Computer Science Department
University of Crete

A Framework for Network Traffic Anonymization

Master's Thesis

Dimitris Koukis

February 2006
Heraklion, Greece

A Framework for Network Traffic Anonymization

Dimitris Koukis

Master's Thesis

Computer Science Department
University of Crete

Abstract

Lack of trust is one of the main reasons for limited cooperation between different organizations. Private data is of paramount importance to administrators and organizations, which are reluctant to cooperate with each other and exchange network traffic traces. The main reasons behind this reluctance to exchange monitoring data are protecting users' privacy and the fear of infrastructure information leakage. Anonymization is a technique that can be used to overcome this reluctance and enhance the cooperation between different organizations with the smooth exchange of monitored data. Anonymization is performed by altering data in such a way that private data and sensitive information are removed. Today, several organizations provide network traffic traces that are anonymized by software utilities or ad-hoc solutions that offer limited flexibility. The result of this approach is the creation of unrealistic traces, inappropriate for use in evaluation experiments. Furthermore, the need for fast on-line anonymization has recently emerged as cooperative defense mechanisms have to share network traffic.

Our effort focuses on the design and implementation of a generic and flexible anonymization framework that provides extended functionality, covering multiple aspects of anonymization needs, and allows fine-tuning of the privacy protection level. The core of the proposed framework is based on AAPI, a flexible and expressive anonymization application programming interface.

Supervisor: Professor Evangelos Markatos

Μία Προγραμματιστική Αφαίρεση για την Διατήρηση Ανωνυμίας Δεδομένων Δικτύων

Δημήτρης Κούκης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Περίληψη

Η έλλειψη εμπιστοσύνης είναι ένας από τους βασικούς λόγους για την περιορισμένη συνεργασία μεταξύ διαφορετικών οργανισμών. Τα ιδιωτικά δεδομένα είναι υψίστης σημασίας στους διαχειριστές και στους οργανισμούς, οι οποίοι είναι διστακτικοί στο να συνεργαστούν μεταξύ τους και να ανταλλάξουν δεδομένα δικτύων. Ο βασικός λόγος για αυτή την διστακτικότητα στην ανταλλαγή δεδομένων, είναι η προστασία των ιδιωτικών δεδομένων των χρηστών και ο φόβος για διαρροή πληροφοριών για την δικτυακή υποδομή του οργανισμού. Η ' διαδικασία διατήρησης ανωνυμίας ' (anonymization) είναι η τεχνική που χρησιμοποιείται για να ξεπεραστεί αυτή η διστακτικότητα και να ενισχυθεί η συνεργασία μεταξύ διαφορετικών οργανισμών με την ομαλή ανταλλαγή δεδομένων επιτήρησης δικτύων. Η τεχνική αυτή τροποποιεί τα δεδομένα έτσι ώστε να αφαιρεθούν οι ιδιωτικές πληροφορίες και ευαίσθητα δεδομένα. Σήμερα πολλοί οργανισμοί παρέχουν δεδομένα αυτού του είδους χρησιμοποιώντας εργαλεία ή αυτοσχέδιες τεχνικές που προσφέρουν περιορισμένες δυνατότητες. Το αποτέλεσμα είναι η δημιουργία μη ρεαλιστικών δεδομένων, ακατάλληλα για πειραματική χρήση. Επιπλέον η ανάγκη για γρήγορη μετατροπή των δεδομένων απευθείας από το δίκτυο, έχει προκύψει καθώς μηχανισμοί άμυνας απαιτούν την ανταλλαγή δεδομένων δικτύου. Σε αυτή την δουλειά επικεντρωνόμαστε στον σχεδιασμό και υλοποίηση ενός γενικού και ευέλικτού μηχανισμού για την ' διαδικασία διατήρησης ανωνυμίας ' που προσφέρει εκτεταμένη λειτουργικότητα, καλύπτοντας πολλαπλές ανάγκες που εμφανίζονται σε αυτό τον τομέα και επιτρέπει τον ακριβή καθορισμό του επιπέδου προστασίας τής ιδιωτικότητας. Ο προτεινόμενος μηχανισμός αποτελείται από μια προγραμματιστική αφαίρεση με το όνομα AAPI.

Επόπτης: Καθηγητής Ευάγγελος Μαρκάτος

Acknowledgments

I am grateful to my supervisor, Evangelos Markatos, for his helpful cooperation during all my studies, his guidance and patience. I would like to thank Kostas Anagnostakis for his support through all these years and his trust in me. Many thanks to Spiros Antonatos, Demetres Antoniades and Panagiotis Trimintzios for their cooperation in the development of AAPI.

My best thanks to my friends and colleagues Spiros Antonatos, Mixalis Polychronakis, Manos Moschous, Giorgos Dimitriou, Manos Athanatos, Demetres Antoniades, Antonis Papadogiannakis, Mixalis Foukarakis, Elias Athanasopoulos for sharing with me the best years of my life, as well as the whole Distributed Computing Lab.

I would like to thank my family for everything they have offered me and helping me make my dreams come true.

Finally, my sweetest thanks to Stavroula for making my days brighter.

Στούς γονείς μου

Part of this work will appear in the Proceedings of the 2006 IEEE International Conference on Communications (ICC 2006), June 2006[20].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Monitor network traffic and traces are a powerful means of evaluation experiments (c.f., [31, 25, 17, 18]), allowing researchers to study network characteristics and behavior. Furthermore, the use of network traces is extended to the Internet security domain, in areas such as the evaluation of intrusion detection systems. In the ideal case for the user, network traces should be shared unchanged, providing full information. However, for both security and privacy reasons, monitored network traffic and traces have to be modified before they become publicly available. This modification process is known as *anonymization*. The aim of the anonymization is triple.

First, to protect the *privacy* of the monitored users. Revealing any sensitive information about the users is totally prohibited. Examples of such information are the web pages that a user has accessed, credit card numbers, unencrypted sessions that might reveal passwords, peer-to-peer connections, e-mails sent and received, etc. In fact, privacy protection is so complicated that most administrators play on the safe side, taking the "reveal nothing" policy. This approach instructs that parts of traffic that might reveal sensitive information, such as the packet payload, are either completely removed or replaced by random values.

Second, to hide any information about the internal infrastructure of the network. Ideally, anonymized network traffic should not by any chance reveal the hosts inside the monitored network that are alive, neither any other of their characteristics, such as their operating system. Also, people which access to the monitored traffic should not be able to extract the monitored network's subnet formation – how many subnets exist and how many hosts each one contains. In order to achieve this goal, existing approaches randomize the IP addresses, thus hiding the identity of hosts and subnet information, and replace packet header fields, that might reveal any of the network characteristics, with constant values. Other approaches, like encrypting IP addresses in a prefix-preserving way [45], are subject to network information leakage as described in [46].

Finally, anonymized monitored traffic has to be as realistic as possible, i.e., as close as possible to the non-anonymized packet stream. Many evaluation experiments done by researchers rely on network traffic traces, thus the results have to be close to those taken using real network traffic. As most anonymized traces that are currently publicly available are unrealistic, not keeping the dymamics of the original and real traffic, most researchers collect private traces to perform their experiments. However, the extrapolation of their results to wide-area scale is difficult, if not impossible.

It is clear that a generic global anonymization scheme cannot exist since different organizations have different needs. Network administrators should be able to specify their anonymization policies at varying levels of detail. For example, traffic from an anonymous ftp server needs only few modifications (e.g client IPs) in order to be distributed publicly, while traffic from an organization's internal network imposes several privacy threats and the need for strict anonymization arises. Existing anonymization tools are not adequate to provide such flexibility and are not capable to address all anonymization needs, since in most cases they were build having a specific range of anonymization policies in mind. In all cases, they work on predefined fields and most of them perform only header-level anonymization.

The anonymization framework that is presented and evaluated in this thesis offers a wide range of anonymization functions that can be applied to *any field* of a packet or a record, up to the application level. The expressiveness of the framework allows creation of anonymized traffic that is able to express any balance between privacy protection and realism. In order to simplify the development of anonymization tools and make anonymization policy definition a quick and simple process, our framework provides an Application Programming Interface (API) named `AAPI`. `AAPI` is very simple to use as any anonymization policy is expressed as a set of function calls, without having to use any unfamiliar scripting languages. Moreover, the framework is extensible enough to provide the user with the ability to implement new anonymization functions. Also, it is trivial to support anonymization for new application level protocols and different traffic sources such as Netflow [8] records. To our knowledge, AAPI is the first framework available for network traffic anonymization.

1.1 Thesis organization

The rest of this thesis is organized as follows. Chapter 2 presents related work. In Chapter 3 we describe the anonymization API and its design in detail. Chapter 4 discusses integration of the anonymization in a network monitoring framework. Chapter 5 presents the experimental evaluation and Chapter 6 concludes the thesis.

Chapter 2

Related work

In this chapter we present related work in the field of anonymization. First, we review the research work that has been carried out on anonymization. Next, we present the available anonymization tools, and finally we present the most frequently used anonymization policies. These policies are mostly used by the organizations and institutions that make traffic traces publicly available.

2.1 Research Efforts

Several research efforts target on the broad field of anonymity (for example [19, 15, 36]) but only few deal with anonymization in network monitoring. Most of them deal with algorithms that defeat attacks on the anonymization.

Peuhkuri [32] deals with two problems of packet traces, enormous storage needs and anonymization. The algorithm proposed for anonymization of IP addresses makes use of cryptography, thus the mapped address is produced by merging a part of the original address with the encrypted value. Although this approach is secure, as it is mentioned, an adversary can inject packets into the trace which will be identified in the anonymized trace so mappings will be revealed. This attack is known as active fingerprinting [30].

Xu, Fan, Ammar et al. [45, 46] focus on the problem of prefix-preserving IP address anonymization. Prefix-preserving means that IP addresses that share a N bit common prefix, when anonymized will be mapped to addresses that that have N bits of their prefix in common. Existing implementations, such as in `tcpdpriv` [16], have many drawbacks like large memory requirements, inconsistent mappings across different anonymization sessions and lack of parallel processing of traces[1]. The approach described in [45, 46] uses stateless cryptography algorithms that have minimal memory requirements. As long as the cryptographic key is the same, the anonymized addresses are preserving their original prefix, that is if two real addresses belong to the same subnet then the anonymized ones will also belong to the same –but different than the original– subnet. Since prefix-preserving mapping is a stateless function applied to IP addresses, parallel anonymization is also feasible. Moreover, the mapping is independent of the order that IP addresses appear in the trace. An implementation for prefix-preserving anonymization, called Crypto-PAn [9], is publicly available from Georgia Tech University. The authors mention that their approach is as secure as the traditional approach of prefix-preserving, where random bits are inserted to the mapping. On the other hand, prefix preserving anonymization is less secure than random one-to-one mapping of IPs, since if the mapping of one address is compromised (for example in the case of a web fingerprinting attack [40]), then more information can be revealed from IPs that share common prefixes.

Prefix-preserving anonymization has also been applied to Netflow [44], Cisco's protocol for collecting IP traffic information. A sofware package for

[1]Parallel processing here is the process of splitting a trace in multiple parts and anonymize them in parallel in different CPUs

IP address anonymization has been modified in order to generate the cryptographic key that is used from a pass phrase. Anonymization is applied only to IP addresses of flows, while all other fields are left unchanged. The authors have extended their tool in [39] where the users are able to anonymize the eight most common fields of a NetFlow record.

Anonymization has also been implemented in hardware, using network processors [35]. Prefix-preserving anonymization has been used and a new algorithm is proposed since existing methods consume either too much memory or too many CPU cycles, and therefore are not able to operate on Gigabit links. The key idea is to pre-compute the entire anonymization function (that is visualized as a binary tree) so that a mapping can be found quickly. In order to minimize memory consumption, the same translation is used for several prefixes, so only unique subtrees of the whole anonymization tree are stored. Also, the most significant bits are hashed rather than being anonymized using the anonymization tree in order to make anonymization more resistant against attacks. Common prefixes are not preserved in these bits. This work has been implemented using the IXP2400 network processor and evaluation shows that the system can keep up with Gigabit links.

The most recent research work in anonymization deals with the creation of a proper policy for packet header anonymization [29]. Starting from the development of a new tool for anonymization, called tcpmkpub [24], the authors try to find which fields of the packet headers should be anonymized and in which way. Decisions are based on known attacks on privacy which they try to defeat. Tcpmkpub is proposed for the implementation of the defined policy and the use of meta-data is suggested in order to provide extra information for the trace. Finally the information loss that the anonymization process introduces is computed and the whole anonymization process is validated.

All aforementioned research works focus only on packet header anonymization. Paxson and Pang in [30] introduce a way to anonymize payload and remove sensitive information rather than removing the entire payload. Packets are reconstructed to flows and application level parsers modify the data stream as specified by the policy. Finally, data is split again into packets and merged with packet headers, thus creating legitimate traffic as if no anonymization/reconstruction had been applied. The algorithm that regenerates packets is implemented in a way that keeps the properties and dynamics of the original traffic as much as possible, but it is not always feasible to retain one-to-one mapping between input and output since modifications have taken place. However, this introduces several drawbacks. For example, malicious traffic may contain overrun packets or a wrong value in the checksum field. When packets are reconstructed, packet header values are calculated so that these malicious packets will not appear in the output trace as in the input. Also the idea of "knowledge separation" is introduced, where a value is not consistently mapped to a certain value but changes according to other parameters. For example, a client IP address is anonymized based on the IP address of the server that it accesses. This way, even active fingerprinting can be defeated, but, on the other hand, this is a trade-off between the information that remains in the trace and the risk of information leakage.

Finally, a totally different approach on anonymization for network monitoring is proposed by Mogul [27]. In this presentation it is mentioned that traditional anonymization techniques are no longer effective because they either leave much sensitive information, or leave few data, so traces are useless. Instead, he suggests that the code should move to the data, meaning that central repositories of code should exist where users will be able to make their experiments. Using this approach, it is clear that there should be a way to check that the code does not use any sensitive information from traces neither in an explicit nor in an implicit way. The main drawback of this approach is that is difficult to be implemented.

2.2 Anonymization Tools

Few tools are publicly available for anonymization of network traffic traces. Here we provide a brief description for each one of them.

Tcpdpriv [16] is the most known tool of its category. It works only on traces written in tcpdump format and removes sensitive information by operating on packet headers. TCP and UDP payload is removed, while the entire IP payload is discarded for other protocols. The program provides multiple levels of anonymization, from leaving fields unchanged up to performing more strict anonymization, like mapping of IP addresses to integers. Level 0 implements an one-to-one sequential mapping IP addresses to integers. Level 1 has similar functionality except that the address is treated as two different portions of 16 bits, while in level 2 each byte of the address is manipulated separately. Level 50 implements prefix-preserving anonymization and finally level 99 leaves IP addresses unchanged. Similarly, the user can specify the level of anonymization for information such as ports, IP class or multicast addresses. Tatu Ylonen has written an article [47] that describes how an adversary can obtain sensitive information from anonymized traces that are created using tcpdpriv with -A50 option. No functionality is provided for altering the TCP or UDP payload, for example replacing the URL with a constant value or removing the username and password from an FTP session. The NLANR traces [28] are anonymized by using tcpdpriv.

There are some tools based on tcpdpriv's code. Ip2anonip [12] is a simple filter that turn IP addresses into host names or anonymous IPs. Ipsumdump [13] dumps packets into ASCII format and uses tcpdpriv to anonymize IP addresses if specified by the user. Tcpdpriv is also used in click router [4] for anonymization of the source and destination IP addresses.

An implementation for prefix-preserving anonymization is available from Georgia Tech University. The software is called Crypto-PAn [9] (Cryptography-based Prefix-preserving Anonymization) and provides an API for prefix-preserving anonymization. Using a simple tool that is provided with the distribution, the user is able to anonymize the IP addresses of a packet trace in a prefix-preserving manner.

The latest addition to the anonymizatin tools inventory is tcpmkpub [24], a tool for packet header anonymization. Tcpmkpub has no built in support for protocols and the user should specify the packet format using the tool's policy language. For each field the user specifies the function that will be applied, choosing from the available (KEEP, ZERO) or by implementing custom callbacks. It also included the functionality of exporting metadata for the anonymized trace, providing more information that can be very useful for analysis.

The main drawback of all the above tools is that they only work up to the network level and cannot anonymize information on the application level, like for example randomizing the `URL` field of an HTTP request. Furthermore, they provide only a few anonymization primitives such as sequential mapping or prefix preserving mapping which can be applied only to a few predefined fields such as IP addresses and TCP ports. The proposed framework enable the user to perform anonymization both on header and application level in any protocol field, using a wide range of available anonymization functions

Bro [22] is a Unix-based Network Intrusion Detection System (IDS). The authors have implemented a plug-in that can be used to anonymize traces using a high-level language [30]. Users can express sophisticated trace transformations by writing short policy scripts. The tool is able to alter both packet headers and payloads. Moreover, it can operate on application-level and manipulate fields that exist in application messages such as filenames in FTP traces or URL in HTTP. This way, the output trace contains payload that is very useful but private information has been removed. Berkeley National Laboratory, distributes publicly anonymized ftp traces [23] using this software. Although their approach is quite flexible, it has several limitations and drawbacks. First, it provides only a few anonymization primitives (constant substitution, sequential numbering, hashing, prefix-preserving mapping and adding random noise), forcing the user to write his own functions in `Bro` language, a custom scripting language. Moreover, as `Bro` works with events, a user can alter packet header fields only for those protocols which have a registered event that supports trace transformation. That is, the anonymization of the IP addresses of a trace would require non-trivial effort to write the suitable policy scripts, one for each protocol (HTTP, FTP, telnet, etc.). The proposed framework, provides a larger set of primitives that can be applied to all packet fields up to and including the application level. Furthermore, the usage of a simple, lightweight API for a standard programming language is more practical and extensible. Finally, as it is shown in chapter 5 the proposed framework is much faster than Bro.

Name	*TCP/IP header support*	*Application payload*	*Anonymization functions*
tcpmkpub [16]	IP addresses, TCP/UDP ports, TCP/IP options	No functionality	Mapping, prefix-preserving
tcpmkpub [24]	Access to all fields, user specified protocol	No functionality	ZERO, KEEP, user defined
Bro [22]	Predefined fields (e.g. IP addresses)	Access to HTTP and FTP header fields through events	Mapping, prefix preserving, user defined

Table 2.1: List of anonymization tools.

2.3 Most Frequently Used Anonymization Policies

Several universities and research centers share public traces following different anonymization policies. In this section, we provide an extensive description of current anonymization policies.

The most popular source of public traces is the Passive Measurement and Analysis (PMA) site of the National Laboratory for Applied Network Research (NLANR). NLANR provides daily traces from several sites in the United States in tcpdump format. The NLANR traces [28] have packet payload removed, source and destination IP addresses are mapped to integers, while time-to-live and IP identification number have constant values. In case of TCP or UDP, the TCP or UDP payload accordingly is removed, otherwise the whole IP payload is removed. All other fields, that is header length, type of service, protocol number, fragment flags and offset, and total length, remain intact. This form of anonymization is useful for processing which requires only packet header fields, such as the counting of packet inter-arrivals or determine a flow volume . As another example, we can perform port-scanning and anomaly-based detection, which do not require access to the payload. However, traces anonymized by NLANR cannot be used for applications which require deep-packet inspection, such as zero-day worm detection [38, 6] and signature-based intrusion detection [37, 22].

The University of California, Los Angeles (UCLA) traces [42] are not in tcpdump format but are provided in simple text format. Each line describes a packet where source and destination IP addresses are mapped to integers. Only ports, flags, sequence numbers, acknowledgment numbers, and window size are recorded. Packet payload is totally removed, thus leaving space only for header processing, similar to the approach of NLANR. A similar form of anonymization (sanitized traces in text format) is also followed by the Lawrence Berkeley National Laboratory (LBL). LBL traces [23] keep even less information; source and destination IP addresses are mapped to integers and only source/destination ports and packet size are recorded. LBL also provides some HTTP and FTP traces in tcpdump format, where the IP addresses are mapped to integers, the rest of the header is unchanged, but sensitive information in the payload like URLs, filenames, or passwords has been replaced by constant values. While providing the payload for specific protocols is better than always hiding it, replacing sensitive fields induces loss of precision. For example, the CodeRed worm [5] attacked the web servers by requesting a carefully crafted URL. If this URL is replaced by a constant value, such attacks will not be detected.

In the University of California, San Diego traces [43], tcpdpriv is used for anonymization. Source and destination IP addresses are mapped to integers, while packet payloads are filled with zero. All other header fields remain unchanged. In the traces provided by the Dartmouth College [11], prefix-preserving anonymization is used for source and destination IP addresses, payload is completely removed but the rest of the fields are unchanged. Both approaches share the same disadvantages with the one followed by NLANR. However, in the case of Dartmouth College, IP addresses are anonymized in a prefix-preserving way, unlike other approaches that map addresses to integers. Prefix-preserving anonymization reveals more information about IP address as addresses that belong to the same real subnet will also belong to the same subnet after anonymization. A brief summary of the anonymization policies applied by the mentioned organizations and departments is is presented in Table 2.2.

Organization	*IP Source/Destination addresses*	*Ports*	*Payload*	*Comments*
NLANR [28]	Mapped to integers	Intact	Removed	IPid and TTL replaced by constant values
UCLA [42]	Mapped to integers	Intact	Removed	Provided as text
LBL [23]	Mapped to integers	Intact	Removed	Provided as text, only IP, ports and packet size recorded
UCSD [43]	Mapped to integers	Intact	Set to zero	All other header fields unchanged
Dartmouth [11]	Prefix-preserving	Intact	Removed	All other header fields unchanged

Table 2.2: Summary of most the commonly used anonymization policies.

Chapter 3

The Anonymization API

The need for anonymization may vary from simple policies, like removing payload and sequential numbering of IP addresses, to complex ones, like for example the case of altering multiple fields in the HTTP protocol. One should be able to create a policy that reveals no private information but on the other side is useful enough to meet his needs. The proposed Anonymization Application Programming Interface (AAPI) addresses all these needs and provides a flexible way to apply anonymization policies to both live traffic and packet traces.

AAPI is an API for the C programming language that allows users to apply anonymization primitives on traffic. The selection of the C language was made for three reasons. From a design point of view, libraries that capture traffic are also written in C, thus AAPI can directly communicate with them. From a users' point of view, it is much simpler to write a set of function calls, rather than trying to describe a policy using unfamiliar script notations. Finally, the performance of the anonymization process is very critical, especially in case of anonymizing live traffic at very high speeds.

3.1 AAPI functions

The notion behind AAPI is that anonymization is a series of functions that are applied to a traffic stream. The core functions of AAPI are divided into three main categories. First, AAPI provides the anonymization functions that *alter fields* of the packets or records in the given traffic stream, e.g randomize or replace them, perform prefix-preserving anonymization on IP addresses, etc. Second, the *filtering* functions BPF filters [26] and string searching have been implemented. Filtering functions allow to distinguish parts of the traffic stream and apply complex policies such as “leave all the UDP packets unchanged but randomize the payload of all TCP packets” or “anonymize all packets that contain the GNUTELLA-Connect pattern”. Finally, *application-level stream* functionality has been intergrated in AAPI, which is called *cooking* and *uncooking*, that provide the ability to compose and decompose application-level streams.

The main function of AAPI is `add_function(set, function, ...)`, where `‘...’` denotes a variable number of arguments, according to the applied function. AAPI expresses each anonymization policy as a single or multiple *sets* of functions. Each *set* is a logical group of functions that are executed sequentially, in the order they had been applied. Sets are created through the `create_set()` function. Once a packet is captured, it is passed through each set and for each set is processed by its functions. A function can prevent the traversal of a packet in the subsequent functions by simply returning zero. This behavior is extremely useful in cases of filtering functions as we show in a following example. The combined flexibility of sets and filtering functions allows the user to express “`if-else`” scenarios or even different anonymization policies within the same program. The function sets

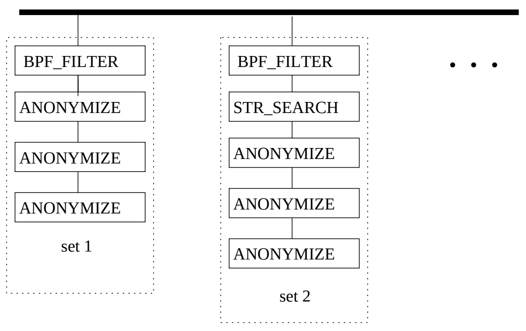


FIGURE 3.1: **Function sets: Each packet is passed through each set and for each set is processed by its functions.**

are visualized in Figure 3.1.

The argument `function` defines which specific function will be applied. Natively, AAPI supports the following functions:

- "ANONYMIZE" (field anonymization)
- "BPF_FILTER" (BPF filtering)
- "STR_SEARCH" (string searching)
- "COOK" (stream reassembly)
- "UNCOOK" (splitting a stream to its original form).

As it will be shown in later sections, user functions can also be added in order to extend the function support.

Whenever we apply the function "ANONYMIZE", which is the main anonymization function that the `add_function` is refined as `add_function(set, function, protocol, field, parameters)`. AAPI provides a variety of anonymization functions, including:

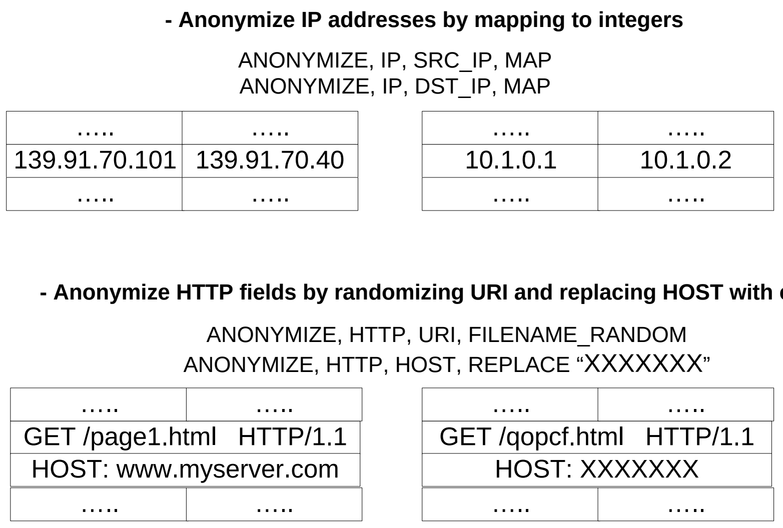
- **hashing** (MD5, SHA, CRC32, AES and DES algorithms)

- **random** for generic fields and for filenames/URIs,
- **mapping** to either sequential values or based on some distribution (uniform, Gaussian, etc.)
- **replacing** with constant integers or strings
- **prefix-preserving** for IP addresses (cryptographic and map based),
- regular expression **substitution**
- **checksum** adjustments for all protocols
- **removing** fields mainly used for application-level protocols.

Thus, AAPI provides a wealth of functionality for user needs. Moreover, new functionality can be added by the user as described later. A complete list of functions is provided in Appendix B.

The parameter `protocol` describes which specific protocol and layer the anonymization function will work on. Our current implementation supports IP, TCP, UDP, ICMP, HTTP and FTP. At the application level, we currently fully support HTTP (including HTTP/1.1 features such as persistent connections) and FTP but the modular design of AAPI permits easily the support for other protocols. The `field` parameter defines the field of the protocol on which the function will be applied. As an example, "time-to-live" (TTL) and "source IP" are two valid fields for the IP protocol. A complete list of fields for each protocol is provided in Appendix A.

Finally, the last argument, is a list of parameters that need to be passed to the function. Note that we cannot apply all anonymization functions to all fields. For example it does not make any sense to remove (STRIP) the source address from the IP header since the packet will not be valid any more.



- Anonymize IP addresses by mapping to integers

ANONYMIZE, IP, SRC_IP, MAP
ANONYMIZE, IP, DST_IP, MAP

…..	…..
139.91.70.101	139.91.70.40
…..	…..

…..	…..
10.1.0.1	10.1.0.2
…..	…..

- Anonymize HTTP fields by randomizing URI and replacing HOST with constants

ANONYMIZE, HTTP, URI, FILENAME_RANDOM
ANONYMIZE, HTTP, HOST, REPLACE “XXXXXXX”

…..	…..
GET /page1.html HTTP/1.1	
HOST: www.myserver.com	
…..	…..

…..	…..
GET /qopcf.html HTTP/1.1	
HOST: XXXXXXX	
…..	…..

FIGURE 3.2: **Example of anonymization on network packets**

A simple map to constant will have the same anonymization effect without compromising the usefulness of the trace. Internally, AAPI performs such sanity checks for each function applied before start processing packets and inform the user for wrong usage of functions.

In Figure 3.2 we show how anonymization is performed using AAPI. In the first example we want to anonymization the IP addresses of a packet by mapping them to integers. In order to apply this anonymization we have to instract AAPI to anonymize (ANONYMIZE) at the IP protocol (IP), the source and destination address (SRC_IP, DCT_IP) using sequantial mapping (MAP). We can see that AAPI transformed the IP address 139.91.70.101 to 10.1.0.1 thus anonymizing it. In the second example we present a more complex anonymization. We anonymize the HTTP header (HTTP) and more specially by randomizing (FILENAME_RANDOM) the URL field (URI) and replacing with a constant (REPLACE ”XXXXXXX”) the host field (HOST). In the figure we can see how these fields are anonymized according to the anonyzation policy we have specified above.

In the following example we will describe an anonymization policy and we will show how it can be implemented with AAPI. The policy is: "*remove the TCP payload for TCP packets, remove of IP payload for all other packets, all packets must have their IP addresses anonymized by mapping them to random integers*".

Before we proceed to the AAPI code, we should note that this policy divides the packets into two categories, TCP and non-TCP. It is thus very useful to apply filtering functions to distinguish the packets and then for each category apply the appropriate anonymization functions. "BPF_FILTER" function returns zero if the filter does not match, elsewhere returns one and the packet is processed by subsequent functions. The given anonymization policy is implemented as follows with our AAPI:

```
int set1=create_set();
int set2=create_set();

add_function(set1,"BPF_FILTER", "tcp");
add_function(set1,"ANONYMIZE", IP,SRC_IP,MAP);
add_function(set1,"ANONYMIZE", IP,DST_IP,MAP);
add_function(set2,"ANONYMIZE", TCP,PAYLOAD, STRIP);

add_function(set2,"BPF_FILTER", "ip and not tcp");
add_function(set2,"ANONYMIZE", IP,SRC_IP,MAP);
add_function(set2,"ANONYMIZE", IP,DST_IP,MAP);
add_function(set2,"ANONYMIZE", IP,PAYLOAD, STRIP);
```

Note that each packet will match to only one set (it can be either TCP or not) and in case of TCP the "STRIP" function is applied to the TCP payload.

3.2 Anonymization of Application-level Streams

Information in high-level protocols, like HTTP or FTP, spans across multiple packets, thus anonymization on this level should be performed on top of a reassembled application stream instead of on a per-packet basis. AAPI has the ability to reassemble packets in order to form a cooked packet, through the "COOK" function [1] . It is thus strongly recommended that a "COOK" function must precede the anonymization functions that work on high-level protocols. Take as an example a user who wants to set the contents of an FTP transfer to zero. The file being transferred is usually split into multiple TCP/IP packets. If we try to apply anonymization without cooking, then only the first packet of the transfer will be classified as "FTP-packet" since it is the only one that contains the protocol headers. The rest of the packets composing the actual file transfer cannot be classified as such, and therefore cannot be anonymized. When cooking is applied, the whole transfer is grouped in a single "application-level" packet so the contents of the whole file can be set to zero.

However, one of the targets of anonymization is that the output should be as close to the input as possible, in order to retain the usefulness of the trace. Therefore, our approach is, after we perform cooking and anonymize the application-level stream, to split the cooked stream back to the original series of TCP/IP packets. Splitting is implemented as an AAPI function called

[1] For the reassembly functionality of COOK function, the libnids [34] library has been used.

"UNCOOK". The cooking function stores the list of headers of the original packets that form the cooked packet. "UNCOOK" takes this list of headers and adds to them the appropriate portion of the payload of the cooked and anonymized packet. In that way, "UNCOOK" constructs as many TCP/IP packets as there were originally in the incoming traffic, with each one having the same header as before the application of cooking, although the payload has been anonymized. It must be noted that after the uncooking, some of the TCP/IP packets may not have any payload after the "UNCOOK" function, although they originally had, when for example we are replacing the whole application-level payload with a hash value.

3.3 Function (Re-)Ordering

Function (re-)ordering is an optional component of the anonymization framework that can be selectively enabled or disabled by the user. The goal of re-ordering is dual. Firstly, we want to automatically detect common pitfalls in the list of the anonymization functions, both in which anonymization functions are applied and in which order. Secondly, we want to ensure that the semantics of anonymization process are correct. Function reordering is done before we start processing any packet stream. There are three main tasks:

- All anonymization functions except "CHECKSUM_ADJUST", which adjusts the checksums to a correct value, that are applied on IP, TCP, UDP or ICMP level are moved first. If they were placed between a "COOK" and an "UNCOOK" function, then the headers stored by "COOK" would not be anonymized and "UNCOOK" will emit non-anonymized packets.
- "CHECKSUM_ADJUST" and functions that alter the packet length

fields are applied at the end of the anonymization. “CHECKSUM_ADJUST” is called last in order to reflect all changes, after the rest of the anonymization functions have been applied. Updating the packet length is also applied at the end because other anonymization functions may modify the original packet’s size. As a result, explicit modifications to the packet length must be performed at the end.

- If the policy requires to use functions that modify an application-level protocol (HTTP, FTP, etc.), they are all grouped together in order to apply “COOK” and “UNCOOK” only once. If a “COOK” function exists, then it is placed before any application-level anonymization function, otherwise it will automatically applied by AAPI. Similarly, if an ‘UNCOOK” function exists, it is applied only after we have performed all application-level anonymization, otherwise it is manually applied. Having “COOK” before and “UNCOOK” after the functions that work on HTTP or FTP level preserves both the correctness and the transparency of the anonymization process. Additionally, if two or more “COOK” or “UNCOOK” functions are accidentally added then duplicate functions are removed in order to eliminate the overhead. We should note here that since “COOK” is performed after all header level modifications, certain fields such as TCP sequence number that are essential for reassembly, should not be modified. Providing that no TCP or IP header fields can be removed, altering fields such as IP addresses or TCP ports using one-to-one mapping, does not affect reassemble.

Reordering also detects and removes common pitfalls in the anonymization policy. For example, consider that a policy first hashes the URL and then removes it. When reordering is applied, the first modification will be

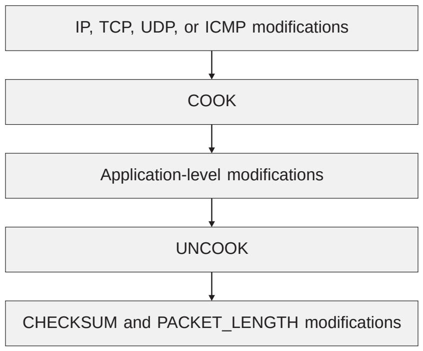

IP, TCP, UDP, or ICMP modifications
COOK
Application-level modifications
UNCOOK
CHECKSUM and PACKET_LENGTH modifications

FIGURE 3.3: **The order of functions after applying reordering.**

removed since it is useless. The proper ordering of anonymization functions is illustrated in Figure 3.3.

3.4 Extensibility

Extensibility is one of the main design goals of AAPI. Extensibility applies in three different aspects of AAPI. It allows to easily a) add new anonymization functions, b) support new protocols, and c) have as input different types of traffic sources.

As far as the first issue is concerned, a user can easily add more anonymization functions into the framework, taking advantage of its modular design. As an example, one may add to AAPI a new anonymization function for IP addresses that hashes the first 8 bits and randomizes the rest. Moreover, we provide a callback functionality, meaning that the user can specify a function that is called for each packet, thereby getting raw access to packets.

In our current implementation, the application-level protocols supported are HTTP, FTP and NetFlow v10. It may be desirable to add new protocol decoders in the framework. For example, users are able to write a Simple Mail Transfer Protocol (SMTP) decoder in order to anonymize email message contents.

Finally, it is straightforward to add support for different input sources. For example, snort alert logs [37] can be supported by simply adding a new decoder that reads such records and provide references to each field. Since our framework is generic in the sense that it just applies anonymization functions to protocol fields, support for snort alert logs is as simple as adding a new protocol, the "SNORT_ALERT_LOG" protocol, with its corresponding fields. This way the framework can anonymize snort alert logs, without any other change, using the same notation and anonymization functions described in this work.

3.5 Anonymization Optimization

Taking advantage of the framework design, optimization of the anonymization process is feasible. It is a common case where different applications have approximate the same anonymization policy, or at least share a common subset of the anonymization functionality.

For example, consider the case where two sets exist, one for anonymization of HTTP traffic and the other for FTP traffic. These sets perform different anonymization on application level, but the same on TCP/IP headers, for example map IP addresses to integers and remove TCP options. Functions that constitute the common subset, can be applied only once in incoming packets and the anonymized packets that are produced will be delivered to the subsequent functions of each set. In Figure 3.4 we present this example. There are two anonymization sets including some anonymization functions. The grey shaded boxes are the common group of functions that exist in both sets. Since this functionality is the same in both sets, it can be performed only once. As we can se in the figure, when anonymization is applied a new set (called `Optimization set`) is created, that contains the group of

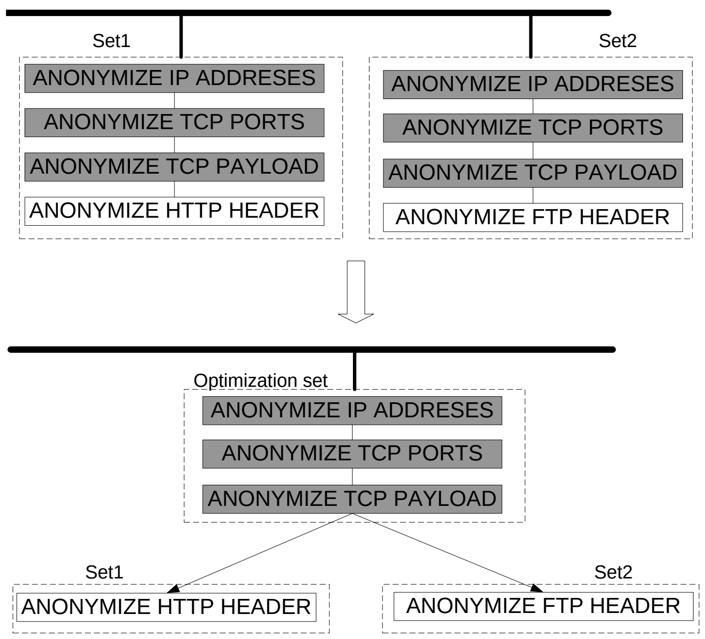


FIGURE 3.4: **Example of anonymization optimization.**

common functions. The rest of each initial set after removing this common subset remains as is (Set1 and Set2). The output of the Optimization set is passed to each set for further processing.

Using optimization, the framework scales efficiently with the number of different anonymization applications (in AAPI each application is a set).

3.6 Input and Output Functionality

AAPI works both offline with traces as well as on-line with real traffic. The framework natively supports live traffic from standard Ethernet interfaces. In case of offline traces, we support traces in the `tcpdump` format. In both cases for the packet capturing functionality, the libpcap [3] library has been used. The modular design of our framework permits the addition of other sources both on-line or offline, as discussed in section 3.4. Currently, all sets read traffic from a single source but we intend to support multiple sources in later versions.

The anonymized packets in AAPI can be recorded to an output trace (in standard tcpdump [3] format). As it would be impractical to have a separate output trace created for each anonymized set, sets can share their output trace. The sharing is simply done by setting the same output filename in multiple sets. For example, two sets can write their anonymized packets in the same output file. In the absence of this functionality, the user would have to merge the two traces using external tools. It should be noted that if policy defines that a packet matches multiple sets, e.g no filtering functions or not mutually excluded filters, then it would be recorded multiple times in the shared output trace, probably with different form.

Chapter 4

Integration with Passive Monitoring

In this chapter we present our efforts in order to integrate AAPI into a network monitoring framework. We give a brief description of the passive monitoring framework (MAPI) and we describe the design of an architecture that enforce anonymization policies to users.

4.1 Introduction to MAPI

The Monitoring API (MAPI) [2] was designed within the IST project SCAMPI [1], and is presented in [33]. It is an expressive programming interface, which enables users to clearly communicate their monitoring needs to the underlying traffic monitoring platform.

MAPI builds on a simple and powerful abstraction, the *network flow*, that allows users to tailor measurements to their own needs but in a flexible and generalized way. In MAPI, a network flow is generally defined as a sequence of packets that satisfy a given set of conditions. These conditions can be arbitrary, raging from simple header-based filters, to sophisticated protocol analysis and content inspection functions.

The main novel abstraction of MAPI is that it elevates network flows to first class status, allowing programmers to perform a set of standard operation on flows similar to other system abstractions such as files and sockets. In particular, users may create or destroy flows, sample or count packets of a flow, apply functions to flows, and retrieve other traffic statistics from a flow. Where necessary and feasible, MAPI also allows the user to trigger custom processing routines not only on summarized data but also on the packets themselves, similar to programmable monitoring systems. The expressiveness of MAPI enables the underlying monitoring system to make informed decisions in choosing the most efficient implementation, while providing a coherent interface on top of different lower-level elements, including intelligent switches, high-performance network processors, and special-purpose network interface cards.

AAPI is a stand-alone framework. We decided to integrate the framework in MAPI in order to offer anonymization functionality in a monitoring infrastructure and also to take advantage of the various optimizations and hardware support that are already integrated in MAPI. MAPI is currently deployed in a distributed monitoring infrastructure [41], so there is a strong need for privacy. Using the AAPI integrated in MAPI, users can be sure that no sensitive data is revealed to others.

Also AAPI can gain from the advantages of MAPI. MAPI supports the collection of data from additional hardware interfaces -such as DAG cards [14] or the SCAMPI card [10]. Also, some of the basic anonymization procedures could be implemented by hardware in the near future, so real-time anonymization at very high speeds will become feasible.

4.2 Authentication & Authorization Architecture

The administrator of each MAPI sensor should be able to define access rights for the users. That means he should be able to specify which users can access the sensor and what privileges each will have. For example, a trusted user from the same organization may be able to retrieve packets with no modifications, while another user should retrieve packets that have anonymized IP addresses and their payload has been removed.

AAPI is able to express this need for flexible anonymization policies on a per user basis. On the other hand there should be a mechanism that enforces the anonymization policy to the users' flow. This way, if a user does not apply the appropriate anonymization, the flow could not be initialized. Therefore, an authentication/authorization mechanism should be introduced in MAPI.

4.2.1 Existing mechanisms

Current authorization mechanisms in MAPI are based on local decisions. That is, the local administrator of each monitoring sensor decides the access level for each user.

Figure 4.2 illustrates the steps that must be performed by the administrator of a monitoring sensor for the specification of the anonymization policy, as well as the steps that take place during the authorization of a user, which includes checking if the flow is compatible with the anonymization policies.

Initially, the administrator must specify the anonymization policy for the monitoring sensor (step 1). The policy is expressed using KeyNote [7] conditions. A sample KeyNote policy is shown in figure 4.1. This policy denoted that the particular administrator enforce the particular user to apply

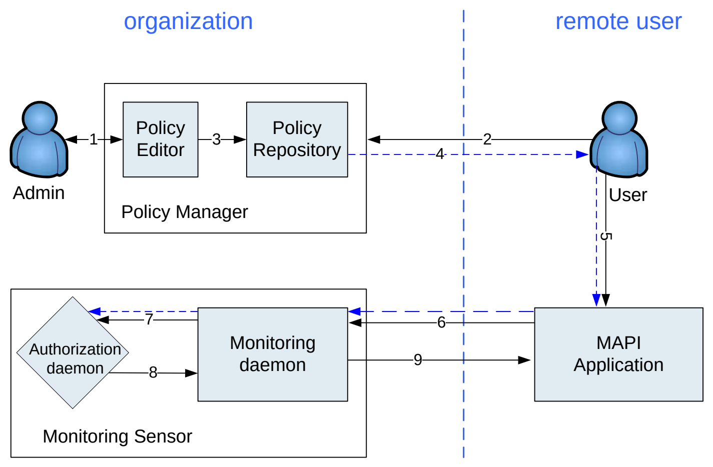
```
Authorizer: "RSA:abc123" #  Admin's key
Licensees:  "RSA:xyz987" #  User's key
Conditions: ANONYMIZE == "defined" &&
            ANONYMIZE.0.pos == 0 &&
            ANONYMIZE.0.param.0 == TCP &&        } remove
            ANONYMIZE.0.param.1 == PAYLOAD &&    } payload
            ANONYMIZE.0.param.2 == STRIP;        }
Signature:  "RSA-SHA1:234354f9"
```

FIGURE 4.1: **Sample KeyNote policy.**

the ANONYMIZE function with parameters TCP, PAYLOAD and STRIP, meaning that the user will capture packets with removed payload.

A user that wants to use the monitoring sensor must first acquire the necessary credentials for that sensor. Credentials delegate authority to a user (or a user group) identified by a public key (or a set of public keys). Thus, the user first has to deliver his/her public key to the administrator (step 2), which is added into the Licensees field of the credentials. The administrator then signs the credentials and stores them in the Policy Repository (step 3). Since the credentials are digitally signed, they can be easily distributed over untrusted networks, so the user can safely download them from the Policy Repository (step 4), in order to use them for accessing the sensor(s). Note that the credentials include the anonymization policy that this user should adhere to.

According to the credentials given by the administrator, the user configures the network flows in his/her application by applying any required anonymization functions (step 5). Before creating a new network flow, the user has to call mapi_set_authdata(), which informs the sensor which are the credentials of the user, and which public key should be used to identify him/her. The authentication in order to prove that he/she is really the user



1. The Administrator specifies an anonymization policy.
2. The user delivers his/her public key.
3. Signed credentials are stored in the Policy Repository.
4. The user obtains his/her credentials.
5. User's network flows are configured to conform with the policy.
6. Each network flow is instantiated.
7-8. authd evaluates the specification of the flow for compliance with the supplied credentials.
9. The flow is activated and the relevant flow descriptor is returned to the user.

FIGURE 4.2: **Access control in MAPI.**

that corresponds to the public key is achieved by supplying a randomly generated integer, together with its encrypted form using the users private key. If the encrypted value decrypted with the supplied public key equals the original integer, the users request is authenticated. When the configuration of the flow is completed, the user instantiates the flow by calling mapi_connect() (step 6). With the dashed arrow we denote the credentials sent to the server while with the black arrow the flow data.

At this point, mapid (the monitoring daemon) has complete information about the flow in question. The user has provided his/her public key and credentials, while mapid is aware of the specification of the flow, as a result of the consecutive mapi_apply_function() calls that the user issued to configure it. All this information is sent to authd[1] (step 7), which checks the authentication and evaluates the specification of the flow for compliance with the supplied credentials (i.e., which include the anonymization policy). authd then returns the result of the evaluation to mapid (step 8). If the flow specification complies with the credentials and the authentication is successful,

[1]Authd is the authorization daemon

mapid activates the flow and returns the relevant flow descriptor to the user (step 9), otherwise the flow is rejected. Steps 6-9 are repeated each time the user wants to create a new flow.

4.2.2 Weaknesses of current scheme

In the current approach, the authorization mechanism is user-based and is specific for one sensor only. If the user wants to access several sensors, for example, he needs to obtain several credentials from several local administrators: one administrator for each sensor. We see two major drawbacks in this approach. First, this solution is not scalable with the number of users. Each administrator of each monitoring sensor has to deliver credentials to each user individually, which means excessive administrative overhead. To make matters worse , when a sensor joins the infrastructure, its administrator has to issue a separate credential for each existing user. Furthermore, deletion of a user causes additional administrative cost since the user should be deleted from each individual monitoring sensor. Finally, credentials from different sensors may conflict, leaving the user to decide which the most suitable combination is (if it is feasible). For example if a sensor specifies strip of payload while another specifies hash of payload, there does not exist a combination that can satisfy both sensors.

4.2.3 Authentication/Authorization mechanism

In order to overcome the weaknesses that were presented in the previous section, we propose an architecture for authorization and authentication based on the abstractions of Virtual Organization (VO) and Certification Authority (CA), abstractions which have been successfully used in the area of GRID Computing. Our vision is to simplify the authentication and authorization mechanisms, to lower the administrative cost, and to free the user from dealing with credentials. The proposed solution introduces a central authority that will be used in order to authorize users that can use MAPIs' sensors.

In MAPI, a VO represents a group of people of common access privileges. People in the same group may also have common research interests, and/or common needs. For example, assume researchers in the field of Internet security that run a project to identify worm outbreaks. In order to accomplish this, they need access to network traffic and more specifically to packet payloads. To enable these researchers to use the MAPI infrastructure, we propose to create a new VO, named wormTeam: all researchers will be members of this VO. Every sensor that supports this VO should define a policy which will apply to all the members of this wormTeam VO. When a user is added to this VO, no individual sensor needs to change its policy for the WormTeam VO. Similarly, when a user is deleted from this VO, again no monitoring sensor needs to change its policy for the VO. As another example, consider a sensor in FORTH, which supports two VOs: FORTHteam and FORTHothers. The first VO would include people working in the FORTH institute and have full access to network packets. All other users would be assigned to the FORTHothers VO, which would have a policy that permits access to anonymized packets only.

Each MAPI user can be a member of several VOs. When, however, MAPI users try to access packets from a particular monitoring sensor, they must specify one of the VOs they belong to. Thus, we need the appropriate mechanisms for registering MAPI users with a VO and then mapping VOs to specific policies.

The proposed approach is illustrated in Figure 4.3.

Initialization of CA When the CA is created, the VOs that are going to be supported are decided. For each of these VOs, a separate policy is

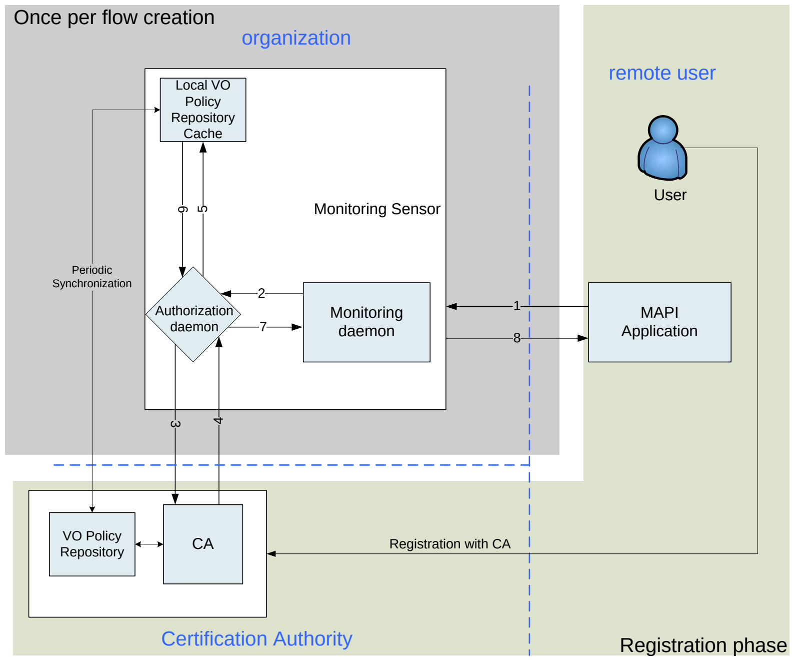


FIGURE 4.3: **Authorization architecture**

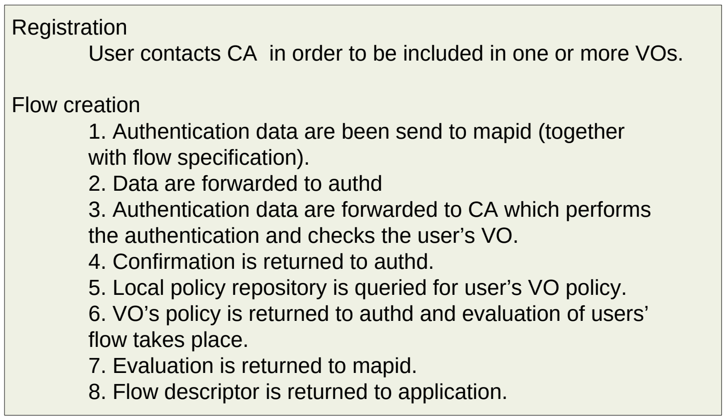
issued and stored in the VO Policy Repository. This policy will be the one that each sensor will expect users belonging to that VO to respect, unless the administrator overrides it (for example with a stricter one) specifically for his sensor. This process is performed only once, during the deployment of the MAPI infrastructure.

Registration phase: When a new user wants to use the MAPI infrastructure, the Certification Authority (CA) should be contacted. The user should provide the CA with his public key and the VOs he wants to be part of. This can be done for example by filling a form. The administrator of the CA will evaluate the request and if it is accepted, the public key of the user will be assigned to the VOs specified. At this point, the user is registered within the MAPI infrastructure and can access data from all sensors which support the Virtual Organizations the user is member of. We should note here that this process is preformed only once.

Flow creation phase: When the user creates a flow (step 1), the user has only to specify authorization data. Authorization data is the public key of the user as well as the challenge as it was presented in the previous section. Moreover, the user has to specify the VO he wants to use (in the case that the user belongs in more than one VOs). No credentials have to be sent to the sensor. The data (step 2) are forwarded to the authd which will perform the authentication and authorization steps. Authentication data are sent to the CA, which is responsible to authenticate the user and confirm that the user belongs to the VO he has specified. Confirmation and VO are sent back (step 3). Then, authd performs a lookup in the local policy repository (step 4), and retrieves (step 5) the policy for the specific VO. This step is performed for optimization. The policies for VOs are stored in the CA but it is an expensive process to query and retrieve it for each flow. Therefore we keep a local cache of the policies in the sensor which is synchronized periodically with the central repository. The credentials are returned to authd (step 6) which evaluates the specification of the flow for compliance with the supplied credentials (that are included in the policy), and returns the result of the evaluation (step 7) to mapid. If the flow specification complies with the credentials and the authentication is successful, mapid activates the flow and returns the relevant flow descriptor to the user (step 8). Otherwise the flow is rejected. The steps are summarized in Figure 4.4.

4.2.4 Advantages of the proposed approach

As mentioned in the introduction of this section, the goal of the proposed approach is to lower the administration cost in MAPI. In the current architecture, when a new user wants to use MAPI, he should contact all the sensors



Registration

User contacts CA in order to be included in one or more VOs.

Flow creation

1. Authentication data are been send to mapid (together with flow specification).
2. Data are forwarded to authd
3. Authentication data are forwarded to CA which performs the authentication and checks the user’s VO.
4. Confirmation is returned to authd.
5. Local policy repository is queried for user’s VO policy.
6. VO’s policy is returned to authd and evaluation of users’ flow takes place.
7. Evaluation is returned to mapid.
8. Flow descriptor is returned to application.

Figure 4.4: **Authentication and authorization steps in the proposed architecture**

and the administrator of each of them should issue credentials for him. As the number of users and sensors increases, this becomes a time consuming process for both the user and the sensor administrator. On the contrary, in the proposed architecture, the only action that should be performed for a new user is to register with the CA. Using the centralized architecture of CA, the user will issue just one request, will be included in one or more VOs and no actions are required by the administrators of each sensor. The process in both architectures is illustrated in Figure 4.5.

The same issues apply for the deletion of a user. In the proposed architecture, the user is directly deleted from the CA which is much simpler comparing to the current situation where each sensor should be contacted in order to delete the user.

Moreover, when a new sensor joins MAPI, each existing MAPI user should contact it in order to register and get credentials for the particular sensor. Considering that there will be a large number of users, this becomes a time consuming process for users but also for the administrator of the sensor. Using the proposed architecture, when a new sensor joins the infrastructure,

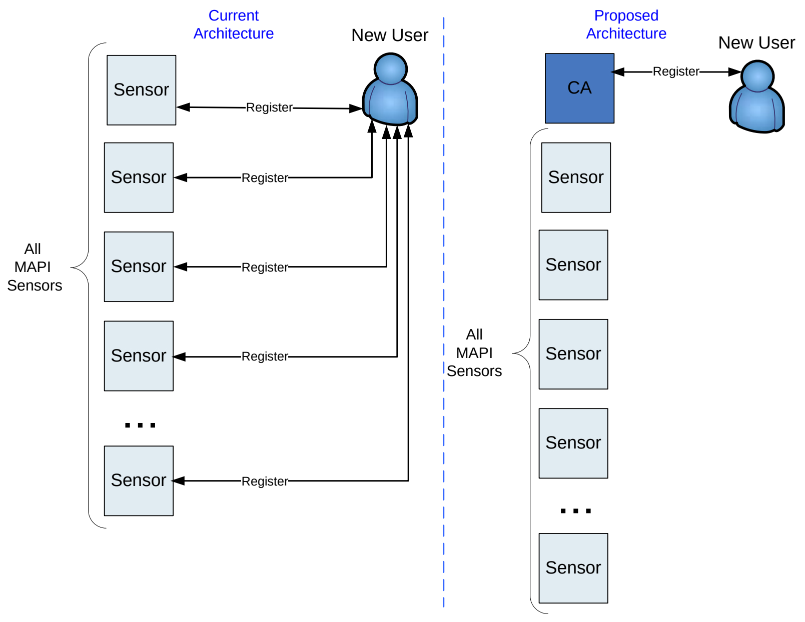


FIGURE 4.5: **A new user joins MAPI**

no further actions should be performed either from the users, or from the administrator. This process is shown in Figure 4.6.

The main simplification that the proposed new scheme provides is that the local administrator of each sensor no longer has to create separate policies for each user. Since the policies are created in the CA, the administrator no longer has to create policies at all. The only responsibility each administrator still has is to create an anonymization policy for each VO that wants to support in a different way than the one that the CA policy specifies. Since the number of VOs is much smaller than the number of users, we expect that the proposed approach to significantly lower the administrative cost. Moreover we should note that the definition of the anonymization policies happen only once, contrary to what happens in the current architecture. As mentioned in the previous chapter, the administrator of each sensor should issue credentials every time a new user wants to use the sensor, which leads to

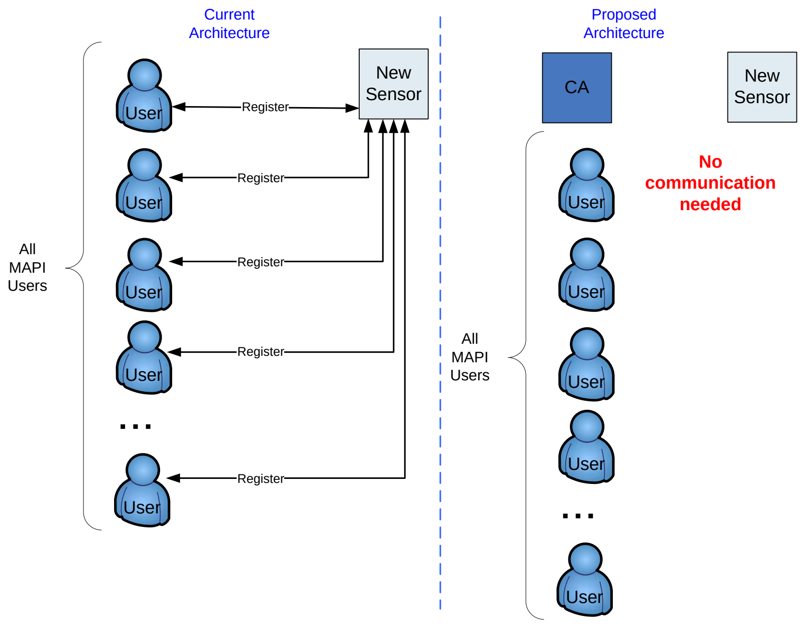


FIGURE 4.6: **A new sensor joins MAPI**

a considerable overhead as the number of users rise. This overhead is avoided using the proposed architecture, since the registration process happens only once in the centralized CA.

Finally, from the user's perspective, it is no longer necessary to deal with credentials during the creation of a flow. Only authentication data should be sent to the sensor, which include the public key and VO of the user. Moreover, since there is a single policy for a VO, no conflicts will appear when an application contacts multiple sensors.

To sum up, it is clear that the proposed architecture simplifies the entire authentication/authorization process, lowers the administrative cost needed and therefore the whole infrastructure scales efficiently with the number of users.

4.2.5 Anonymization as a transparent process

Taking advantage of this architecture it is possible make anonymization even more transparent to the user. According to the current design of MAPI, but also the one that is proposed above, the user should apply anonymization functions to each network flow in order to conform to the policy before (s)he is able to receive any information from the network flow. In order to make the usage of the MAPI infrastructure as simple as possible, it is possible to free the user from specifying the anonymization functions. This is possible since upon authentication of a user, the sensor ”knows” the anonymization policy for the particular user (or the particular VO the user belongs to).

Therefore, instead of asking the user to apply the anonymization functions and then check that they conform to a policy, an alternative approach would be to make the sensor responsible for the anonymization of the traffic. For example, the mapi daemon may create a virtual network device for each anonymization policy that the underlying sensor supports. For example, if the FORTH sensor has two policies for VOs (FORTHteam and FORTHothers) it may also create two virtual devices for each real monitoring sensor. For example, for network device eth0 there may also exist the devices eth0_FORTHteam and eth0_FORTHother. Each of these devices will provide the packets received from eth0, anonymized using the policy that is assigned to the FORTHteam and the FORTHother virtual organizations, respectively. This means that the appropriate anonymization functions will always be applied to packets before they are given to the user’s application.

When a user wants to use a MAPI sensor, he has just to specify the network device the user wants to open. After identifying the user and the VO (s)he belongs to, mapid may open the appropriate virtual network device for this user. This is feasible since the sensor knows the VO this user belongs to, and using the mapping from VO to policies (and therefore to virtual network devices) can open the right device for this user.

Let is illustrate the above using the following example. In order a user to access a MAPI sensor, according to the current authentication/anonymization scheme, one should write an application like the one shown in Figure 4.7.

```
1. For each sensor:
Register with sensor
2. Store credentials locally
3. Application

#create flow
fd = mapi_create_flow("eth0");

#apply anonymization
mapi_apply_function(fd,"ANONYMIZE", IP, SRC_IP, MAP);
mapi_apply_function(fd,"ANONYMIZE", IP, DST_IP, MAP);
mapi_apply_function(fd,"ANONYMIZE", IP, TTL, PATTERN_FILL,
     INTEGER, 64);
mapi_apply_function(fd,"ANONYMIZE", IP, ID, PATTERN_FILL,
     INTEGER, 242);
mapi_apply_function(fd,"ANONYMIZE", TCP, PAYLOAD, STRIP, 0);
mapi_apply_function(fd,"ANONYMIZE", UDP, PAYLOAD,STRIP, 0);
mapi_apply_function(fd,"ANONYMIZE", HTTP, URI, FILENAME_RANDOM);
mapi_apply_function(fd,"ANONYMIZE", HTTP, HOST, REPLACE,
     "WWW_SERVER");
mapi_apply_function(fd,"ANONYMIZE", IP, CHECKSUM,
     CHECKSUM_ADJUST);

#set authentication data
mapi_set_authdata(fd, "public_key", "private_key", "credentials");

#connect
mapi_connect(fd) ;
```

FIGURE 4.7: **Current anonymization approach**

If we use the scheme proposed in this document the same application can be written as presented in Figure 4.8.

```
1. Register with CA (single step)
2. Application

#create flow
fd = mapi_create_flow("eth0");

#set authentication data
mapi_set_authdata(fd,"public_key","private_key","WormTeam");

#connect
mapi_connect(fd) ;
```

FIGURE 4.8: **Proposed anonymization approach**

We can see that in the second case, the program is much simpler. Apart from this, the user no longer deals with anonymization functions. The code dealing with anonymization that has ”disappeared” from the application, as it is automatically applied by the mapid.

Chapter 5

Performance Evaluation

In this chapter we evaluate the performance of the anonymization API. First, we compare the performance of our framework against existing anonymization tools. Then, we evaluate the cost of individual anonymization functions and how much the framework benefits from from several performance optimizations. Finally, we discuss the usefulness of the produced traces.

5.1 Comparison with existing tools

In this section we compare the performance of AAPI with the existing anonymization tools, tcpdpriv and Bro.

5.1.1 Tcpdpriv

In order to measure the performance of the framework, we have implemented some simple anonymization policies both as AAPI applications, and `tcpdpriv` processes. One simple anonymization policy we have implemented states that: "*IP addresses are mapped to sequential integers, IP and TCP options fields are set to zero, the TCP/UDP payload is also zeroed, while the packet's checksums are updated*". This policy is similar to those that were pre-

policy	*tcpdpriv (sec)*	*AAPI-based tool (sec)*
MAP	10.78	7.41
Prefix-Preserving	10.85	9.39
MAP, no checksum	6.83	6.67

Table 5.1: Performance comparison between tcpdpriv and AAPI-based anonymization.

sented in section 2.3. Also, in order to explore the performance of prefix preserving anonymization schemes, which are commonly proposed for anonymizing IP addresses, we also applied the PREFIX_PRESERVING_MAP function instead of the sequential mapping to IP addresses defined in the original policy.

AAPI allows us to implement the anonymization functionality in less than 40 lines of code as shown in Appendix C. In the experiment we used a Intel Pentium 4, running at 3.0 GHz with 512 MB main memory. We used a 2 GB tcpdump trace (approximately 2.6 million packets) as traffic input for anonymization, which consisted exclusively of TCP packets from a web portal mirroring. In Table 5.1 we show the user time[1] in seconds for both the AAPI-based aplication and `tcpdpriv`. As it can be observed, our tool is marginally faster than `tcpdpriv`. The main reason for this difference is the poor implementation of checksum fix on `tcpdpriv`. If we remove the checksum fix functionality from both applications, their performance is equivalent. Note that this result is indicative of the performance of AAPI-based tool since `tcpdpriv` is a highly specialized tool for simple anonymization policies

[1]Only user time is measured since all tools including AAPI use the standard libpcap [3] library for packet capturing. Therefore the overhead is the same for all tools and thus not taken into account.

policy	*Bro (sec)*	*AAPI-based tool (sec)*
MAP	133.00	4.35
URL replace	134.48	58.85

Table 5.2: Performance comparison between Bro and AAPI-based anonymization.

without offering all the functionality supported by AAPI.

5.1.2 Bro

We compared AAPI with the `Bro` system, which, in contrast to `tcpdpriv`, has support for application-level anonymization. We conducted two experiments. In the first, we implemented the same policy that we mentioned above that also involves mapping the IP addresses to integers. In the second experiment we implemented a policy that required application-level anonymization: "*replace the URL in HTTP packets with the string SAMPLE_URL*". In Table 5.2 we present the measured user time (in seconds) to complete the anonymization of the same trace as in the previous expreriment.

Due to the limitations of `Bro`, as discussed in section 2.2, even for a simple action like changing the IP address in a packet, it has to reassemble the trace up to the HTTP level before it can anonymize the IP addresses. In the case of mapping IP addresses, our approach is up to 30 times faster than `Bro`, as we do not have to perform stream reassembly. In the case of URL replacement, where both tools need to perform cooking, the AAPI application needs about half the time required by `Bro`. AAPI is faster because it is a framework specially designed for anonymization, in contrast with `Bro`, which is an IDS system with additional functionality useless to anonymization which introduces this overhead.

5.2 The Cost of Anonymization Functions

Having a complex anonymization policy with a long list of anonymization functions does not come for free. In some cases, like simply setting the Time-To-Leave (TTL) to zero, an anonymization function may be a fairly lightweight process. On the other hand, some functions like "COOK" can be very slow. So it is clear that the anonymization can be a time consuming procedure. While for the non-realtime anonymization of traffic traces this performance may not be an issue, the on-line anonymization of realtime traffic has to be as fast as possible in order to keep up with the incoming traffic from the high-speed Gigabit links.

Infrastructures that work with live traffic from diverse administrative domains, such as zero-day worm detection systems [6, 38], demonstrate the need for real-time anonymization. It is highly unlikely that organizations would share their traffic without first anonymizing it. In order to have a view of the cost of the various anonymization functions, in this section we try to quantify this cost by testing the most commonly used functions and various combinations of these functions. This will give an insight of what is the performance penalty of each one of them. All experiments were performed on a PC with an Intel Pentium 4, running at 3.0 GHz, with 512 MB main memory. As input source we used the 2 GB tcpdump trace presented in previous experiments.

Our results are summarized in Table 5.3. The metric we use is user time, measured with the `time` command-line utility.

The prefix-preserving function is based on the Crypto-PAn package, while prefix-preserving-map[2] is a much simpler algorithm for prefix-preserving anonymiza-

[2]Prefix-preserving-map performs prefix-preserving anonymization using a mapping table instead of cryptography

Function	*User time (sec)*
Set TTL and IPid to constant	3.304
Map src/dst IP	4.356
Map IPs, set TTL and IPid constant	5.152
Prefix-preserving-map src/dst IP	7.068
No cooking,randomize URL	6.060
Map src/dst IP,randomize URL, checksum adjust	12.777
Cooking	19.812
Cooking, URL replace, uncooking	28.580
Prefix-preserving src/dst IP	87.721

Table 5.3: Cost of basic anonymization functions

tion without using cryptographic methods. The prefix-preserving function based on the Crypto-PAn package results in low performance. The optimization of this function is left for future work.

As we can see, apart from cooking and cryptographic prefix-preserving, most functions do not impose a large overhead and therefore can be used also for on-line anonymization. It is also feasible for more complex anonymization policies such as those that are presented in section 2.3 to be applied on-line.

5.2.1 Complexity analysis

Most anonymization functions are linear to the size of data that they process. For example a hash function that is applied on payload is linear to the size of payload. Mapping functions' complexity is O(N/M) where N is the size of input and M the size of the hash table which is used. The prefix-preserving function does not depend on the size of data since it is applied only on IP addresses therefore it's complexity is O(1) but including a big constant beacuse of cryptography. Finally the COOK/UNCOOK function's is linear to the number and size of packets that are processed.

5.3 Optimizing Anonymization

In this section we evaluate the performance enhancement delivered from optimization. It should be noted that since optimization is performed (as noted in section 3.5) when different anonymization sets share common subset of anonymization functions, this feature in not of great interest for applications made with AAPI. A typical application most of the times will include only a small number of sets. On the other hand, in on-line systems that need to perform anonymization and deliver traffic to multiple users with different needs, like MAPI, optimization is necessary. This means multiple flows with common subset of functions and an opportunity for optimization.

In the first experiment, we created a rather "heavy" policy with multiple anonymization functions and applied it to all user flows. We anonymized a 400MB network trace (approximately 1.7 million packets , 75% TCP) while increasing the number of concurrent users. We performed this experiment both with and without optimization. In Figure 5.1 we present the time needed in order to anonymize the trace against the number of concurrent users. We can see that for 10 concurrent users, the time needed when using optimization is one quarter of the time in the non-optimized case.

In the previous experiment we assumed that there was only a single anonymization policy for all users. A more realistic scenario is that there is a certain number of policies and users are mapped in one of them (as described in section 4.2.3). In this experiment we defined three policies and

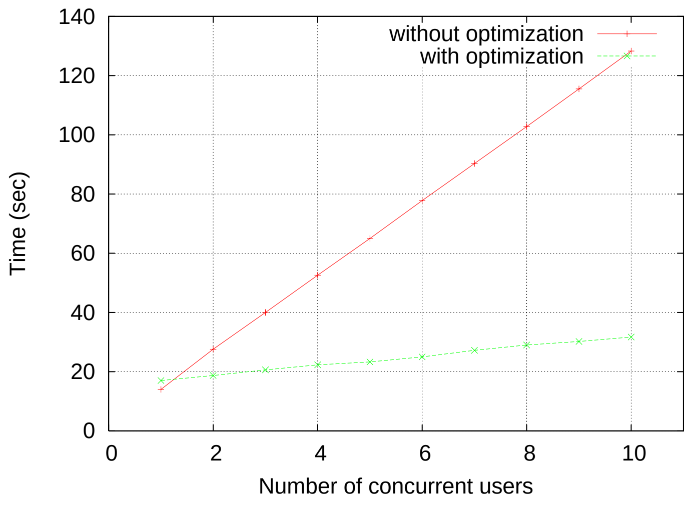


FIGURE 5.1: **Optimization for a single policy.**

when each user connected to the sensor, we chose randomly one of them. We used the same trace as in the previous experiment. In Figure 5.2 we present the time needed for anonymization against the number of concurrent users.

We can see that for a small number of users, enabling optimization increases the time needed. This is due to the fact that optimization itself introduces an unavoidable overhead in order to be applied, as it introduces extra copies of packets. Since the number of users is the same as the number of policies it is clear that this is not a case for optimization. As the number of users increases, we can see the performance enhancement. In on-line systems, the number of users is expected to be orders of magnitude larger than the number of policies (as desribed in chapter 4), so the performance overhead will not appear.

5.4 Usefulness of Anonymized Traces

The anonymization policy defines the level of information hiding on traffic. On one hand, we need anonymization which “hides” information, while on

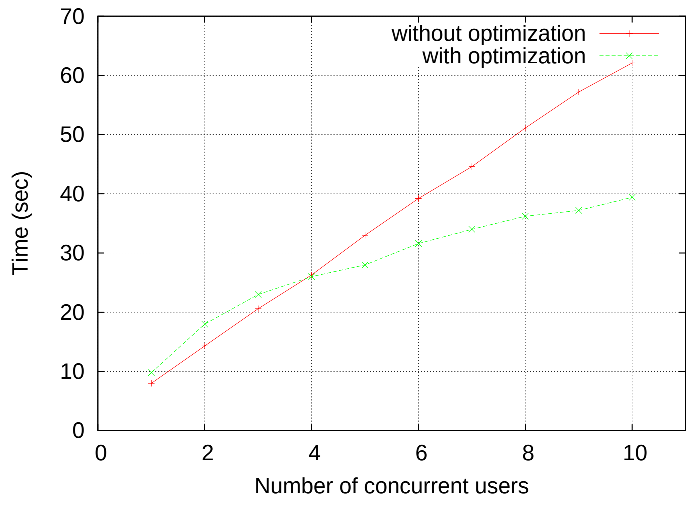


FIGURE 5.2: **Optimization for three policies.**

the other hand, the information we delete from a trace decreases its usefulness in terms of the characteristics that researchers can find within that trace. In this section we will look into this trade-off. We assume that the more flexible anonymization policy and the more fine-grained it is, we get the most "useful" trace, with the minimum lost information.

Existing tools do not provide enough flexibility for fine-grained policies, thus their output is used in limited cases. The goal of the following experiments is to demonstrate that our approach allows for fine-grained anonymization policies that are able to produce output which hides as little information as possible. The policy applied is "*prefix-preserving anonymization of IP address, set the TTL and IP identification number to constants, removal of the HTTP payload — but not of HTTP headers*". The metric of usefulness that we use is the number of alerts that were generated by the Snort [37] intrusion detection system. Our input was a 400 MB trace (approximately 1.7 million packets), which was collected during the DARPA evaluation test [21].

We anonymized the trace with both `tcpdpriv` and with a simple application based on AAPI. Because of its flexibility, AAPI can express the policy described above, in contrast with `tcpdpriv` that has only a hard coded predefined policy. We passed both output traces from the Snort IDS [37] and counted the number of alerts generated. We ran Snort with two different sets of rules: one that contains header-only rules, and one with content rules, i.e., rules that require access to both headers and payload. The results are presented in Table 5.4. The last row denotes the sum of the two cases.

	tcpdpriv	*AAPI*	*Real trace*
header-only rules	45 (100%)	45 (100%)	45
content rules	0 (0%)	527 (28%)	1892
complete snort ruleset	45 (2%)	572 (30%)	1937

TABLE 5.4: Number of alerts produced by Snort IDS for web trace

As we can see, `tcpdpriv` preserves only a small percentage on the initial alerts, derived solely from header rules. AAPI on the other hand, uses a less strict approach that creates a much more useful trace. We performed the same experiment with other traces (from the same evaluation test) and results where almost the same. We should note that the results depend on the number of the alerts that the trace contains and the ratio between header and content rules.

It is clear that this example is not realistic since the HTTP header may contain sensitive information, for example URI or host fields, that should be anonymized. Using AAPI, one could create a policy that does not anonymize packets which contain attacks and anonymize those that do not. This way, alerts will be preserved in the output trace and at the same time no private information will be revealed since all other packets will be anonymized. Approaches with other tools are more rough, as in the case of content rules in which the output trace could not produce any alerts. Even if we change the policy, `tcpdpriv` will still generate zero alerts as it sets the payload to zero.

This experiment is indicative for the flexibility of AAPI, showing that it enables the user to balance between usefullnes and privacy and not restricting him to a hard coded policy, as tcpdpriv does.

Chapter 6

Summary and Concluding Remarks

We have presented the design of a generic network traffic anonymization framework. Its key point is configurability, which allows the user to define any anonymization policy as a series of functions that are applied on packets. The main design goal is to facilitate the development of custom anonymization tools, that are able to implement both simple and complex policies in only a few lines of simple code. AAPI is able to let the user balance privacy and usefuness, and therefore the value of the output trace depends solely on the decisions of the user and the anonymization policy that is defined and is not addressed in this work.

The major advantage of our framework is that it works up to application-level, offering a large set of anonymization primitives and in the same time trying to optimize the necessary functions. All in all, to our knowledge this work currently constitutes the most complete framework for anonymization of realtime traffic and offline traces. Furthermore, the framework is implemented in a modular way which makes it fully extensible in terms of functionality, protocols and new traffic sources.

Through the integration of AAPI with a network monitoring infrastructure, we realized the need for a mechanism to enforce anonymization policies transparently to users. The proposed architecture targets a distributed version of MAPI, and simplifies the whole process of authentication/authorization. Moreover, taking advantage of the design, anonymization becomes a transparent process to the user since the appropriate policy is applied by the sensor.

We evaluated the performance of our anonymization primitives and their combination. Our results have shown that in most commonly used policies, AAPI outperforms existing similar applications, which offer only a subset of the AAPI functionality. Furthermore, as a result of the framework's flexibility and expressiveness, we show that output traces can preserve much more useful information compared to existing tools, without any sensitive data leakage.

Future works on anonymization should target on the development and support of more application level protocols, such as SMTP or DNS. Moreover, the performance of some anonymization functions should be improved. A possible solution would be a hardware implementation that cooperates with AAPI. Finally, a study should be performed on evaluation of anonymization policies, in terms of the usefulness of output trace and private data. This can be used as a guide for the users that want to share network monitoring data, help them find the appropriate balance between privacy and usefulness.

Appendix A

Predefined Protocol Field Names

The following is the list of predefined names that can be used as the `field_description` parameter:

- **Common to all protocols**: PAYLOAD
- **Common to IP, TCP, UDP, ICMP**: CHECKSUM
- **IP**: SRC_IP, DST_IP, TTL, TOS, ID, IP_PROTO, VERSION, IHL, OPTIONS, FRAGMENT_OFFSET, PACKET_LENGTH
- **Common to TCP and UDP**: SRC_PORT, DST_PORT
- **TCP**: SEQUENCE_NUMBER, ACK_NUMBER, FLAGS, WINDOW, TCP_OPTIONS, URGENT_POINTER, OFFSET_AND_RESERVED
- **UDP**: UDP_DATAGRAM_LENGTH
- **ICMP**: TYPE, CODE

- **HTTP**: HTTP_VERSION, METHOD, URI, USER_AGENT, ACCEPT, ACCEPT_CHARSET, ACCEPT_ENCODING, ACCEPT_LANGUAGE, ACCEPT_RANGES, AGE, ALLOW, AUTHORIZATION,CACHE-CONTROL, CONNECTION_TYPE, CONTENT_ENCODING, CONTENT_TYPE, CONTENT_LENGTH, CONTENT_LOCATION, CONTENT_MD5, CONTENT_RANGE, COOKIE, DATE, ETAG, EXPECT, EXPIRES, FROM . HOST, IF_MATCH, IF_MODIFIED_SINCE, IF_NONE_MATCH, IF_RANGE, IF_UNMODIFIED_SINCE, LAST_MODIFIED, LOCATION, KEEP_ALIVE, MAX_FORWRDS, PRAGMA, RANGE, REFERRER, RETRY_AFTER, SET_COOKIE, SERVER, TE, TRAILER, TRANSFER_ENCODING, UPGRADE, USER_AGENT, VARY, VIA, WARNING, WWW_AUTHENTICATE, X_POWERED_BY, RESPONSE_CODE, PROXY_AUTHENTICATE, PROXY_AUTHORIZATION, RESP_CODE_DESCR

- **FTP**: USER, PASS, ACCT, FTP_TYPE, STRU, MODE, CWD, PWD, CDUP, PASV, RETR, REST, PORT, LIST, NLST, QUIT, SYST, STAT, HELP, NOOP, STOR, APPE, STOU, ALLO, MKD, RMD, DELE, RNFR, RNTO, SITE, FTP_RESPONSE_CODE, FTP_RESPONSE_ARG

Appendix B

Anonymization Functions

The following is the complete list of useful functions that could be applied to the various protocol fields.

- **UNCHANGED**: leaves field unchanged. This function takes no arguments.

- **MAP**: maps a field to an integer. Each field will have different mapping except SRC_IP and DST_IP which share common mapping as well as SRC_PORT and DST_PORT. The rest of the fields share a common mapping based on their length: fields with length 4 have a common mapping, fields with length 2 have their own and finally fields with length 1 share their own mapping. Mapping cannot be applied to payload and IP/TCP options, only in header fields. This function takes no arguments.

- **MAP_DISTRIBUTION**: field is replaced by a value extracted from a distribution like uniform or Gaussian, with user-supplied parameters. The first parameter defines the type of distribution and can be UNIFORM or GAUSSIAN. If type is UNIFORM the next 2 arguments specify the range inside which the distribution selects uniformly numbers. If type is GAUSSIAN the next 2 arguments specify the median and standard deviation. Similarly to MAP function, MAP_DISTRIBUTION can only be applied to IP, TCP, UDP and ICMP header fields, except IP and TCP options.

- **STRIP**: removes the field from the packet. Optionally, STRIP may not remove the whole field but can keep a portion of it. The user defines the number of bytes to be kept. STRIP cannot be applied to IP, TCP, UDP and ICMP headers except IP and TCP options and can be fully applied to all HTTP and FTP fields.

- **RANDOM**: replaces the field with a random number. This function takes no arguments.

- **FILENAME_RANDOM**: a sub-case of RANDOM. If the field is in a filename format, e.g. "picture.bmp" then the extension is left untouched while the filename is replaced by random characters.

- **HASH**: field is replaced by a hash value. Supported hash functions are MD5, SHA, SHA_2, CRC32 and AES and TRIPLE_DES for encryption. Note that MD5, SHA, SHA_2 and CRC32 may generate values with less or greater length than the original field. The hash functions when applied to IP, TCP, UDP and ICMP header fields, their last bytes are used to replace the field. For all the other fields, the padding behavior is supplied as a parameter. If the hashed value has less length, the user can pad the rest bytes with zero by defining PAD_WITH_ZERO or can strip the remaining bytes by defining STRIP_REST as an argument to the function. If the hashed values has length greater than the original field, then the rest of packet contents are shifted accordingly. In all cases, the packet length in protocol headers is adjusted to the new length.

- **PATTERN_FILL**: field is repeatedly filled with a pattern . The pattern can be an integer or string. This function takes as a parameter the type of pattern, INTEGER for integer and STR for strings, and the pattern to be used.
- **ZERO**: a sub-case of pattern fill where field is set to zero. This function takes no arguments
- **REPLACE**: field is replaced by a single value (a string). The packet length is reduced accordingly, based on the length of the replace pattern. The final length cannot exceed the maximum packet size. This function takes the pattern to be used as an argument.
- **PREFIX_PRESERVING**: can only be applied to source and destination IP addresses and performs a key-hashing, preserving the prefixes of IP addresses.
- **PREFIX_PRESERVING_MAP**: can only be applied to source and destination IP addresses and performs a preserving the prefixes of IP addresses using mapping table.
- **REGEXP**: field is transformed according to regular expression. As an example, performing anonymize(p, TCP, PAYLOAD, REGEXP, "(.*) password:(.*) (.*)",NULL,"xxxxx",NULL) in a packet p we can substitute the value of a "password:" field with the "xxxxx" string. Each "(.*)" in the regular expression indicates a match and the last argument is a set of replacements for each match (NULL leaves match unmodified).

- **CHECKSUM_ADJUST**: if we want the anonymized packet stream to be used by other applications, the anonymization modifications to each packet requires careful treatment of the checksum. This function can be only applied to CHECKSUM field.

- **SUBFIELD**: with this function we can apply any of the functions defined above to a *subfield* of the given field. Therefore the arguments of **SUBFIELD** are the two offsets over the identified protocol field, which are the bounds of the subfield, followed by any of the above field anonymization functions with their parameters. The identified field anonymization function which is passed as parameter to **SUBFIELD** will be applied to the *subfield* that is bounded by the given offsets.

Appendix C

Sample application using AAPI

In this example, using AAPI, we create an application with similar functionality to tcpdpriv [16].

```
#include "aapi.h"

int main(int argc, char *argv[]) {

    int sd;

    if (argc < 3)
    {
        fprintf(stderr, "Usage %s input output\n", argv[0]);
        return -1;
    }

    //create sets
    sd1 = create_set();
    sd2 = create_set();

    //set traffic source
    set_source(TCPDUMP_TRACE, argv[1]);
    //set output
    set_output(sd1, TCPDUMP_TRACE, argv[2]);
    set_output(sd2, TCPDUMP_TRACE, argv[2]);
    //anonymization for tcp/udp traffic, map IPs
    //zero TCP/IP options & TCP/UDP paylaod, fix checksums
    add_function(sd1, "BPF_FILTER", "tcp or udp");
    add_function(sd1, "ANONYMIZE", IP, SRC_IP, MAP);
    add_function(sd1, "ANONYMIZE", IP, DST_IP, MAP);
    add_function(sd1, "ANONYMIZE", IP, OPTIONS, ZERO);
    add_function(sd1, "ANONYMIZE", TCP, TCP_OPTIONS, ZERO);
    add_function(sd1, "ANONYMIZE", TCP, PAYLOAD, ZERO);
    add_function(sd1, "ANONYMIZE", UDP, PAYLOAD, ZERO);
    add_function(sd1, "ANONYMIZE", IP, CHECKSUM, CHECKSUM_ADJUST);

    //anonymization for all other IP packets, map IPs
    //zero IP options & IP payload, fix checksum
    add_function(sd2, "BPF_FILTER", "ip and not (tcp or udp)");
    add_function(sd2, "ANONYMIZE", IP, SRC_IP, MAP);
    add_function(sd2, "ANONYMIZE", IP, DST_IP, MAP);
    add_function(sd2, "ANONYMIZE", IP, OPTIONS, ZERO);
    add_function(sd2, "ANONYMIZE", IP, PAYLOAD, ZERO);
    add_function(sd2, "ANONYMIZE", IP, CHECKSUM, CHECKSUM_ADJUST);

    //start processing packets
    start_processing();
    return 1;

}
```

The AAPI program is about 40 lines of code while the tcdpriv code is about 2000 lines.

Appendix D

Anonymization using MAPI

This is a simple application that shows some basic anonymization features of MAPI.

```
#include <stdio.h>
#include <mapi.h>

int main(int argc, char *argv[]) {

    int fd;

    fd=mapi_create_flow("eth0");
    if(fd==-1) {
        printf("Flow cannot be created. Exiting..\n");
        exit(-1);
    }

    //Anonymization of TCP packets

    mapi_apply_function(fd,"BPF_FILTER","tcp");

    //map IP addresses to sequential integers (1-to-1 mapping)
    mapi_apply_function(fd,"ANONYMIZE", IP, SRC_IP, MAP);
    mapi_apply_function(fd,"ANONYMIZE", IP, DST_IP, MAP);

    //replace with zero, tcp and ip options
    mapi_apply_function(fd,"ANONYMIZE", IP, OPTIONS, ZERO);
    mapi_apply_function(fd,"ANONYMIZE", TCP, TCP_OPTIONS, ZERO);
    //remove payload
    mapi_apply_function(fd,"ANONYMIZE", TCP, PAYLOAD, STRIP, 0);
    //checksum fix in IP fixes checksums in TCP and UDP as well
    mapi_apply_function(fd,"ANONYMIZE", IP,
     CHECKSUM, CHECKSUM_ADJUST);
    mapi_apply_function(fd, "TO_BUFFER");

    /* connect to the flow */
    connect_status = mapi_connect(fd);
    if(connect_status < 0)
    {
        printf("Connect failed");
        exit(0);
    }

    while(1) /*forever, wait for matching packets */
    {
        pkt = mapi_get_next_pkt(fd, fid);
        do_something_with_packet(pkt);
    }

    return 0;

}
```

In the above example, we create a network flow that captures only TCP packets. Then we apply anonymization on IP addresses, TCP/IP options, TCP payload and finally we fix TCP/IP checksums.

Bibliography

[1] IST-SCAMPI project. `http://www.ist-scampi.org`.

[2] MAPI official homepage. `http://mapi.uninett.no/`.

[3] Tcpdump/libpcap official site. `http://www.tcpdump.org`.

[4] The Click Modular Router Project. http://www.pdos.lcs.mit.edu/click/.

[5] Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the Internet Measurement Workshop (IMW)*, 2002.

[6] P. Akritidis, K. Anagnostakis, and E. Markatos. Efficient content-based fingerprinting of zero-day worms. In *Proceedings of the International Conference on Communications (ICC 2005)*, May 2005.

[7] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote trust-management system version 2. *Network Working Group, RFC 2704*, Sept. 1999.

[8] Cisco Systems, Inc. Netflow. `http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml`.

[9] College of Computing, Georgia Tech. Cryptography-based Prefix-preserving Anonymization. `http://www.cc.gatech.edu/computing/Telecomm/cryptopan`.

[10] T. S. Consortium. Scampi architecture and components: Scampi deliverable d1.2. `http://www.ist-scampi.org`.

[11] Dartmouth College. Archive of wireless-network trace data. `http://cmc.cs.dartmouth.edu/data/index.html`.

[12] Dave Plonka. ip2anonip. `http://dave.plonka.us/ip2anonip`.

[13] Eddie Kohler. ipsumdump. `http://www.cs.ucla.edu/~kohler/ipsumdump`.

[14] Endace. DAG Network Monitoring Interface Card. `http://www.endace.com/networkMCards.htm`.

[15] M. J. Freedman, E. Sit, J. Cates, and R. Morris. Introducing tarzan, a peer-to-peer anonymizing network layer. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, Cambridge, MA, March 2002.

[16] Greg Minshall. Tcpdpriv. `http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html`.

[17] H. Jiang and C. Dovrolis. Passive estimation of tcp round-trip times. *Computer Communications Review*, July 2002.

[18] S. Jin and A. Bestavros. Sources and characteristics of web temporal locality. In *MASCOTS*, pages 28–35, 2000.

[19] Katherine Deibel and Andrew Petersen and Andrew Schwerin. Cone of Silence: A Layered Approach for Network-level Protocol Anonymization. `http://www.cs.washington.edu/homes/deibel/papers/cse561-cos/cse561-cos.pdf`.

[20] D. Koukis, S. Antonatos, D. Antoniades, P. Trimintzios, and E. Markatos. A generic anonymization framework for network traffic. In *To appear in In Proceedings of the IEEE International Conference on Communications (ICC 2006)*, 2006.

[21] M. L. Laboratory. Darpa intrusion detection evaluation data sets. `http://www.ll.mit.edu/IST/ideval/data/data_index.html`.

[22] Lawrence Berkeley National Laboratory. Bro Intrusion Detection System. `http://www.bro-ids.org`.

[23] LBL. LBL Internet Traffic Archive. `http://ita.ee.lbl.gov/html/traces.html`.

[24] LBNL/ICSI Enterprise Tracing Project. tcpmkpub. `http://www.icir.org/enterprise-tracing/tcpmkpub.html`.

[25] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM Computer Communication Review*, 27(3), July 1997.

[26] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

[27] J. Mogul. Trace anonymization misses the point. 2002.

[28] NLANR. PMA NLANR traces. `http://pma.nlanr.net/PMA/Traces`.

[29] R. Pang, M. Allman, V. Paxson, and J. Lee. The devil and packet trace anonymization, January 2006.

[30] R. Pang and V. Paxson. A High-Level Programming Environment for Packet Trace Anonymization and Transformation. In *Proceedings of the ACM SIGCOMM Conference*, August 2003.

[31] V. Paxson and S. Floyd. Wide-area traffic: the failure of Poisson modeling. pages 257–268. August 1994.

[32] M. Peuhkuri. A method to compress and anonymize packet traces. *Internet Measurement Workshop (San Francisco, California, USA: 2001)*, pages 257–261, 2001.

[33] M. Polychronakis, K. G. Anagnostakis, E. P. Markatos, and A. Øslebø. Design of an application programming interface for ip network monitoring. In *Proceedings of the 9th IEEE/IFIP Network Operations and Management Symposium (NOMS2004)*, April 2004.

[34] Rafal Wojtczuk. Libnids. `http://libnids.sourceforge.net/`.

[35] R. Ramaswamy, N. Weng, and T. Wolf. An IXA-based network measurement node. In *Proc. of Intel IXA University Summit*, Hudson, MA, Sept. 2004.

[36] M. K. Reiter and A. D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[37] M. Roesch. Snort: Lightweight intrusion detection for networks. November 1999. (available from *http://www.snort.org/*).

[38] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation*, Dec. 2004.

[39] A. Slagell, Y. Li, and K. Luo. Sharing network logs for computer forensics: A new tool for the anonymization of netflow records. *Computer Network Forensics Research (CNFR) Workshop*, 2005.

[40] Q. Sun, D. Simon, Y. Wang, W. Russell, V. Padmanabhan, and L. Qiu. Statistical identification of encrypted web browsing traffic. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2002.

[41] P. Trimintzios, M. Polychronakis, A. Papadogiannakis, M. Foukarakis, E. P. Markatos, and A. Øslebø. Dimapi: An application programming interface for distributed network monitoring. In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Apr. 2006.

[42] UCLA. UCLA CSD Packet Traces. `http://lever.cs.ucla.edu/ddos/traces/`.

[43] UCSD. Wireless LAN Traces. `http://ramp.ucsd.edu/pawn/sigcomm-trace/`.

[44] A. S. J. Wang and W. Yurcik. Network log anonymization: Application of crypto-pan to cisco netflows. *NSF/AFRL Workshop on Secure Knowledge Management (SKM)*, 2004.

[45] J. Xu, J. Fan, M. Ammar, and S. B. Moon. On the design and performance of prefix-preserving ip traffic trace anonymization. *Internet Measurement Workshop (San Francisco, CA, USA: 2001)*, pages 263–266, 2001.

[46] J. Xu, J. Fan, M. Ammar, and S. B. Moon. Prefix-preserving ip address anonymization: Measurement-based security evaluation and a new cryptography-based scheme. *ICNP 2002*, 2002.

[47] T. Ylonen. Thoughts on how to mount an attack on tcpdpriv's "-a50" option. http://ita.ee.lbl.gov/html/contrib/attack50/attack50.html.