consideration the different IP Address Classes they belong to. The Embedded Processing Components to be used are the *IPv4AddressHandler*, which is used to extract the first octet from the whole IPv4 address, the *IPv4AddressClassChecker*, which takes as input the first octet of an IPv4 address and checks its first bits to determine the corresponding IP address class, the *IPv4AddressToISPMapper*, which is called to map a specific IPv4 address to an explicit ISP name and finally the *DataAggregationHandler*, which aggregates the results in their final form.

```
Certificate:
  Data:

    Version: 2
    Serial Number: 1234266363562
    Signature Algorithm: SHA512withRSA
    Issuer: CN=PPC
    Holder: PRISM
    Validity:
      Not Before: Tue Feb 17 16:23:40 EET 2009
      Not After: Tue Dec 19 16:47:00 EET 2009
     Subject Public Key Info:
           Public Key Algorithm: rsaEncryption
           RSA Public Key: (512 bit)
               Modulus (512 bit):
              1e1a91078b1eddea1989134badc9c89c2207819d7a9
              67874c73354667fcbe57598b34d6771b2108559762a
              9138c784f7488a605f92c53454afae8475063c707d
    Attribute: {
                  {type: Role,      value: NetworkAdministrator}
                  {type: Service,   value: ISPProviderClassInfoProvision}
                  {type: Workflow,  value:

<WORKFLOW>
           <INVOKE embedded_component="IPv4AddressHandler"
             source_data_type="IPv4SourceAddress"
             target_data_type="IPv4SourceAddress1stOctet"/>
           <INVOKE embedded_component="IPv4AddressClassChecker"
             source_data_type="IPv4SourceAddress1stOctet"
             target_data_type="IPv4AddressClass"/>
              <SWITCH>
                <WHILE get_next_element (from="IPv4AddressClass") != NULL>
                 <CASE data-type="ClassA">
                     <INVOKE embedded_component="IPv4AddressToISPMapper"
                       source_data_type="IPv4SourceAddress"
                       target_data_type="ClassAISPName"/>
                     <INVOKE embedded_component="DataAggregationHandler"/>
                       source_data_type="ClassAISPName"
                       target_data_type="ClassAISPNameList"/>
                   <DISCLOSE/>
                 </CASE>
                 <CASE data-type="ClassB">
                   .......
                 </CASE>
                 <CASE data-type="ClassC">
                   .......
                 </CASE>
                 <CASE data-type="ClassD">
                   .......
                   </CASE>
               </WHILE>
             </SWITCH>
       </WORKFLOW>
       }
     }

    Extensions:
      X509v2AttCert Basic Constraints:
        CA: FALSE
```

Figure 12: DTWSL workflow example encapsulated into CertPDT Certificate

5 Partitioning monitoring applications within the PRISM Architecture

PRISM's architecture presents many possibilities for privacy-preserving network monitoring. For simplicity, maintainability, privacy protection, and performance reasons, newly-developed monitoring applications using PRISM should be designed as partitioned into front-end, back-end, and presentation components. For the adaptation of existing monitoring applications, the focus instead is to minimise the effort needed to bring the application into a privacy-preserving environment. Wherever applicable, the preferred approach is to send the application its input data (usually packet or flow traffic data) in its native format. This traffic is reduced and modified at the front-end to eliminate privacy sensitive information in the data sent to the external application. A second approach is to place the whole application within the front-end, using the application itself to do data reduction and using the back-end to control access to the results, which themselves may still contain some privacy-sensitive information. The third approach is a full partition of the application into front-end, back-end and external parts, as with newly-designed PRISM applications. The most applicable approach depends on the particular monitoring application and the monitoring purpose. In the following subsections we will discuss different monitoring purposes and show examples of how to satisfy the purpose in a privacy-preserving way within the PRISM architecture.

5.1 IDS scenarios

Intrusion detection systems (IDS) monitor and analyse network traffic in order to detect ongoing attacks directed towards the end-systems or the network infrastructure itself. IDS are usually defined according to three relevant characteristics:

1. Location
    a. **Host-based** intrusion detection systems (HIDS) are deployed directly on the devices they are supposed to protect. The advantage of host-based systems lies mainly in their great adaptability to the devices (i.e., operating system and applications) they are supposed to protect.
    b. **Network-based** intrusion detection systems (NIDS) run directly on the network infrastructure which is to be monitored for attacks, and they offer the advantage of early interception of attack patterns, thus opening the way to network resource preservation and efficient mitigation of large-scale attacks.
2. Detection strategy
    a. **Signature detection** essentially relies on the comparison of the observed traffic flows with a database of known attacks. Whenever such a system detects a traffic signature matching that of a known attack, it raises a corresponding alarm. A major drawback of such systems is that they are not able to react to novel types of attacks not yet included into the signature database.
    b. **Anomaly detection** systems learn the normal behaviour of a network and raise the alarm whenever the network displays anomalous and thus potentially dangerous traffic patterns. Although anomaly detection offers the possibility of detecting previously unknown attack patterns, it unfortunately also raises the probability of triggering false alarms, especially if naïve policies are implemented.
3. Reaction
    a. **Passive** systems are used for the detection of attacks and anomalies, but they do not offer the possibility of automatic intervention, i.e., threat mitigation. Instead, they rely on the human component for appropriate reactions.

b. **Reactive** systems offer the possibility of automatic attack mitigation or even prevention, without requiring any human intervention. Such systems are usually referred to as **Intrusion Prevention Systems** (IPS), and for seamless network operation they must be tuned such that the number of false alerts is minimised to the extent possible.

In this context as far as the PRISM system is concerned, passive NIDS of either the signature- or anomaly-detection type are the most relevant. In the following sections, we will describe two different approaches towards intrusion detection and intrusion prevention within the PRISM system, which serve as examples for a plethora of concepts which can be realised within the given architecture (cf. [D2.2.1]).

5.1.1 IDS in the front-end

One of the fundamental prerequisites of meaningful IDS/IPS operation lies in having access to the full scope of the traffic transmitted via the observed links. Therefore, it would be very difficult to deploy an NIDS in the back-end of the PRISM system, as this would require the transportation of the complete data stream from the FE to the BE, which would of course be tightly coupled to enormous bandwidth requirements between the FE and the BE, and to exceptionally high requirements upon the storage component of the BE.

Due to these constraints, the most reasonable option for deploying a general purpose NIDS within the PRISM system is to partition the corresponding *traffic analysis* in the following way:

- **The traffic inspection component of the NIDS should become part of the FE.** In this way, the full scope of the traffic will be available to the NIDS, without the requirements to transport huge amounts of data among distributed architectures. In this context, it is important to mention that we do not necessarily see the NIDS as part of a highly specialised type of device in the FE (e.g., the network processor card), but that we could rather imagine a separate piece of hardware with full access to the traffic stream implementing the PRISM front-end interfaces. It should also be noted that, given the popularity of NIDS applications, particularly signature-based NIDS, and a widespread desire among network operators to deploy these systems at high traffic rates that much work in industry is going in to specialised NIDS observation hardware.
- **NIDS reporting should be accessible via the BE.** In practically all cases (considering full traffic rates of up to 1 Gb/s) it should be possible to transport the NIDS alarm reports in real time between the FE and the BE. This architectural choice enables a single standard specification of NIDS rules in the FE, and the subsequent role-based access differentiation of the types of information accessible to different types of users. As an example, the network operator will normally have access granted to all the alarms occurring in the network, while individual users of the same operator's network might obtain only a subset of this information from the PRISM system which is relevant for their own operation (e.g., only information about attacks which are directed towards their own IP address space).

In the next section, we present a specific example for an NIDS targeted towards Session Initiation Protocol (SIP) signalling attacks, which employs a somewhat more advanced splitting of roles by applying a higher amount traffic pre-processing in the FE (essentially based on timers and bloom-filter based counters), which is especially important for identifying attacks in the context of highly stateful networking protocols.

5.1.2 Recognising SIP flood messages

In order to illustrate IDS within PRISM architecture, let us next consider a new monitoring application for detecting so-called SPAM over the Internet Telephony (SPIT). The term SPIT refers to activities where unsolicited, typically automated, calls are made to a large number of victims with some aim such as marketing, disturbing, etc. As VoIP becomes increasingly popular, we expect that SPIT attacks will become more common. One SPIT attack was carried out recently in Germany [Heise08,IPcom08], where numerous SIP INVITE packets [RFC3261] were send to random IP addresses. As a result, a potentially a large number of (badly implemented) VoIP phones started to ring in the middle of the night. The attack did not include any actual voice communication, but was limited to the call establishment phase. In particular, the attack consisted of a single packet sent to each targeted host.

The good news is that this type of attacks is relatively easy to identify at the core or access network level, where one can monitor traffic between several hosts. To this end, let us first consider how such an attack can be identified. We give three complementary approaches here:

Anomaly detection approaches:

- The attack is most likely targeted only to the *de facto* standard port, 5060 (TCP/UDP). Hence, a single IP address sending too many INVITE messages with this port over a relatively short time interval (especially in the middle of the night!) strongly suggests malicious behaviour. For example, a frequency of 30 INVITE messages or more per minute should not occur normally. However, a potential problem is that VoIP gateways / proxies may generate a large number of valid INVITE messages. This can be mitigated by using a whitelist of known gateways. We note also that such an attack may be distributed, with more than one host participating in the attack.
- An attacker choosing the target addresses randomly frequently hits both i) non-existing IP addresses, and ii) hosts where VoIP is not in use (or to be precise, where there is no server listening on port 5060). Hence, either there is no reply at all, or a TCP RESET or ICMP “destination unreachable” response is sent back to the originating host. Thus, measuring the rate of single packet flows with no response, or the rate of TCP RESET / ICMP “destination unreachable” packets, reveals the attacker.

Honeypot-based approach:

- Thirdly, one can use honeypots distributed among real users. Packets sent to such hosts are generally unsolicited and correspond to some sort of malicious activity. In this case, packets sent to port 5060 are very likely to be a VoIP related scan or even attack.

From the above, it is clear that all approaches boil down to measuring rate of a certain type of event per host, i.e., counting. Note that when one analyses a system with random arrival points in time, the quantity of the rate can be defined in various ways. In our case, it is convenient to consider discrete time with time bins in the order of 1 minute. Consequently, the rate corresponds to a sequence of non-negative numbers.

5.1.2.1 Deployment within the PRISM system

The partitioning of duties in this case among components of the PRISM architecture is rather straightforward. Let us first consider the front-end:

1. **Measurement**: The FE's first task is to count the number of events per host, where an event can, e.g., refer to

   i. Packets with INVITE keyword,
   ii. Number of single packet flows with a) no response during, e.g., 1 minute, or b) a response indicating an error (e.g., TCP RESET).

   In both cases, one can limit the operation to port 5060. Depending on the expected maximum number of events and the computational capabilities available, the counting per source IP address can be carried out either exactly or by using Bloom filters. The rate corresponds to the number of packets that arrived during the time-bin divided by the length of time interval.

2. **Notification**: Upon exceeding a given threshold the FE will notify the BE about this event. In order to avoid multiple notifications during a single time slot, the FE can use another Bloom filter to record the notified addresses. Alternatively, a single counter saturating, e.g., at $k+1$ is also sufficient, so that reaching a value of k triggers the sending of a notification. In either case, the data structures such as Bloom filters are reset at the end of each time interval.

   In a simple solution, the FE reveals the suspicious IP address directly to the BE, who can then, e.g., carry out further analysis including counting the number of (consecutive) time slots during which the threshold is exceeded.

   Alternatively, in order to protect privacy, one can use the secret-sharing escrow mechanism as described in [BIA08]. That is, in response to exceeding a given threshold during a single time bin, the FE reveals only a part of the suspicious IP address by using Shamir's secret sharing scheme. This makes sure that the BE does not know anything about the IP address until a sufficient *number* of shares (where the *number* is a design parameter) has been delivered to it. In other words, this process corresponds to *counting* the number of time bins during which the threshold has been exceeded.

   Resetting: When applying Shamir's scheme, it is important to reset the secret (i.e., the Shamir's polynomial) on a regular basis, like e.g., once every k time slots. This avoids the effects of infinite system memory. Otherwise, all traffic would eventually be interpreted as attack traffic.

The counting functionality of events per IP address necessitates stateful operation at the FE. The straightforward implementation of counting events per IP address wastes a lot of memory (e.g., 4 bits per 32-bit IP address translates to 2GB of memory). The application of a dynamic structure accessed through a hash function can mitigate this issue; this is an especially viable option when the analysis is run on sufficiently powerful PC-level hardware. If the resources are particularly sparse when compared to the amount of data, then counting Bloom filters can be used. Hence, stateful operation is unlikely to be a real problem here. We note that instead of revealing the suspicious IP addresses, basically the same scheme can be used to reveal also (some of) the suspicious flows (cf. [BIA08]) in order to facilitate manual inspection afterwards. A more elegant general solution to this class of problem, where the rate threshold is controlled in a leaking bucket fashion, is described in section 3.3.2.3.

The operation at the BE is similarly straightforward. Its main task is to listen for notifications, and as they arrive, disclose the suspicious IP addresses. The suspicious IP addresses may then

be reported to an external entity for potential countermeasures. Finally, we note that the described operation can be also implemented in distributed fashion with several FE components without changing the fundamental logic in FE or BE.

5.1.2.2 Summary of the Requirements from the PRISM system

The described application requires from the FE the following functionalities:

- *Flow key filtering,* the ability to inspect only traffic meeting certain criteria (e.g., ports and protocols).
- Optional *substring search* to identify keywords within the observed protocol.
- *Counting* a number of events, where the range depends on the expected rate of events in the nominal case.

This application has no special requirements on the back-end.

5.2 Billing non-repudiation

A concrete example in which network measurement systems can offer unique value to the operators is the non-repudiation of data volumes attributable to each individual customer, as with the availability of the recorded data streams the operator can easily demonstrate the exact volumes of traffic generated by each user. Technically, in the case of mobile operators, this can be performed based upon the International Mobile Subscriber Identity (IMSI) of the users extracted from stored traffic traces. However, such identifiers are extremely sensitive and should be considered confidential information, which must not be exposed unless mandated by the situation. For volume-based billing models at least the IMSI number, the packet-size, and the timestamp are recorded for each packet. Nevertheless often more information is stored (e.g. IP addresses) in order to have more details during a billing repudiation or abuse investigation. Furthermore, access to this information for employees of the provider is not strictly limited to billing purposes or billing repudiation.

5.2.1 Deployment within the PRISM system

The PRISM architecture can be applied to this problem, by collecting traffic information and protecting sensitive identifiers such as the IMSI at the front-end. The front-end filters the traffic per IMSI and forwards two independent information flows to the back-end. The first information flow is meant for billing purposes, and consists of hashed (i.e., anonymised) IMSI and the number of total bytes sent to the back-end. The back-end has a mapping between customers and hashed IMSIs, which can be used for billing the traffic volume. For non-repudiation purposes flow level information is collected at the front-end. The IMSI is hashed with different hash function than the one used for billing. The information sent to the back-end consists of time, source and destination IP addresses, and the total number of bytes. If a repudiation process is launched by a specific customer, she must disclose her IMSI. With her IMSI, the flow level information can be extracted from the back-end. Without the IMSI of the user, it is not possible for a provider employee to get flow-level information for a specific customer; only the billing information can be gathered. The separation of the information using two different hash functions consequently allows a provider to perform billing without detailed flow-level information. Only in case of repudiation is the flow level information read and handed over to the customer. The information that a provider can offer the customer in case of repudiation is flexible, and consequently the information gathered in the front-end and sent to the back-end my vary in its level of detail, as, e.g., flow-level volume information might suffice completely as opposed to packet-level volume information.

5.2.2 Summary of Requirements from the PRSIM system

The following processing steps are performed in the front-end:

1. *Counting* bytes per IP address for an interval
2. *Flow aggregation*, the creation of flow records with timestamps and counters per 5-tuple.
3. Application of *multiple hash functions* for different purposes data derived from the same source

Functions in the back-end:

1. Map hash(IMSI) to the customer identifier

5.3 Application Detection

Application detection is of great interest for network providers. Detecting P2P traffic is only the best-known motivation of many for application detection. Application detection using passive monitoring techniques is based on traffic classification. Traffic classification is also used for traffic engineering purposes: optimizing link usage, planning new network architectures, and introducing new mechanisms (e.g. QoS) for traffic engineering. Based on the commonly-used approaches, traffic classification can be divided into three different classes:

1. Port-based:
   Port-based traffic classification is fast and simple. The more traffic is from applications using well-known port numbers the higher the accuracy of this algorithm is. However, applications which have an interest in circumventing traffic classification efforts or firewall protections often misuse well-known port numbers; in addition, P2P applications generally do not have well-known port numbers assigned and use multiple ports as an integral part of the protocol. Consequently port-based traffic classification may perform poorly, e.g. less than 70% accuracy [MOR05], especially for those applications most interesting to classify.
2. Statistical:
   Statistical traffic classification [CRO07] detects applications based on identifying statistical properties of the captured IP packets. Traffic can be classified based on packet size, inter-arrival time, or arrival order.
3. Deep packet inspection:
   Traffic classification based on deep packet inspection (DPI) produces extremely accurate classification. The classification is based on the detection of pre-defined unique payload signatures.

In the following sections we describe how two different monitoring applications can be adapted to operate in a privacy-preserving environment. The focus is to use the application as it is and to reduce and modify the input traffic such that no privacy-relevant information is given to the application.

5.3.1 Detecting P2P traffic by using Appmon

Appmon is a traffic classification application that uses deep packet inspection to determine the application for each traffic flow present in an observed network. This scenario focuses on using Appmon to detect P2P traffic. However, since it is based on DPI, it has access to full payload information of every observed packet, and therefore clearly compromises the privacy of the end-users of the network. This scenario addresses this problem using the PRISM architecture.

5.3.1.1 Deployment within the PRISM system

Appmon used together with the PRISM system can detect the applications without compromising user privacy as follows:

The PRISM front-end captures and classifies packets based on port numbers. According to [KIM08] the accuracy of per-port traffic classification strongly depends on the port and degrades when an application uses ephemeral non-default ports or when default ports of an application coincide with port masquerading P2P applications. To identify traffic that does not use the well-known port numbers, or hides behind well known-port numbers, special algorithms within the front-end must be applied in order to improve the accuracy of the traffic classification.

Therefore such traffic is sent to the back-end and classified with payload-based classification applications [ANT06]. The decision whether to forward traffic is based on two questions:

- Is the port known to be used with port masquerading by P2P applications?
- Is the amount of traffic decisive for the overall results (misclassifying e.g. 1% of the traffic can be neglected)?

When a flow matches the above criteria the traffic is handed over to Appmon via the back-end. To allow Appmon to operate on the data, at least the following information must persist from the original packet data:

- 100 bytes of payload for detection based on protocol control messages
- full payload (for example for passive FTP)
- packet size
- port numbers
- involved hosts (IP addresses can be anonymised)

5.3.1.2 Summary of requirements from the PRISM system

Privacy preserving application detection using Appmon requires from the front-end the following functionalities:

1. *Flow key filtering,* the ability to inspect only traffic meeting certain criteria (e.g., ports and protocols).
2. *Counting* packets per port number for an interval.

The back-end requires providing the following functionalities:

1. *Construction of a packet trace* from information received from the front-end, obfuscated with random values.
2. *Replay* of the constructed trace, as Appmon can only operate in “live” capture mode.

5.3.2 TSTAT Skype detection engine in a privacy preserving environment

Detection of Skype traffic is of interest for providers that offer a VoIP service, and much effort has gone into reliable detection of Skype traffic. TSTAT is a tool which implements a Skype traffic detection engine based on the algorithms described in [BON07]. This paper presents a comprehensive analysis of Skype traffic characteristics and identifies patterns that are typical for Skype traffic; this is not a trivial task given that all Skype traffic is encrypted.

Therefore, in order to identify Skype communications, the entire traffic stream on the network is analysed by the TSTAT; as with DPI approaches, this has a clearly negative impact on the privacy of the network's end-users. This scenario combines TSTAT with PRISM to mitigate this.

5.3.2.1 Deployment within the PRISM system

PRISM used together with TSTAT can successfully detect Skype traffic and ensure user privacy. In this case, TSTAT is used as external monitoring application; however, in contrast with its normal operation, it does not observe all of the traffic on the network. Here, the traffic is filtered, reduced and modified in the front-end. For filtering traffic a subset of the algorithms described in [BON07] are implemented in the front-end. The front-end only forwards network traffic that is encrypted, and has Skype-like characteristics and is thus likely Skype traffic. Based on the aspect that only packets that are encrypted on the network are forwarded to TSTAT and privacy sensitive information from the headers is deleted, no privacy sensitive information exists in the information given to TSTAT by the back-end. To pre-filter Skype-suspicious traffic the decision in the front-end can be based on some characteristics described in [BON07]:

- Average packet rate between 10 and 100pkt/sec
- Flow duration is longer than a given threshold (10s)
- For Skype VoIP flows: Average packet size is small (between 30 byte and 300 byte)
- For Skype out traffic: Port 12340 is used and first 4 bytes of UDP payload match a given signature; remaining bytes have Shannon entropy higher than a given threshold.
- Byte 1 and 2 and bit 4-8 of the 3rd byte of the UDP payload match a given signature; remaining bytes have Shannon entropy higher than a given threshold.
- TCP payload bytes have Shannon entropy higher than a given threshold.

The portion of the packet handed over to the back-end depends on the information used by TSTAT to identify Skype traffic. At least the following characteristics must remain present in the modified trace.

- Skype encrypted payload
- inter-packet time (relative packet arrival time)
- packet-size
- protocol and port numbers
- involved hosts (note that IP addresses should be anonymised but must remain unique)

Note that proprietary protocols such as the one used by Skype can change at any point in time; this detector is thought to work at the time of writing, but this may change without warning.

5.3.2.2 Summary of Requirements from the PRISM system

From the front-end the following functions are required:

1. *Flow aggregation*, the creation of flow records with timestamps and counters per 5-tuple.
2. *Flow rate, duration, and packet size measurement.*
3. *Shannon entropy calculation* on packets within a flow.

For the back-end must provide adaptation to a system requiring live trace information as above.

5.4 Summary of Requirements from the PRISM system

Network traces are of great interest within the network research community. Almost all research areas use network traces, or could benefit from using them; examples include the development of new routing schemes or protocols, the improvement of queuing algorithms, the detection of new applications, and so on. However, currently only a few traces are available to the research community. Raw, unobfuscated trace data is often used within the organizations that collect it, but cannot be shared to provide a basis for the "science" of network research, as experiments on undisclosed data cannot be repeated, and there can often be uncertainty as to whether the experiment's results are data-dependent. On the other hand there are well-known, publicly available traces where all privacy-sensitive information has been deleted, though this often has the consequence that the traces cannot be used for a wide range of research purposes. Recent trends in attacks against network data anonymisation have also led to the threat of the reduced availability of even this inadequate research data source. PRISM can offer trace files for different purposes, two examples are shown in the following subsections. The content of the trace files is strictly limited to the information needed for a specific purpose.

5.4.1 Publicly available Traces for Traffic Classification

The development of new algorithms for traffic classification is ongoing. Due to the lack of publicly available traces, the evaluation of the new algorithms is mostly done on self-collected traces. Consequently, the comparison of different algorithms is difficult and often not even carried out at all [KIM08]. Papers like [KIM08] compare different existing algorithms based on their own private traces. If a new algorithm is developed it will not be possible to compare it to their results, because the traces are not publicly available. Consequently, only a few existing algorithms can be compared in a proper way. As argued in [CRO07], a solution to allow traffic classification on anonymised traces would be to add meta-data to the traces (e.g., the application behind a flow found by deep packet inspection). Such traces ensure user privacy and allow for research on traffic classification.

5.4.1.1 Deployment within the PRISM system

PRISM allows the publication of traces to the research community, where end-user privacy is protected and sufficient information remains in the traces to allow useful research. For privacy-preserving trace publication all privacy sensitive information must be deleted or anonymised. As shown in [D2.1.1], IP addresses must be considered personal data, so IP address information must be anonymised. In full traces the real IP addresses are generally not crucial to the research at hand, so as long as host uniqueness and/or network structure are preserved, anonymisation has no negative impact on trace utility. Packet payload must also be protected, as many users are not using encrypted connections.

In this scenario, the PRISM front-end collects packets, does short-term analysis on the received packets for e.g. classification, and generates meta-data of the results. For long-term analysis and storage, the packets are sent to the back-end. The trace-file generated by the back-end is handed over to the researchers, together with the meta-data. For the purpose of trace publication for traffic classification in the front-end an application detection engine based on payload inspection is performed [KIM08]. The result of this classification is called the metainformation of the packet. IP addresses within the packet are anonymised, and the payload is deleted. The remaining part of the packet together with the metainformation is sent

to the back-end and stored. Packets that cannot be identified by short-term analysis in the front-end are sent to the back-end to be further analysed. The results of the back-end analysis (metainformation) are stored together with anonymised packet information. Based on a request from a researcher, the back-end generates a trace-file which together with the metainformation is handed over to the research community.

Since multiple different anonymisations of the same data set can be used to regenerate the original data, the PRISM back-end must keep a log of which data was anonymised and how, and to whom the data was given. This logging serves to warn the network operator of the danger of de-anonymisation and to deny publication in case of such danger.

5.4.1.2 Summary of Requirements from the PRISM system

To enable privacy preserving publication of routing traffic the front-end provides the following:

1. String search functions to detect the application
2. Deletion of packet payload and other individual fields
3. Anonymisation of IP addresses

The back-end must provide the following:

1. *Construction of a packet trace* from information received from the front-end and back-end analysis results
2. Additional traffic classification analysis functions.

5.4.2 Publicly Available Traces with Routing Traffic

Other network traffic besides packets containing privacy-sensitive data could be of interest for researchers. Examples for the specific purpose of routing research are routing protocols, Cisco Discovery Protocol (CDP), or spanning tree information. Traces containing control plane information are of great interest to the research community in the analysis of the behaviour of Internet routing protocols in real world environments.

Such trace files are useful to evaluate the behaviour of routing protocols in real environments, improve routing protocols or test new routing protocols. In an example we focus on a wireless community network, where mesh routing protocols are used and traces may be handed over to the research community for developing and analysing routing protocols.

5.4.2.1 Deployment within the PRISM system

To analyse routing, the front-end filters traffic such that only packets with routing information in it will match. The IP addresses are anonymised such that the network from which the traces were generated cannot be identified. Further header fields that are unnecessary for the purpose are removed or anonymised. This packet stream is handed over to the back-end and stored there. A researcher may query the relevant data from the PRISM back-end, with the back-end verifying that the given operator had the necessary privileges for publication and the researcher has the privileges to receive traces. A packet trace with the information in the database together with random fields is generated and forwarded to the application.

Again, as above, multiple different anonymisations of the same data set can be used to regenerate the original data. Therefore, the PRISM back-end must keep a log of which data was anonymised and how, and to whom the data was given. This logging serves to warn the

network operator of the danger of de-anonymisation and to deny publication in case of such danger

5.4.2.2 Summary of Requirements from the PRISM system

To enable privacy preserving publication of routing traffic the front-end provides the following functionalities:

1. String search functions to detect routing traffic,
2. Function to delete individual fields of a packet.

The back-end must then provide the following functionalities:

1. *Construction of a packet trace* from information received from the front-end, obfuscated with random values.
2. *Audit logging* for tracking publication details.

6 Conclusions and outlook

The PRISM system aims at preserving the users' privacy at the same time when various traffic monitoring activities, essential, e.g., for the network maintenance and planning, are carried out. To this end, the fundamental new approach introduced by PRISM takes the monitoring application and the user role into account for access control and requires the specification of a monitoring purpose for which the data is collected and processed. The work presented in this deliverable demonstrates some concrete examples of how monitoring applications can be used and integrated within the PRISM architecture for dedicated monitoring purposes. It furthermore provides insights into the detailed mechanisms provided for front-end processing and selected back-end components, which allow a very flexible usage for the different kinds of monitoring applications.

The aim of this work is to leave any integrated monitoring application itself untouched, to the extent possible; we have several motivations for this. The major one is, direct integration allows network operators and other entities to use applications they have already deployed, but made privacy-aware with the data protection concepts and approaches presented in D3.1.2 by using PRISM as a “virtual observation point” which provides only appropriately protected data. This also allows us to take advantage of future releases of existing monitoring applications.

The work carried out in the context of this deliverable will be the basis for the final deliverable of work package 3.2. In D3.2.3 the currently loosely coupled functional components based in the front-end, back-end, or in the external applications have to be connected in order to provide a complete solution for selected monitoring purposes as described in Section 5. D3.2.3 will also incorporate the anonymisation mechanisms and tools as elaborated by WP3.1.

References

[ANT06] D. Antoniades, M. Polychronaki, S. Sntonatos, et. al. "Appmon: an Application for Accurate per Application Network Traffic Characterization. In Proceedings of BroadBand Europe. Dezember 2006.

[ANT09] [23] Antichi G., Ficara D., Giordano S., Procissi G., Vitucci F., "Counting Bloom filters for Pattern Matching and Anti-Evasion at the Wire Speed", IEEE Network Magazine, Special issue on Network Intrusion Detection, January 2009.

[BIA08] G. Bianchi, S. Teofili, and M.Pomposini, "New directions in privacy-preserving anomaly detection for network traffic", In Proccedings of the 1st ACM workshop on Network data anonymization. November 2008.

[BIS05] [6] G. D. Bissias, M. Liberatore, D. Jensen, B. N. Levine, "Privacy Vulnerabilities in Encrypted HTTP Streams", in Proc. of 5th Workshop on Privacy Enhancing Technologies, Cavtat, Croatia, May 2005.

[BLO70] [15] B. Bloom. Space/Time Tradeoffs in Hash Coding with Allowable Errors, in Communications of the ACM 13:7 (1970), 422--426.

[BON06] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in LNCS 4168, 14th Annual European Symposium on Algorithms, 2006, pp. 684–695.

[BON07] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi and P. Tofanelli "Revealing skype traffic: when randomness plays with you" SIGCOMM Comput. Commun. Rev., Vol. 37, No. 4. (October 2007), pp. 37-48.

[BYU05] [8] J. W. Byun, E. Bertino, N. Li, "Purpose based access control of complex data for privacy protection", Proc. of the 10th ACM symposium on Access control models and technologies, Stockholm, Sweden, June 2005.

[CLA08] B. Claise, ed. "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information". RFC 5101, January 2008.

[COF01] [11] K. G. Coffman and A. M. Odlyzko, "Internet growth: Is there a "Moore's Law" for data traffic?", Technical report, AY&T Labs, June, 2001.

[CRO06] [7] M. Crotti, F. Gringoli, P. Pelosato, L. Salgarelli, "A statistical approach to IP-level classification of network traffic", in Proc. the 2006 IEEE International Conference on Communications, June 2006.

[CRO07] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli. Traffic Classification through Simple Statistical Fingerprinting. ACM SIGCOMM Computer Communication Review Volume 37, pp5-16, January 2007.

[D2.1.1] F. Gaudino et. al Assessment of the Legal and Regulatory Framework PRISM Deliverable.

[D2.2.1] G. Bianchi et. al High level System Architecture Specification, PRISM Deliverable.

[D3.1.2] C. Schmoll et. al. Preliminary Data Protection Algorithms Specification and Analysis, PRISM Deliverable.

[ELD99] [10] C.A. Eldering and M.L. Sylla and J.A. Eisenach, "Is there a Moore's law for bandwidth?", IEEE Communications Magazine, 37:10, October 1999.

[EU00] [3] Charter of Fundamental Rights of the European Union, O.J. C 364/1, 18.12.2000.

[EU95] [4] Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data; O. J. L. 281, 23 November 1995.

[FIC08a] [21] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, ``Multilayer Compressed Counting Bloom Filters", in Proceedings of IEEE INFOCOM '08, Phoenix, AZ, April 2008.

[FIC08b] [22] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, ``Blooming Trees: Space--Efficient Structures for Data Representation", in Proceedings of IEEE International Conference on Communications 2008, ICC'08, Beijing, China, May 2008.

[HEISE08] Erste grössere Attacke gegen deutsche VoIP-Netze, Heise Security web-page, 23.09.2008, http://www.heise.de/security/Erste-groessere-Attacke-gegen-deutsche-VoIP-Nutzer--/news/meldung/116335

[HIN02] [5] A Hintz, "FingerprintingWebsites Using Traffic Analysis", in Proc. of 2ndWorkshop on Privacy Enhancing Technologies, San Francisco, CA, USA, April 2002.

[IPCom08] Klaus Darilion, ``Analysis of a VoIP Attack", Technical report, IPCom, October 2008, http://www.ipcom.at/fileadmin/public/2008-10-22_Analysis_of_a_VoIP_Attack.pdf

[KIM08] H. Kim, kc claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet Traffic Classification Demystified: Myths, Caveats, and the Best Practices. In Proceedings of ACM CoNEXT, Dezember 2008.

[MAS04] [9] F. Massacci, N. Zannone, “Privacy is Linking Permission to Purpose”, Proc. of the 12th International Workshop on Security Protocols, 2004.

[MON08] [18] A. Montanari et al. “Counter braids”, Information Theory Workshop, 2008. ITW '08. IEEE.

[MOO03] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. “The Spread of the Sapphire / Slammer Worm”.Proceedings of IEEE Security and Privacy, July 2003.

[MOO65] [13] G. E. Moore, “Cramming more components onto integrated circuits”, Electronics, 38(8), April 9, 1965.

[MOR05] A. Moore and K. Papagiannaki. Toward the accurate identification of network application. In PAM, April 2005.

[RAM03] [20] S. Ramabhadran and G. Varghese, “Efficient implementation of a statistics counter architecture”, ACM SIGMETRICS 2003.

[RFC3261] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, RFC3261, ``SIP: Session Initiation Protocol'', 2002.

[SUN02] [12] Sundar Iyer, Rui Zhang and Nick McKeown, “Routers with a single stage of buffering”, ACM SIGCOMM Comput. Commun. Rev., 32:4, 2002, pp. 251–264.

[VAR02] [19] Estan, Varghese, New Directions in Traffic Measurement and Accounting, SIGCOMM 2002.

[VAR04] G. Varghese et al. “Building a better Netflow”, in Proceedings of ACM SIGCOMM 2004.

[VAR08] [14] G. Varghese, J. A. Fingerhut and F. Bonomi, ``Detecting evasion attacks at high speeds without reassembly'', in Proceedings of ACM SIGCOM '08, Pisa, Italy 2008.

[WAN04] Jia Wang and Oliver Spatschek, “Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement”.