

INFORMATION SOCIETY TECHNOLOGIES (IST)
PROGRAMME



Large Scale Monitoring of BroadBand Internet Infrastructure
Contract No. 004336

D1.1a: Anonymization Framework Definition

Abstract: This document presents the complete a framework for applying anonymization policies within the LOBSTER infrastructure. This includes an extended review of the state-of-the-art in most frequently used anonymization policies and tools, a complete anonymization architecture, the related protocol field anonymization functions, an anonymization policy editor tool, and a list of example anonymization policies.

Contractual Date of Delivery	30 June 2005
Actual Date of Delivery	5 July 2005
Deliverable Security Class	Public

The LOBSTER Consortium consists of:

FORTH	Coordinator	Greece
VU	Principal Contractor	The Netherlands
CESNET	Principal Contractor	Czech Republic
UNINETT	Principal Contractor	Norway
ENDACE	Principal Contractor	United Kingdom
ALCATEL	Principal Contractor	France
FORTHnet	Principal Contractor	Greece
TNO	Principal Contractor	The Netherlands
TERENA	Principal Contractor	The Netherlands

Contents

1	Introduction	7
2	State of the Art in Traffic Anonymization and the Related Policies	9
2.1	Most Frequently Used Anonymization Policies	9
2.2	Anonymization Tools	10
2.3	Related Research Work	12
3	Anonymization Architecture	15
3.1	Admission Control in MAPI	15
3.2	Anonymization Infrastructure Overview	16
3.3	Anonymization Mechanisms within a Sensor	18
3.3.1	Anonymization as a Transparent Process	20
3.3.2	Function Reordering	20
4	Defining Anonymization Policies	23
4.1	Anonymization Policy Editor Tool	23
4.2	Guidelines for Policy Preparation	25
4.2.1	Examples	27
4.3	Security Implications of the Anonymization Functions	29
5	The SiSaL Scripting Sanitization Language	31
6	Summary	35
A	Predefined Protocol Field Names	37
B	Complete List of the Protocol Field Anonymization Functions	39
C	Implementation of the NLANR Anonymization Policy in LOBSTER	43
D	Implementation of the Dartmouth Anonymization Policy in LOBSTER	45
E	Implementation of the University of California, San Diego Anonymization Policy in LOBSTER	47

CONTENTS

F Implementation of the LBL HTTP and FTP Anonymization Policy in LOBSTER	49
---	-----------

List of Figures

3.1	Anonymization framework in LOBSTER	17
3.2	Reordering of Anonymization Functions	21
4.1	Anonymization Policy Editor tool	24
4.2	Keynote code generated by the tool	25
4.3	MAPI code generated by the tool	26

LIST OF FIGURES

Chapter 1

Introduction

Monitored network traffic, in the ideal case, should be shared unchanged in order to provide full information to network-based monitoring applications, network administrators, as well as security scientists. However, for *security*, *privacy*, and *business competition* reasons monitored traffic is usually modified before it becomes publicly available. This modification is known as **network traffic anonymization** process.

Since LOBSTER is an infrastructure that promotes sharing for both network packets and statistics, a framework for handling anonymization process is necessary. The amount of information to be hidden by the anonymization process is specified by, as accurate as possible, policies.

Lack of trust among collaborating parties may turn out to be the single most important obstacle to the wide dissemination of passive network monitoring across infrastructures such as LOBSTER. Indeed, although different parties want to collaborate in order to identify new security attacks, they may be reluctant to exchange their data in raw form. To address privacy concerns related to the exchange of raw data, network traces need to be anonymized before they are presented to (local or remote) monitoring applications. In addition, different monitoring applications may have different anonymization needs. For example, during the time of a crisis, the chief network administrator of an Internet Service Provider (ISP) may have access to full header and payload information of all suspect packets in order to trace the origin of a cyberattack. On the other hand, the average network administrator may have only access to encrypted data that allows him/her to monitor the network without jeopardizing the privacy of the users.

To accommodate all needs, we need a flexible way to anonymize the network packets according to the privileges and requirements of user applications. Such a flexibility can be achieved with the introduction of *user-specific* and *flow-specific* anonymization policies. The aim of this document is to describe a complete proposal for a framework for creating and applying anonymization policies within the LOBSTER infrastructure.

The goal of anonymization process is double. The first goal is to protect the privacy of monitored users. Revealing sensitive information about the users is totally unacceptable. Examples of such information are which web pages a user accessed, credit card numbers, unencrypted sessions that might reveal passwords, peer-to-peer connections, e-mail sends and receives, etc. Our framework provides all necessary means to remove or hide all sensitive information from network traffic. The second goal is to hide information about the internal infrastructure of the network. Ideally, an anonymized trace should not reveal which hosts inside the monitored network are alive (along with their characteristics, such as operating system identification), and any subnet formation – how many subnets exist and how many hosts each one contains. The proposed framework offers the needed flexibility to adjust the level of protection, from leaving information unchanged up to randomizing it.

The proposed framework provides an expressive application programming interface (API) that offers a wide set of anonymization functions to be applied on network traffic. The API is integrated inside the diMAPI implementation and is accessed through standard diMAPI function calls. Furthermore, in our effort to help administrators build well-defined anonymization policies, our framework comes with a policy editor tool that hides complexity of policy authoring and encapsulates the anonymization logic.

Chapter 2

State of the Art in Traffic Anonymization and the Related Policies

In this section we will present the anonymization policies most frequently used by the various organizations and institutions that make traffic traces publicly available. It is important to point out that the anonymization procedure used by these organizations is mainly applied offline to already captured traces, while our framework, as we will show in later chapters, can be applied both offline and online.

Next we will present the software tools which are available in order to enforce anonymization policies. Finally, we will present the research efforts in the area of traffic anonymization mechanisms.

2.1 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 inside the United States in the tcpdump format. The NLANR traces [12] 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 and signature-based intrusion detection.

The University of California, Los Angeles (UCLA) traces [18] 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, acknowledgement numbers, and window size are recorded. Packet payload is totally removed, thus leaving space only for header processing, similarly 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 [10] 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, the rest of header is unchanged but sensitive information in the payload like URL, filenames or passwords has been replaced by constant values. While providing the payload for specific protocols is better than always hiding the payload, replacing sensitive fields induces loss of precision. For example, the CodeRed worm 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 [19], `tpcdpriv` 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 [5], 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. However, prefix-preserving anonymization can be reversed if a certain number of mappings is revealed. An attacker can send a packet with certain characteristics to a sensor and observe its anonymized address into the trace, forming a mapping. A brief summary of the anonymization policies applied by the mentioned organizations and departments can be viewed at Table2.1.

2.2 Anonymization Tools

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

`Tcpdpriv` [8] is the most known tool of its category. It works only on traces written in tcpdump format and removes sensitive information by operating on

2.2. ANONYMIZATION TOOLS

<i>Organization</i>	<i>IP S/D addresses</i>	<i>Ports</i>	<i>Payload</i>	<i>Comments</i>
NLANR [12]	Mapped to integers	Intact	Removed	IPid and TTL replaced by constant values
UCLA [18]	Mapped to integers	Intact	Removed	Provided as text
LBL [10]	Mapped to integers	Intact	Removed	Provided as text, only IP, ports and packet size recorded
UCSD [19]	Mapped to integers	Intact	Set to zero	All other header fields unchanged
Dartmouth [5]	Prefix-preserving	Intact	Removed	All other header fields unchanged

Table 2.1: Summary of most commonly used anonymization policies.

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 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 leave IP addresses unchanged. Similarly, the user can specify the level of anonymization for information such as ports, IP class information or multicast addresses. Tatu Ylonen has written an article [23] 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 [12] are anonymized by using the `tcpdpriv` tool.

There are some tools based on `tcpdpriv`'s code. `Ip2anonip` [6] is a simple filter that turn IP addresses into host names or anonymous IPs. `Ipsumdump` [7] dumps packets into ascii format and uses `tcpdpriv` to anonymize IP addresses if specified by the user. `Tcpdpriv` is also used in `click router` [1] for anonymization of the source and destination IP addresses.

`Bro` [9] 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 [13]. Users can express sophisticated trace transformation 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 [10] using this software.

An implementation for prefix-preserving anonymization is available from Georgia Tech University. The software is called Crypto-PAn [4] (Cryptography-based Prefix-preserving Anonymization). Using these tools, the user is able to anonymize IP addresses of a packet trace in a prefix-preserving manner. This means that IP addresses that share a N bit common prefix, when anonymized will be mapped to addresses that will have in common N bits in their prefix [21, 22]. This implementation has several advantages compared to tcpdpriv, including memory consumption and consistent mapping across traces.

2.3 Related Research Work

Peuhkuri in “A Method to Compress and Anonymize Packet Traces”, 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.

The paper “On the Design and Performance of Prefix Preserving IP Traffic Trace Anonymization” [21], focuses on the problem of prefix-preserving IP anonymization. Existing implementations such as tcpdpriv [8] have many drawbacks like memory consumption, inconsistent mappings across different anonymization sessions and lack of parallel processing of traces. They propose an algorithm that makes use of cryptography that is stateless so it has small memory requirements. Apart from this, the mapping for a certain address is the same, assuming that the cryptographic key used is the same. This way parallel processing is feasible. Also mapping is independent of the order that IP addresses appear in the trace. The authors mention that their approach is as secure as the traditional approach where random bits are inserted to the mapping. On the other hand prefix preserving anonymization is less secure than the random one-to-one mapping of IPs since if the mapping of one address is compromised, then more information can be revealed since IPs share common prefixes.

Mogul [11] mentions 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.

Paxson and Pang in “A High-level Programming Environment for Packet Trace Anonymization and Transformation” [13] 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. However, this introduces several drawbacks. For example malicious traffic may contain overrun packets or wrong value in the checksum field. When packets are reconstructed, packet header values are calculated so these malicious packets will not appear in the output trace as in the input. This can be avoided by storing the original headers and use them in reconstruction. 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. The anonymization policy is expressed in a language that is specific to this software. The user can specify the field to be altered using regular expressions and the modification to be done. The software supports multiple ways of altering a field, such as one-to-one mapping, hashing, prefix-preserving or even adding random noise in order to defeat some form of fingerprinting attacks. 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 on the information that remains and can be used in the trace.

Prefix-preserving anonymization has also been applied to netflows [20]. The Crypto-PAn [4] software has been used and modified in order to generate the cryptographic key that is used from a passphrase. Anonymization is applied only to IP addresses of flows, while all other fields are left unchanged. Evaluation has shown that a dual Xeon 2.4 machine is able to anonymize about 75000 netflow records per second, exploiting the fact that anonymization can be carried out parallel.

Anonymization has also been implemented in hardware, using network processors [15]. 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 anonymization 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 anonymized using the anonymization tree in order to make anonymization more resistant against attacks thus 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.

CHAPTER 2. STATE OF THE ART IN TRAFFIC ANONYMIZATION AND
THE RELATED POLICIES

Chapter 3

Anonymization Architecture

The enforcement of anonymization policies in LOBSTER is going to facilitate the existing admission control system that is implemented within the Monitoring Application Programming Interface (MAPI) in the context of the IST SCAMPI¹ project. Therefore, before we introduce the LOBSTER anonymization framework, we will briefly outline the current architecture for Admission Control policy enforcement in MAPI.

3.1 Admission Control in MAPI

Admission control in MAPI has been designed as an independent module, known as `authd` [14]. `authd` is a daemon process that runs standalone and uses shared memory IPC to communicate with `mapid`. `authd` is responsible for authenticating a user, as well as checking his/her privileges against the specifications of a flow before activating it. Whenever a user wants to create a network flow, the authorization system enforces that:

- The user has the appropriate privileges to create this flow.
- The user has applied the anonymization functions defined by the anonymization policy.²

If the application does not have the appropriate privileges, or *credentials*, to create a flow with these characteristics, or if the appropriate anonymization functions have not been applied, then the flow will be rejected.

The information required for the authentication of a user is the following:

1. the user's public key,
2. an arbitrary positive integer , and

¹www.ist-scampi.org

²This second check was NOT present in the SCAMPI architecture.

3. the same number encrypted with the user's private key.

This information is enough to assure that the user actually holds the public/private key pair of the public key used for identification (and hence, that the user is who he/she claims to be).

The information required to authorize a new flow includes: the user's *public key*, the user's *credentials*, written in the Keynote format [2], and a detailed description of the flow in question, including the functions that are going to be applied to the flow (e.g., filters, counters, samplers), the number of instances of a specific function type, the position of the instances (in the linear list of applied functions), the arguments passed to each function, and the device with which the flow is associated.

For example, in the following credential assertion, an authorizer (e.g., an administrator of a monitoring sensor) grants a licensee (e.g., a potential user) the right to use the `STR_SEARCH` function if and only if the `SAMPLE` function has previously been applied to the flow with a parameter less than `0.2`, presumably to prevent a computationally intensive function like string searching from being applied to every packet.

```
Authorizer: "RSA:abc123" # Admin's key
Licensees:  "RSA:xyz987" # Users key
Conditions: (device_name ~= "eth[0-9]") &&
             ((STR_SEARCH != defined) ||
              (SAMPLE == defined &&
               SAMPLE.first < STR_SEARCH.first &&
               SAMPLE.param.1 < 0.2))
Signature:  "RSA-SHA1:213354f9" # signed by the Admin
```

The types of assertions that may be expressed in credentials are detailed in [16]. A complete description of the Admission Control system is provided in [17].

3.2 Anonymization Infrastructure Overview

In LOBSTER, the administrator of a monitoring sensor is responsible for specifying the anonymization policy that governs it. Each sensor may have several different policies for different users or user groups. The administrator issues credentials for potential users of that sensor, which enforce the use of certain anonymization functions. Users that have not applied the anonymization functions specified in their credentials are not authorized to use the sensor.

Figure 3.1 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.

3.2. ANONYMIZATION INFRASTRUCTURE OVERVIEW

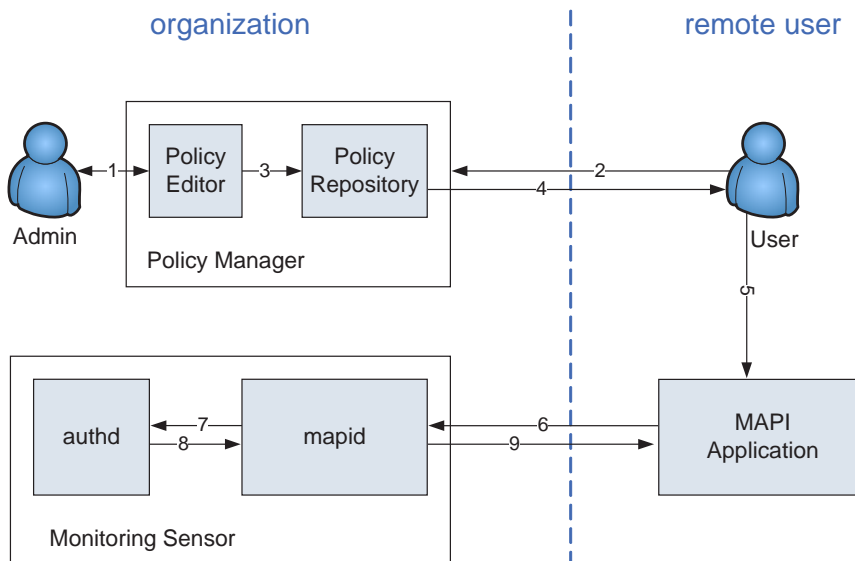


Figure 3.1: **Anonymization framework in LOBSTER.** (1) The Administrator specifies an anonymization policy. (2) The user delivers his/her public key. (3) Signed credentials are stored in the Policy Repository. (4) The user obtains his/her credentials. (5) User's network flows are configured to conform 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.

Initially, the administrator must specify the anonymization policy for the monitoring sensor. For convenience, can use a *Policy Editor* (**step 1**) to define the anonymization policy, instead of writing the conditions directly in the KeyNote language. The Policy Editor, which is presented in more details in Section 4.1, provides a user-friendly Graphical User Interface (GUI) to the sensor administrator for specifying anonymization policies, and automatically produces the relevant conditions in the KeyNote format.

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 anonymiza-

tion 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 that corresponds to the public key is achieved by supplying a nonce value (an integer number), together with the encrypted version of this integer using the users private key. If the encrypted nonce decrypted with the supplied public key equals the original value of the nonce, the users request is authenticated. By using the flow descriptor as the nonce value, the authentication is also tied to this particular request. When the configuration of the flow is completed, the user instantiates the flow by calling `mapi_connect()` (**step 6**).

At this point, `mapid` 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` (**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, `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.

3.3 Anonymization Mechanisms within a Sensor

In order to enforce the administrator's policies we need a) an interface to the user for requesting³ specific anonymization for his flows, and b) a mechanism inside MAPI that will realize the anonymization requests. The former is implemented via the use within `mapi_apply_function()` of a new function called `ANONYMIZE`, and the latter with number of anonymization functions inside the sensor that will realize the anonymization requests.

The user's anonymization requests will be implemented as a series of standard functions which will be invoked through the `mapi_apply_function` call as:

```
mapi_apply_function(fd, "ANONYMIZE", protocol,  
                   field_description, anonymization_function,  
                   function_parameters);
```

The `protocol`, `field_description`, `anonymization_function`, `checksum_behavior` parameters of the `ANONYMIZE` function take values from a list of predefined values. `function_params` include any additional parameters of the specified `anonymization_function`. The `anonymization_functions`

³The user must request anonymization which is compliant with the administrator's policy.

3.3. ANONYMIZATION MECHANISMS WITHIN A SENSOR

represent the mechanism inside MAPI that is facilitated in order to enforce an anonymization policy.

In the following we give short descriptions of each parameter of ANONYMIZE, as well as an initial listing of the predefined values for each of them.

`protocol`: IP, TCP, UDP, ICMP and initially well-known application protocols like HTTP and FTP are supported.

`field_description`: specific fields of each protocol. Common fields will be interpreted according to the specified `protocol`. An example of field is `FLAGS`, which corresponds to the TCP flags field. As another example, `PAYLOAD` is a field that is interpreted according to the protocol. If protocol is TCP, only the TCP payload will be transformed. If protocol is HTTP, only the contents of the HTTP transfer will be altered while HTTP headers will remain intact. The complete list of fields can be found in Appendix A.

`anonymization_function`: the various functions according to which fields will be transformed. Examples of anonymization functions are `MAP`, which maps a field to an integer, `STRIP`, which totally removes a field and `HASH` that replaces a field with a hash value. The list of predefined functions, along with the restrictions on which fields they can be applied, is located in Appendix B.

If anonymization is performed on application layer protocols (HTTP, FTP, etc), then it must be performed on top of a reassembled stream in order to be semantically correct. When the anonymization functions used are only for transport layer and below protocols (TCP, UDP, IP, etc), there is no need to reassemble the stream, and therefore packet stream “cooking” should not be used.

For the implementation of the anonymization API, we have modified or implemented a number of functions according to our needs.

- **The main anonymize function**: It is actually a wrapper that will invoke anonymize functions after examining that parameters are correct (for example, the IP protocol has no URL field).
- **Packet decoder**: decodes the raw bytes of a packet to higher-level structures.
- **Protocol decoders**: In order to support anonymization in application-level, decoders for the HTTP and FTP protocols are needed.
- **IP defragmentation/ TCP reassembly**: IP defragmentation merges IP fragments to a normal IP packet. TCP reassembly assembles TCP fragments to a larger stream. `libnids` was used as the base library (<http://sourceforge.net/projects/libnids/>)

- **Anonymization functions:** mapping, random and pattern fill functions were written from scratch. Open-source implementation of hashing functions are available for all MD5, SHA and CRC32 algorithms. Regular expression function are implemented using the PCRE library (www.pcre.org)

The expressiveness of our API allows the developers to make an application that anonymizes packets with few function calls. As an example, an application that anonymizes packets following the NLANR policy, described in Chapter 2, takes no more than 20 lines. The source code of such application can be viewed in Appendix C.

3.3.1 Anonymization as a Transparent Process

Information on high-level protocols, like HTTP or FTP, spans across multiple packets, thus anonymization on this level should be performed on top of a stream instead of a per-packet basis. MAPI has the ability to reassemble packets in order to form a cooked packet, through the “COOKING” function. It is thus highly and strongly recommend that a COOKING function must precedes 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 in multiple TCP packets. Applying anonymization without cooking, only the first packet of the transfer can be classified as FTP and consequently the rest of packets composing the file transfer cannot be classified and anonymized. When cooking is applied, the whole transfer is contained in a single packet so the contents of the whole file can be set to zero.

However, anonymization process must be transparent to functions that succeed the anonymization functions. It would be both restrictive and undesirable behavior to force these functions to be performed on cooked packets. Our approach is to split the cooked—and anonymized—packet back to the original packets. Splitting is implemented as a MAPI function, called “UNCOOK”. The cooking function was modified to store the list of headers of the original packets that form the cooked packet. “UNCOOK” takes this list of headers and adds them the appropriate portion of the payload of the cooked and anonymized packet. In that way, “UNCOOK” constructs as many packets as they were originally, each one having a header just like it was before the “COOKING” function but with an anonymized payload. It must be noted that some packets that originally had payload, may not have payload after the “UNCOOK” function, e.g., when replacing the whole payload with a hash value. Functions that are below “UNCOOK” are called for each of the packets that “UNCOOK” constructs.

3.3.2 Function Reordering

Function reordering is an optional component of the anonymization framework that can be selectively enabled or disabled from the configuration of the MAPI daemon.

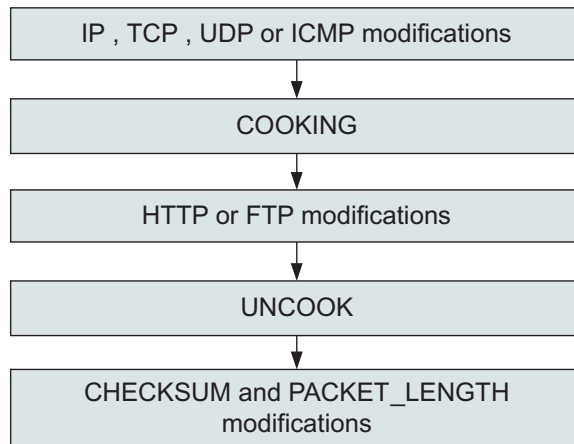


Figure 3.2: **Reordering of Anonymization Functions**

Its main goal is double. Firstly, to automatically detect common pitfalls in the anonymization functions applied, both in what anonymization functions are applied and in which order. Secondly, to ensure that the semantics of anonymization process are correct. This component is specific for anonymization functions and does not affect any other functions that user applies either before or after anonymization part. The function reordering is performed upon flow connection. It examines all functions in the flow and tries to find the first and the last “ANONYMIZE” function. A “COOKING” that is followed by an “ANONYMIZE” function is also part of anonymization functions as well as an “UNCOOK” function that is preceded by an “ANONYMIZE”. Function reordering performs three major tasks:

- All anonymization functions except “CHECKSUM_ADJUST” that are applied on IP, TCP, UDP or ICMP level are moved first. By moving these function first, there are two benefits. First, if these functions were located between a “COOKING” and an “UNCOOK” function, then the headers of the packets that “UNCOOK” constructs wouldn’t be anonymized as no anonymization precedes “COOKING” or follows “UNCOOK”. Second, we can take advantage of the underlying hardware capabilities and push some or all of these functions to the hardware. If, for example, a “HASH” to the source IP address was accidentally applied after a “COOKING” function, then it could not be transferred to hardware level.
- “CHECKSUM_ADJUST” and functions that alter packet length are moved last. “CHECKSUM_ADJUST” is moved last so as to reflect all changes, after all other anonymization functions have been performed. Packet length is moved last because all functions that modify the packet size, internally set the packet length to the correct size. As a result, explicit modifications to the packet length must be performed at the end.

- If functions that modify the HTTP or FTP protocol are applied, they are grouped together. If a “COOKING” function exists, then it is moved before this group, elsewhere it is manually applied. If an “UNCOOK” function exists, it is moved after this group, elsewhere it is manually applied. Having “COOKING” 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 “COOKING” or “UNCOOK” functions are accidentally added then duplicate functions are removed.

The function reordering process is illustrated in Figure 3.2.

Chapter 4

Defining Anonymization Policies

In this section we will present a number of issues related to defining Anonymization Policies with the LOBSTER framework. Our main goal is to aid the network administrators that want to define anonymization policies for the monitoring sensors within their organization. Toward this target we present an Anonymization Policy Editor Tool, which automates the procedure of creating anonymization policies in the Keynote notation. Finally we present a set of rules for the creation of meaningful anonymization policies.

4.1 Anonymization Policy Editor Tool

Policy creation is a boring process which requires writing tens of lines describing accurately the desired behavior. Furthermore, creating a policy for imposing anonymization inside MAPI can be a lengthy process, as the number of packet fields is large. Additionally, all anonymization functions cannot be applied to all fields and some of them require extra parameters.

In our effort to reduce the administrative overhead and assure that created policies are semantically correct, a tool that graphically generates policies is needed. Such a tool will be used by the network administrators to set up the anonymization policies¹ for the sensors within their domain.

In this section we will make a first attempt to design such an anonymization policy maker tool, which is consistent with the overall framework we presented in the previous sections.

As seen in Figure 4.1 the packet header is displayed graphically. The user can right click on a field and select which anonymization function to apply. For user's convenience, more friendly names for functions are displayed instead of the definitions as described in Section 3.3. The tool is able to display IP, TCP, UDP, ICMP, HTTP and FTP headers—all the protocols described in Appendix A—with all their fields. The restrictions enforced by the anonymization framework, e.g.,

¹We can extend this tool to be able to configure other policies as well, such as admission control, but this discussion is outside the scope of this note.

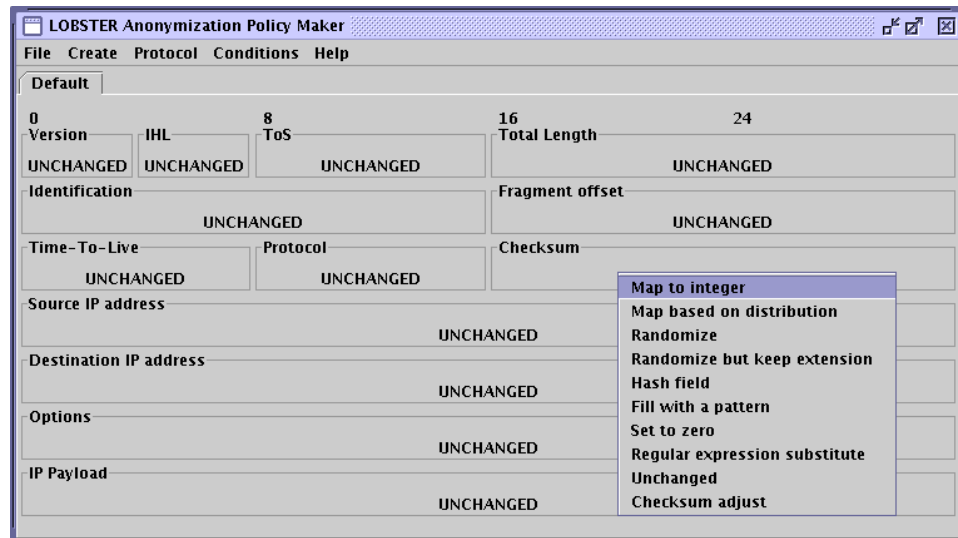


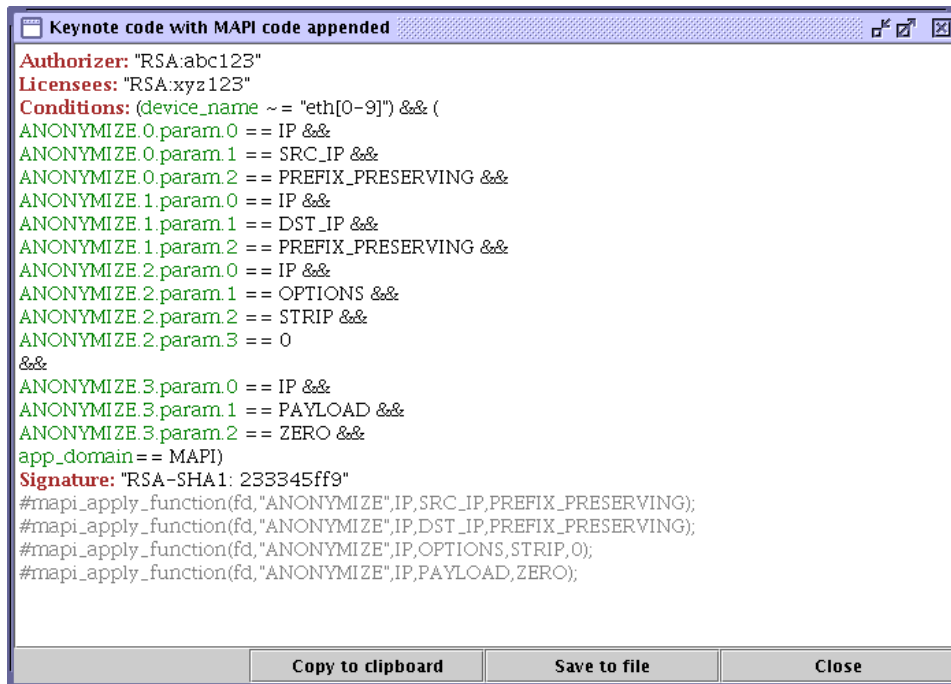
Figure 4.1: Anonymization Policy Editor tool

source and destination IP addresses cannot be removed totally, are integrated inside the tool. In that way, the user is prevented from selecting a function that cannot be applied to a specific field.

After selecting which function should be applied to which field, the user can generate code in three forms:

- **Keynote code:** Anonymization functions are translated into keynote code. An example can be seen in Figure 4.2. The user has applied “prefix preserving” hashing to both source and destination IP addresses, has removed the IP options and set the IP payload to be filled with zeros.
- **MAPI code:** A series of `mapi_apply_function` calls are generated which provide a quick way to apply the functions selected to a MAPI program. The same example described in keynote code can be seen as MAPI calls in Figure 4.3
- **Keynote+MAPI code:** It is the same with the keynote code option but the MAPI code is appended as comments. It provides a comprehensive way to see how keynote code can be applied in terms of MAPI code. The appended code can be also seen in Figure 4.2.

The user can select to apply “COOKING” before the anonymization functions or “UNCOOK” after anonymization functions. If selected, “COOKING” and/or “UNCOOK” functions are integrated into both Keynote and MAPI code. Additionally the policy maker can allow or enforce specific functions before anonymization process. Enforcing a function before anonymization means that the function must



```

Authorizer: "RSA:abc123"
Licensees: "RSA:xyz123"
Conditions: (device_name ~ = "eth[0-9]") && (
ANONYMIZE 0.param.0 == IP &&
ANONYMIZE 0.param.1 == SRC_IP &&
ANONYMIZE 0.param.2 == PREFIX_PRESERVING &&
ANONYMIZE 1.param.0 == IP &&
ANONYMIZE 1.param.1 == DST_IP &&
ANONYMIZE 1.param.2 == PREFIX_PRESERVING &&
ANONYMIZE 2.param.0 == IP &&
ANONYMIZE 2.param.1 == OPTIONS &&
ANONYMIZE 2.param.2 == STRIP &&
ANONYMIZE 2.param.3 == 0
&&
ANONYMIZE 3.param.0 == IP &&
ANONYMIZE 3.param.1 == PAYLOAD &&
ANONYMIZE 3.param.2 == ZERO &&
app_domain == MAPI)
Signature: "RSA-SHA1: 233345ff9"
#mapi_apply_function(fd,"ANONYMIZE",IP,SRC_IP,PREFIX_PRESERVING);
#mapi_apply_function(fd,"ANONYMIZE",IP,DST_IP,PREFIX_PRESERVING);
#mapi_apply_function(fd,"ANONYMIZE",IP,OPTIONS,STRIP,0);
#mapi_apply_function(fd,"ANONYMIZE",IP,PAYLOAD,ZERO);

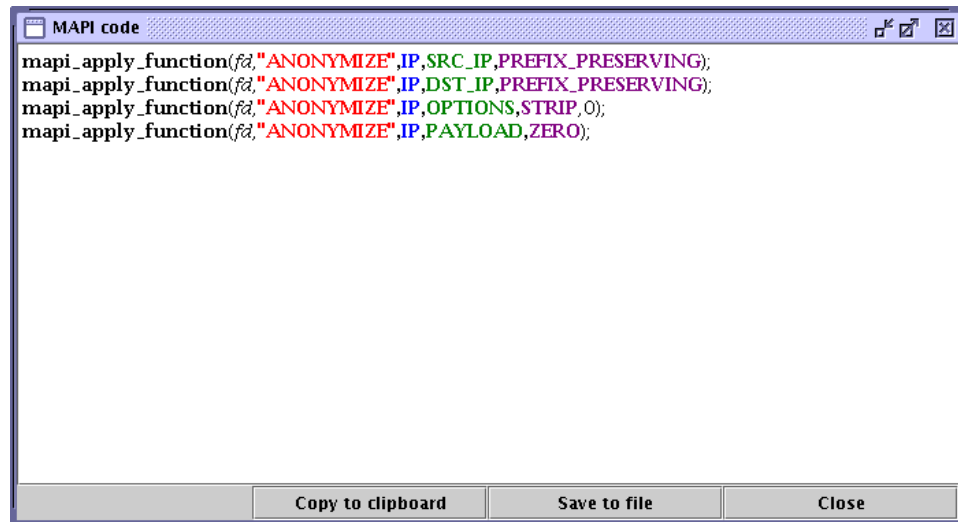
```

Figure 4.2: Keynote code generated by the tool

be used at least once, and the position of its instance must be lower than the position of the first anonymization function. When enforcing more than one function, the position of the first enforced function must be lower than the one of second enforced function, the position of the second enforced function must be lower than the one of third enforced function and so on, and the position of the last enforced function must lower than the position of the first anonymization function. The user is able to set the correct parameters for the enforced functions so as the generated policy is as accurate as possible (there are no default values for “BPF FILTER” for example). Allowing a function to be applied before the anonymization functions does not impose any restrictions. The allowed functions can be either applied or not but when applied the position of their first instance must be lower than the one of first anonymization function. Furthermore, the policy maker provides the ability to select which functions cannot be used at all, whether before or after the anonymization process. Functions that cannot be applied are removed from the set of allowed and enforced functions before anonymization.

4.2 Guidelines for Policy Preparation

Applying anonymization functions by hand would be a difficult process that human errors may lead to conflicts when trying to interpret or enforce, while the final



```
mapi_apply_function(fd,"ANONYMIZE",IP,SRC_IP,PREFIX_PRESERVING);
mapi_apply_function(fd,"ANONYMIZE",IP,DST_IP,PREFIX_PRESERVING);
mapi_apply_function(fd,"ANONYMIZE",IP,OPTIONS,STRIP,0);
mapi_apply_function(fd,"ANONYMIZE",IP,PAYLOAD,ZERO);
```

Figure 4.3: MAPI code generated by the tool

outcome will be different than the desired one. In this section we will try to clarify some the contradictory points when trying to set anonymization policies.

Altering the payload of a protocol Applying functions like ZERO or HASH to the whole payload of a protocol automatically flags that anonymization in the deeper-nested protocols cannot take place. For example, if it is defined that the IP payload will be replaced by a hash value or it will be set to zero, then all functions concerning TCP or UDP will have no effect.

Attention when altering upper-protocol fields Changing the values in fields inside upper-level protocols, like HTTP or FTP, may generate payloads that do not conform with protocol standards or cause problems in their parsers. Take as an example replacing an HTTP field with a hash value. The hash value may contain a “\r\n” character pair that will be treated by HTTP parsers as the end of the field. The parser will treat the rest of the hash value as the next HTTP header, which will have no effect and causes the rest of the headers to be treated as normal data. It is advisable that fields in upper-layer protocols are replaced with constant value, whose behavior when injected into payload is predictable.

Attention to packet length Hash and replace functions generally alter the packet length . Functions that explicitly change the packet length, like for example setting the packet length to a fixed value, must be applied after all other functions or else their changes will be undone by hash and replace functions.

Checksum Altering packet fields, invalidates the checksum of the corresponding protocol (IP, UDP, ICMP). The role of “CHECKSUM_ADJUST” function is to re-calculate the packet checksum and thus it must be applied after all other anonymization functions have been applied. Note also that UDP or ICMP checksums have to be re-calculated first and then the IP checksum.

4.2.1 Examples

In this section we present a simple example for the anonymization policy configuration of a monitoring sensor within an organization, in our case FORTH.

We assume that FORTH has a LOBSTER sensor that monitors a link to the Internet. The policy we want to apply is the following: FORTH's internal users may have access to full packet headers, but cannot access any payload data. The same holds for the applications of other non-FORTH LOBSTER users, for which in addition the IP addresses should to be anonymized.

The credentials for an intra-FORTH user require the removal of the packet payload as follows (note: some assertions are omitted for readability):

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

This credential assertion is issued by the FORTH's administrator as identified by his public key in the Authorizer field), a user is granted the right to create a flow, if and only if the first function applied to that flow is the function ANONYMIZE, with arguments TCP, PAYLOAD, STRIP. Thus, the first function call for each newly created flow should be the following:

```
mapi_apply_function(fd, ANONYMIZE, TCP, PAYLOAD, STRIP);
```

Note that in the Licensees field, a single user is identified by his/her public key. However, it is also possible to include several keys in the same field in order to authorize multiple users.

The following credentials require a user outside FORTH to remove the packet payload and anonymize the IP addresses using address mapping.

```
Authorizer: "RSA:abc123" # The FORTH's Admin key
Licensees: "RSA:xyz999" # other user's key
Conditions: ANONYMIZE == "defined" &&
            ANONYMIZE.0.pos == 0 &&
            ANONYMIZE.0.param.0 == IP &&
            ANONYMIZE.0.param.1 == SRC_IP &&
            ANONYMIZE.0.param.2 == MAP &&
            ANONYMIZE.1.pos == 1 &&
            ANONYMIZE.1.param.0 == IP &&
            ANONYMIZE.1.param.1 == DST_IP &&
```

CHAPTER 4. DEFINING ANONYMIZATION POLICIES

```
        ANONYMIZE.1.param.2 == MAP &&
        ANONYMIZE.2.pos == 0 &&
        ANONYMIZE.2.param.0 == TCP &&
        ANONYMIZE.2.param.1 == PAYLOAD &&
        ANONYMIZE.2.param.2 == STRIP;
Signature: "RSA-SHA1:213344f9"
```

A network flow which complies with the above credentials should be configured using the following the following anonymization functions:

```
mapi_apply_function(fd, ANONYMIZE, IP, SRC_IP, MAP);
mapi_apply_function(fd, ANONYMIZE, IP, DST_IP, MAP);
mapi_apply_function(fd, ANONYMIZE, TCP, PAYLOAD, STRIP);
```

This is a more complex anonymization that is similar to the policy of NLANR as described before. IP addresses are mapped to integers, payload is stripped, TTL and ID value are replaced with constants and finally, checksum is fixed.

```
mapi_apply_function(fd, "ANONYMIZE", IP, SRC_IP, MAP)
mapi_apply_function(fd, "ANONYMIZE", IP, DST_IP, MAP)
mapi_apply_function(fd, "ANONYMIZE", TCP, PAYLOAD,
    STRIP, 0)
mapi_apply_function(fd, "ANONYMIZE", UDP, PAYLOAD,
    STRIP, 0)
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", IP, CHECKSUM,
    CHECKSUM_ADJUST)
mapi_apply_function(fd, "ANONYMIZE", TCP, CHECKSUM,
    CHECKSUM_ADJUST)
mapi_apply_function(fd, "ANONYMIZE", UDP, CHECKSUM,
    CHECKSUM_ADJUST)
```

The next policy, also performs anonymization only on packet headers and strips the payload. Here we use the prefix-preserving function for IP addresses, and apply a gaussian distribution to the values of packet length and TTL and a random value to identification field.

```
mapi_apply_function(fd, "ANONYMIZE", IP, SRC_IP,
    PREFIX_PRESERVING)
mapi_apply_function(fd, "ANONYMIZE", IP, DST_IP,
    PREFIX_PRESERVING)
mapi_apply_function(fd, "ANONYMIZE", TCP, PAYLOAD, STRIP, 0)
```

4.3. SECURITY IMPLICATIONS OF THE ANONYMIZATION FUNCTIONS

```
mapi_apply_function(fd,"ANONYMIZE", UDP, PAYLOAD, STRIP, 0)
mapi_apply_function(fd,"ANONYMIZE", IP, PACKET_LENGTH,
    MAP_DISTRIBUTION, GAUSSIAN, 700, 500)
mapi_apply_function(fd,"ANONYMIZE", IP, TTL,
    MAP_DISTRUBUTION, GAUSSIAN, 50, 30)
mapi_apply_function(fd,"ANONYMIZE", IP, ID, RANDOM)
mapi_apply_function(fd,"ANONYMIZE", IP, CHECKSUM,
    CHECKSUM_ADJUST)
mapi_apply_function(fd,"ANONYMIZE", TCP, CHECKSUM,
    CHECKSUM_ADJUST)
mapi_apply_function(fd,"ANONYMIZE", UDP, CHECKSUM,
    CHECKSUM_ADJUST)
```

Finally, the policy shown below performs anonymization on application level (HTTP). The URI is replaced by random characters, the HOST field is replaced by a constant value and COOKIE and HTTP PAYLOAD (the content of the URI for a http response) are removed.

```
mapi_apply_function(fd,"ANONYMIZE", IP, SRC_IP, MAP)
mapi_apply_function(fd,"ANONYMIZE", IP, DST_IP, MAP)
mapi_apply_function(fd,"ANONYMIZE", HTTP, URI,
    FILENAME_RANDOM)
mapi_apply_function(fd,"ANONYMIZE", HTTP, HOST,
    REPLACE, "WWW_SERVER")
mapi_apply_function(fd,"ANONYMIZE", HTTP, COOKIE,
    STRIP, 0)
mapi_apply_function(fd,"ANONYMIZE", HTTP, SET-COOKIE,
    STRIP, 0)
mapi_apply_function(fd,"ANONYMIZE", HTTP, PAYLOAD,
    STRIP, 0)
mapi_apply_function(fd,"ANONYMIZE", IP, CHECKSUM,
    CHECKSUM_ADJUST)
mapi_apply_function(fd,"ANONYMIZE", TCP,CHECKSUM,
    CHECKSUM_ADJUST)
```

4.3 Security Implications of the Anonymization Functions

There are several well known attacks to anonymization schemes that can be used in order to reveal information that theoretically has been removed during anonymization. These issues are further discussed in [3] focusing in the security implications of the anonymization of IP source/destination addresses of IP packet headers. Authors suggest that hashing should not be applied to these fields. That is because of the short length of this field², the adversary can compute hashes for the entire IPv4

²32 bits

space in a matter of hours. If it is absolutely necessary to use hashing, one should hash pairs of IP addresses or use keyed hashing.

Furthermore, all current anonymization schemes, are vulnerable to injection attacks by an active adversary. This is the case where specially crafted packets are sent by the adversary, processed and anonymized by the monitor node and the adversary gets access to the anonymized packets. Making these packets recognizable in the anonymized traffic (for example using special values in header fields), the adversary can find the mapping between original and anonymized IP addresses. This gets even worse in the case of prefix-preserving anonymization, since anonymized addresses share common prefixes. So by finding a matching for an IP address, information about all other IP addresses that share a prefix with this one, are revealed. In the cases above further protection is necessary, so the authors of [3] proposed two algorithms that provide additional resistance against these attacks.

Chapter 5

The SiSaL Scripting Sanitization Language

The specification of an anonymization policy as a list of required and admissible MAPI (filter, anonymize) functions is quite expressive. In addition, the automated tool, such as the Anonymization Policy Editor Tool we present in section 4.1, will help administrators to express anonymization policies. In some cases though it is desirable to provide alternative ways to administrators to express the same anonymization policy. For example network and systems administrators are used to scripting languages¹ for the configuration of various network elements. Therefore, in this section we will make the first steps towards the design of a *rule-based language* called *SiSaL* (Scripting Sanitization Language), which will be an alternative way of expressing the same anonymization functionality. SiSaL allows complicated anonymization policies to be specified in a concise manner. This helps the anonymization policy specification for administrators familiar with scripting languages, avoids errors, inspires confidence, and may well induce more lenient policies than the administrator would otherwise dare to employ.

For example, an administrator may be prepared to publish some email packets with only minimal anonymization, but only if they match a list of known virus signatures. Although this policy could be expressed in the framework described in the previous chapters, the specification would be difficult to read. Instead, we will support *rewrite rules*, that describe the full set of packet operations in a single, compact, description.

For example, the policy described above could be expressed with the following set of rewrite rules:

```
pattern smtp = ip with [protocol=6] tcp with [port=25];
pattern local = 130 161 158 byte;
pattern virus = * "virus" *;
```

¹See for example the CISCO IOS scripting configuration language: <http://www.cisco.com/univercd/cc/td/doc/product/software/>.

```
filterViruses:
  header:smtp with [src=local,dst=local] packet:virus
    => header with [src=0,dst=0] packet;
  header:smtp with [dst=local] packet:virus
    => header with [dst=0] packet;
  header:smtp with [src=local] packet:virus
    => header with [src=0] packet;
  header:smtp packet:virus
    => header packet;
  * => reject;
```

This specification first defines a `smtp` pattern that specifies the IP and TCP headers for a SMTP packet. This pattern uses pre-defined patterns with the layout of IP and TCP; with the `with` clauses they are made more specific so that the IP header only matches the TCP protocol, and the TCP header only matches communication to port 25 (the SMTP port). The `local` pattern specifies the format of local addresses: four bytes, of which three have a specified value, and last one may have an arbitrary value. The `virus` pattern specifies the form of virus patterns. For the sake of simplicity we specify it here as a span of bytes of arbitrary length that contains the string `virus` somewhere in the span. Finally, the `filterViruses` rule specifies that SMTP packets that have the virus pattern as payload are rewritten to nearly identical copies, but that any local source or destination addresses are zeroed. Only the first rewrite specification that matches is applied, so the last line of the specification states that all packets that do not match an earlier rewrite rule are to be rejected. (This is in fact the default.)

The advantage of this description is that the entire filtering and anonymization policy is contained in one compact and easily understandable definition. This clarity avoids errors, inspires confidence, and may well induce an administrator to allow more lenient policies than he/she would otherwise dare.

The rewrite rules rely on a library of pre-defined patterns that describe the layout of for example IP and TCP headers. Also, in the example a successful match leads to the construction of a new packet from fragments of the matched packet. However, we envision more complicated actions, such as nested applications of rules on packet fragments, updates of statistics counters, and the application of MAPI functions.

The rewrite rules form a language that we call SiSaL (Scripting Sanitization Language). SiSaL can be integrated in MAPI by translating SiSaL scripts to a series of invocations of standard MAPI function, or by generating C code for new MAPI functions from SiSaL scripts, or a combination of the two.

An administrator can enforce anonymization policies by explicitly approving particular SiSaL scripts. These scripts can be stored in the policy repository described in Section 3.2.

As a further refinement, part of the security information may also be moved into the SiSaL script. For example, the following script specifies a rule that transforms packets from security domain `raw` (presumably the raw sensor output) to security domain `local` (presumably the group of local personnel).

```
discloseLocal: raw -> local
  hdr:ip data:* => hdr with [src=0,dest=0] data
```

(This rule copies all packets verbatim, except that it zeroes the source and destination address.)

Of course additional security infrastructure is still necessary: administrators must approve such scripts and add their cryptographic signature, users must still provide credentials, the approved security domains for all users must be administered, etc.

On a more speculative note: specification of the packet processing rules and security policies in SiSaL makes them more accessible to verification tools. Although beyond the scope of the LOBSTER, previous work on formal verification may well be applicable to SiSaL scripts, and may allow security and anonymization properties to be formally verified.

CHAPTER 5. THE SISAL SCRIPTING SANITIZATION LANGUAGE

Chapter 6

Summary

Privacy protection is a sensitive issue. Users do not want their private information shared and exposed. On the other hand security applications and traffic monitoring research efforts require access to a lot of information about the traffic that passes through various vantage points. In contrast with traditional approaches that hide almost all information from a captured traffic trace to ensure privacy protection, anonymization inside LOBSTER allows customizable definition of anonymization policies on a *per-network flow basis*.

The administrator of each sensor defines the anonymization policy — as a set of rules, according to which the network traffic is anonymized for each network flow of the trace. The policy enforcement mechanisms we have identified allow the administrator to enforce practically any change required in the packets stream, either in the TCP/IP layer, like for example randomizing the IP addresses, or even at higher application layers, like for example removing cookies from an HTTP session.

Users of this anonymization framework are the consumers of the anonymized traffic trace. They have to be compliant with the anonymized policies as they are defined by the LOBSTER sensor administrator. In our framework we ensure this compliance through the appropriate authentication daemons, which will refuse the flow creation in case of a network traffic flow request from a user with a non-compliant anonymization policy.

CHAPTER 6. SUMMARY

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, PROXY_AUTHENTICATE, PROXY_AUTHORIZATION, RANGE, REFERRER, RETRY_AFTER, SET_COOKIE, SERVER, TE, TRAILER, TRANSFER_ENCODING, UPGRADE, USER_AGENT, VARY, VIA, WARNING, WWW_AUTHENTICATE, X_POWERED_BY, RESPONSE_CODE, 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,

APPENDIX A. PREDEFINED PROTOCOL FIELD NAMES

STOR, APPE, STOU, ALLO, MKD, RMD, DELE, RNFR, RNTO, SITE,
FTP_RESPONSE_CODE, FTP_RESPONSE_ARG

Appendix B

Complete List of the Protocol Field 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.

APPENDIX B. COMPLETE LIST OF THE PROTOCOL FIELD ANONYMIZATION FUNCTIONS

- **FILENAME_RANDOM**: a sub-case of RANDOM. If the field is in a file-name 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.
- **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 B. COMPLETE LIST OF THE PROTOCOL FIELD
ANONYMIZATION FUNCTIONS

Appendix C

Implementation of the NLANR Anonymization Policy in LOBSTER

In this appendix we provide an example application of the LOBSTER Anonymization Framework that implements the NLANR anonymization policy, i.e., it anonymizes packets according to the NLANR rules. IP addresses are mapped to integers, TTL and identification number are set to constant values, while TCP or UDP payload is removed. In case we do not have TCP or UDP, the whole IP payload is removed.

The given application assumes that the monitored interface is ``eth0``.

```
#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);
    }

    fd2=mapi_create_flow("eth0");
    if(fd2==-1) {
        printf("Flow cannot be created. Exiting..\n");
        exit(-1);
    }
}
```

APPENDIX C. IMPLEMENTATION OF THE NLANR ANONYMIZATION POLICY IN LOBSTER

```
}

//if we do not have TCP or UDP then we have to
//remove the whole IP payload

mapi_apply_function(fd,"BPF_FILTER","not (tcp or udp)");
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", IP, PAYLOAD,
    STRIP, 0);
//checksum fix in IP fixes checksums in TCP and UDP as well
mapi_apply_function(fd,"ANONYMIZE", IP, CHECKSUM,
    CHECKSUM_ADJUST);

//in case of TCP or UDP we remove the TCP or UDP
//payload accordingly

mapi_apply_function(fd2,"BPF_FILTER","tcp or udp");
mapi_apply_function(fd2,"ANONYMIZE", IP, SRC_IP, MAP);
mapi_apply_function(fd2,"ANONYMIZE", IP, DST_IP, MAP);
mapi_apply_function(fd2,"ANONYMIZE", IP, TTL,
    PATTERN_FILL, INTEGER, 64);
mapi_apply_function(fd2,"ANONYMIZE", IP, ID,
    PATTERN_FILL, INTEGER, 242);
mapi_apply_function(fd2,"ANONYMIZE", TCP, PAYLOAD,
    STRIP, 0);
mapi_apply_function(fd2,"ANONYMIZE", UDP, PAYLOAD,
    STRIP, 0);
mapi_apply_function(fd2,"ANONYMIZE", IP, CHECKSUM,
    CHECKSUM_ADJUST);
//checksum fix in IP fixes checksums in TCP and UDP as well
mapi_apply_function(fd2,"ANONYMIZE", IP, CHECKSUM,
    CHECKSUM_ADJUST);

}
```

Appendix D

Implementation of the Dartmouth Anonymization Policy in LOBSTER

Another example of how we can implement an anonymization policy inside LOBSTER within a few lines of code. In this policy, IP addresses are anonymized in a prefix-preserving way while the TCP or UDP payload is removed.

```
#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);
    }

    mapi_apply_function(fd,"ANONYMIZE", IP, SRC_IP, PREFIX_PRESERVING);
    mapi_apply_function(fd,"ANONYMIZE", IP, DST_IP, PREFIX_PRESERVING);
    mapi_apply_function(fd,"ANONYMIZE", TCP, PAYLOAD,
        STRIP, 0);
    mapi_apply_function(fd,"ANONYMIZE", UDP, PAYLOAD,
        STRIP, 0);

    //checksum fix in IP fixes checksums in TCP and UDP as well
```

APPENDIX D. IMPLEMENTATION OF THE DARTMOUTH
ANONYMIZATION POLICY IN LOBSTER

```
mapi_apply_function(fd, "ANONYMIZE", IP, CHECKSUM,  
                    CHECKSUM_ADJUST);  
}
```

Appendix E

Implementation of the University of California, San Diego Anonymization Policy in LOBSTER

An example similar to NLANR anonymization policy but in this case payload is set to zero. The code below is mainly the core functionality of the default `tcpdpriv` settings.

```
#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);
    }

    fd2=mapi_create_flow("eth0");
    if(fd2==-1) {
        printf("Flow cannot be created. Exiting..\n");
        exit(-1);
    }
}
```

APPENDIX E. IMPLEMENTATION OF THE UNIVERSITY OF
CALIFORNIA, SAN DIEGO ANONYMIZATION POLICY IN LOBSTER

```
//if we do not have TCP or UDP then we have to
//remove the whole IP payload

mapi_apply_function(fd,"BPF_FILTER","not (tcp or udp)");
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", IP, PAYLOAD, ZERO);
//checksum fix in IP fixes checksums in TCP and UDP as well
mapi_apply_function(fd,"ANONYMIZE", IP, CHECKSUM,
    CHECKSUM_ADJUST);

//in case of TCP or UDP we remove the TCP or UDP
//payload accordingly

mapi_apply_function(fd2,"BPF_FILTER","tcp or udp");
mapi_apply_function(fd2,"ANONYMIZE", IP, SRC_IP, MAP);
mapi_apply_function(fd2,"ANONYMIZE", IP, DST_IP, MAP);
mapi_apply_function(fd2,"ANONYMIZE", IP, TTL,
    PATTERN_FILL, INTEGER, 64);
mapi_apply_function(fd2,"ANONYMIZE", IP, ID,
    PATTERN_FILL, INTEGER, 242);
mapi_apply_function(fd2,"ANONYMIZE", TCP, PAYLOAD, ZERO);
mapi_apply_function(fd2,"ANONYMIZE", UDP, PAYLOAD, ZERO);
mapi_apply_function(fd2,"ANONYMIZE", IP, CHECKSUM,
    CHECKSUM_ADJUST);
//checksum fix in IP fixes checksums in TCP and UDP as well
mapi_apply_function(fd2,"ANONYMIZE", IP, CHECKSUM,
    CHECKSUM_ADJUST);

}
```

Appendix F

Implementation of the LBL HTTP and FTP Anonymization Policy in LOBSTER

This policy is focused mainly on the HTTP and FTP protocols. IP addresses are mapped and all other header fields remain unchanged. However, requested URLs at the HTTP level and passwords and filenames in the FTP protocol are substituted with constant values.

```
#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);
}

//if we do not have TCP or UDP then we have to
//remove the whole IP payload

mapi_apply_function(fd,"ANONYMIZE", IP, SRC_IP, MAP);
mapi_apply_function(fd,"ANONYMIZE", IP, DST_IP, MAP);

//work on the HTTP level
```

APPENDIX F. IMPLEMENTATION OF THE LBL HTTP AND FTP
ANONYMIZATION POLICY IN LOBSTER

```
mapi_apply_function(fd,"ANONYMIZE", HTTP, URI,  
    REPLACE, STR, "www.abcd.com");  
  
//let's anonymize FTP as well  
//replace the password  
mapi_apply_function(fd,"ANONYMIZE", FTP, PASS,  
    REPLACE, STR, "mypassword");  
  
//replace the filename transferred, "RETR" is the  
//ftp command for fetching a remote file  
mapi_apply_function(fd,"ANONYMIZE", FTP, RETR,  
    REPLACE, STR, "a_file_transferred");  
  
mapi_apply_function(fd,"ANONYMIZE", IP, CHECKSUM,  
    CHECKSUM_ADJUST);  
  
}
```

Bibliography

- [1] The Click Modular Router Project. <http://www.pdos.lcs.mit.edu/click/>.
- [2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote trust-management system version 2. *Network Working Group, RFC 2704*, Sept. 1999.
- [3] T. Brekne, A. Oslebo, and A. Arnes. Anonymization of ip traffic monitoring data—attacks on two prefix-preserving anonymization schemes and some proposed remedies. In *To appear in the Proceedings of Privacy Enhancing Technologies*, 2005.
- [4] College of Computing, Georgia Tech. Cryptography-based Prefix-preserving Anonymization. <http://www.cc.gatech.edu/computing/Telecomm/cryptopan>.
- [5] Dartmouth College. Archive of wireless-network trace data. <http://cmc.cs.dartmouth.edu/data/index.html>.
- [6] Dave Plonka. ip2anonip. <http://dave.plonka.us/ip2anonip>.
- [7] Eddie Kohler. ipsumdump. <http://www.cs.ucla.edu/~kohler/ipsumdump>.
- [8] Greg Minshall. Tcpsdpriv. <http://ita.ee.lbl.gov/html/contrib/tcpsdpriv.html>.
- [9] Lawrence Berkeley National Laboratory. Bro Intrusion Detection System. <http://www.bro-ids.org>.
- [10] LBL. LBL Internet Traffic Archive. <http://ita.ee.lbl.gov/html/traces.html>.
- [11] J. Mogul. Trace anonymization misses the point. 2002.
- [12] NLANR. PMA NLANR traces. <http://pma.nlanr.net/PMA/Traces>.
- [13] 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.
- [14] G. Portokalidis and H. Bos. Admission control daemon, 2004. <http://admctrl.sourceforge.net>.
- [15] R. Ramaswamy, N. Weng, and T. Wolf. An IXA-based network measurement node. In *Proc. of Intel IXA University Summit*, Hudson, MA, Sept. 2004.
- [16] The SCAMPI Consortium. Deliverable D1.3: Final Architecture Design, Nov. 2003. <http://www.ist-scampi.org/publications/deliverables/D1.3.pdf>.
- [17] The SCAMPI Consortium. Deliverable D2.3: Enhanced SCAMPI Implementation and Applications, Apr. 2004. <http://www.ist-scampi.org/publications/deliverables/D2.3.pdf>.
- [18] UCLA. UCLA CSD Packet Traces. <http://lever.cs.ucla.edu/ddos/traces/>.
- [19] UCSD. Wireless LAN Traces. <http://ramp.ucsd.edu/pawn/sigcomm-trace/>.

BIBLIOGRAPHY

- [20] 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.
- [21] 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.
- [22] 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.
- [23] T. Ylonen. Thoughts on how to mount an attack on tcpdprivs "-a50" option. <http://ita.ee.lbl.gov/html/contrib/attack50/attack50.html>.