A Generic Anonymization Framework for Network Traffic

D. Koukis, S. Antonatos, D. Antoniades, E.P. Markatos
Institute of Computer Science (ICS)
Foundation for Research
& Technology – Hellas (FORTH)
P.O. Box 1385, Heraklion Crete,
GR-71110, GREECE
{koukis,antonat,danton,markatos}@ics.forth.gr

P. Trimintzios
European Network and Information
Security Agency (ENISA)
P.O. Box 1309, Heraklion Crete,
GR-71001, GREECE
panagiotis.trimintzios@enisa.eu.int

Abstract

Lack of trust is one of the main reasons for the limited cooperation between different organizations. The privacy *of users is of paramount importance to administrators and organizations, which are reluctant to cooperate between each other and exchange network traffic traces. The main reasons behind reluctance to exchange monitored data are the protection of the users' privacy and the fear of information leakage about the internal infrastructure. Anonymization is the technique to overcome this reluctance and enhance the cooperation between different organizations with the smooth exchange of monitored data. 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 allowing fine-tuning of privacy protection level. The proposed framework is composed by an anonymization application programming interface (AAPI). The performance results show that AAPI outperforms existing tools, while offering significantly more anonymization primitives.*

1 Introduction

Monitored network traffic and traces are powerful means for evaluation experiments, allowing the researchers to study network characteristics and behavior. Furthermore, network traces are used by network security people, in evaluating intrusion detection systems for example. In the ideal case, 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 is known as the anonymization process.

The anonymization process has three objectives. First, to protect the *privacy* of monitored users. Revealing sensitive information about the users is totally prohibitive. 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-mail 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 packet payload, are either completely removed or replaced by random values.

Second objective is to hide any information about the internal infrastructure of the network. Ideally, the 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 operating system identification. Also people which access 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 header fields with constant values that might reveal any of the network characteristics. Other approaches, like encrypting IP addresses in prefix-preserving way [24], are subject to network information leakage.

Finally, anonymized traffic traces have to be as realistic as possible, that means as close as possible to the non-anonymized packet stream. Many evaluation experiments done by researchers rely on monitored traffic traces, thus the results have to be close to those taken by plain traffic from the network. As most anonymized traces that are currently publicly available are unrealistic, 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 could not exist since different organizations have different needs. Network administrators should be able to specify their anonymization policies at varying levels of detail granularity. Existing anonymization tools are not adequate enough to provide such flexibility and are not capable to address all anonymization needs, since most of the times 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.

In this work we propose and evaluate an anonymization framework, which 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 our 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 the anonymization policy definition a quick and simple process our framework provides an Application Programming Interface (API) named `AAPI`. AAPI is simple to use since 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 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 [5] records. The performance of AAPI, in terms of processing speed and resource needs, is comparable to other, much simpler, anonymization tools (such as tcpdpriv) that offer less functionality.

The rest of this paper is organized as follows. In Section 2 we present the related work. In Section 3 we describe the detailed design of our generic anonymization framework, and in Section 4 we evaluate our framework in terms of performance and expressiveness. Finally, we summarize our work and results in Section 5.

2 Related Work

`Tcpdpriv` [12] is the most known anonymization tool. It takes as input traces written in `tcpdump` [3] format and removes sensitive information by operating only on packet headers. TCP and UDP payload is simply removed, while the entire IP payload is discarded for protocols other than TCP or UDP. The tool provides multiple levels of anonymization, from leaving fields unchanged up to performing more strict anonymization, like mapping IP addresses to integers or prefix-preserving anonymization. Ip2anonip [8], a tool based on `tcpdpriv`, is a simple filter that turns IP addresses into host names or anonymous IPs. Ipsumdump [9] dumps packets into `ASCII` format and uses `tcpdpriv` to anonymize IP addresses if specified by the user.

The main drawback of all the above tools is that they 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 which can be applied only to a few predefined fields such as IP addresses and TCP ports.

Peuhkuri in [17] deals with persistent anonymization of IP address among different packet traces. The proposed algorithm is only for anonymization of IP addresses. This algorithm makes use of cryptography, thus the mapped address is produced by merging a part of the original address with a value encrypted with a key provided by the user. This approach maintains the mapping of addresses to encrypted values along anonymization sessions, however its functionality is limited to IP addresses only.

Xu, Fan, Ammar et al. in [24, 25] focus on the problem of prefix-preserving IP anonymization. Existing implementations, such as `tcpdpriv` [12], have many drawbacks like memory consumption, inconsistent mappings across different anonymization sessions and lack of parallel processing of traces. The approach described in [24, 25] uses stateless cryptography algorithms that require small memory amount. 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 from the original– subnet. As prefix-preserving mapping is a stateless function applied to IP addresses, while parallel anonymization is also feasible. An implementation for prefix-preserving anonymization, called Crypto-PAn [6], is publicly available from Georgia Tech University. Although this implementation has several advantages compared to `tcpdpriv`, including prefix-preservation, memory consumption and consistent mapping across traces, its functionality is also limited only to IP addresses, thus it can not be considered as a full anonymization suite that is able to express *any* anonymization policy.

Prefix-preserving anonymization has also been applied to Netflow [23]. The Crypto-PAn software has been used and 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 [22] where the users are able to anonymize the eight most common fields of a NetFlow record.

Paxson and Pang in [16] introduce a way to anonymize the payload of a packet and remove sensitive information instead of removing the entire payload as the other approaches do. Packets are reconstructed into data stream flows and application level parsers modify the data streams as specified by a policy written in a high-level language. The user can specify the field to be altered using regular expressions and the modification to be done. After the anonymization of the stream has taken place, stream is split again into packets and merged with original packet headers, thus creating legitimate traffic. In this way, anonymization and reconstruction process becomes a transparent procedure for streams.

The anonymization process described in [16] has been implemented as a plug-in for `Bro` [15], a Unix-based Network Intrusion Detection System (IDS), thus permitting it to anonymize both on-line traffic as well as offline traces. Although this approach is quite flexible, it has several limitations and drawbacks. First, it provides limited anonymization primitives —constant substitution, sequential numbering, hashing, prefix-preserving and adding random noise—, forcing the user to write his own functions in `Bro` language, a custom scripting language. Our framework, on the contrary, 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. Secondly, 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.). Using our API, implementing anonymization policies is a matter of few lines of code. To ease the users we also provide separate tools that incorporate all the functionality of our framework.

Work on anonymity has also been done in [11], [19], [13] but in a different context. Their perspective is to hide the identity of the sender/recipient of a message and provide anonymity and privacy to users of these infrastructures.

3 The Anonymization Application Programming Interface

The need for anonymization policies may vary from very simple policies, like removing payload and sequential numbering of IP addresses, up to complex policies, 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 and record traces.

The AAPI is an API based on 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 the designers' point of view, libraries that capture traffic are also written in C, thus the AAPI can directly communicate with them. From the 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, performance of the anonymization process is very critical, especially in case of anonymizing realtime traffic at very high speeds.

3.1 Functions of AAPI

A central notion of 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, we have the anonymization functions that *alter fields* of the packets or records in the given traffic stream, e.g randomize them or replace them, do prefix-preserving anonymization on IP addresses, etc. Second, we have the *filtering* functions, that are BPF filters and string searching. 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

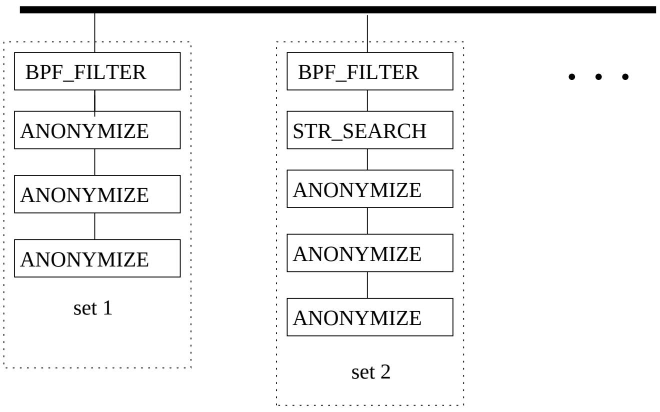

Figure 1. Function sets: Each packet is passed through each set and for each set is processed by its functions

the GNUTELLA-CONNECT pattern". Finally, we have *application-level stream* functions, which we call *cooking* and *uncooking*, that provide our framework the ability to compose and decompose application-level streams.

The main function call of AAPI is the `add_function(set, function, ...)`, where `'...'` denotes variable number of arguments, depending on the specific function to be applied. 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 one after the other, in the order they had been applied. Sets are created through the `create_set()` function. Once a packet is captured, is passed through each set and for each set is processed by its functions. We should note here that 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 express "`if-else`" scenarios or even express different anonymization policies within the same program. The function sets are visualized in Figure 1.

The argument `function` defines which specific function will be applied. Natively, AAPI supports "ANONYMIZE" (field anonymization),"BPF_FILTER" (BPF filtering), "STR_SEARCH" (string searching),"COOK" (stream reassembly) and "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, the `add_function` is refined as `add_function(set, function, protocol, field, parameters)`. `function` in this case specifies the particular anonymization function will be applied. 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, and **removing** fields mainly used for application-level protocols, thus providing adequate functionality for every user needs. Moreover new functionality can be added by the user as described later.

The parameter `protocol` describes which specific protocol and layer the anonymization function will work on. Our current implementation supports IP, TCP, UDP, ICMP, HTTP or FTP. At application level currently we 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.

Finally, the last parameter 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. 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 user for wrong usage of functions.

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 observe 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(set1,"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 on 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. It is thus highly and strongly recommend 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 contained 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 "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 them the appropriate portion of the payload of the cooked and anonymized packet. In that way, "UNCOOK" constructs as many TCP/IP packets as they were originally in the incoming traffic, with each one having the same header as before the application of cooking, though the payload will be anonymized. It must be noted that after the uncooking some of the TCP/IP packets of original incoming 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. The 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 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 packets 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 the 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 removed since it is useless. The proper ordering of anonymization functions is illustrated in Figure 2.

3.4 Extensibility

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

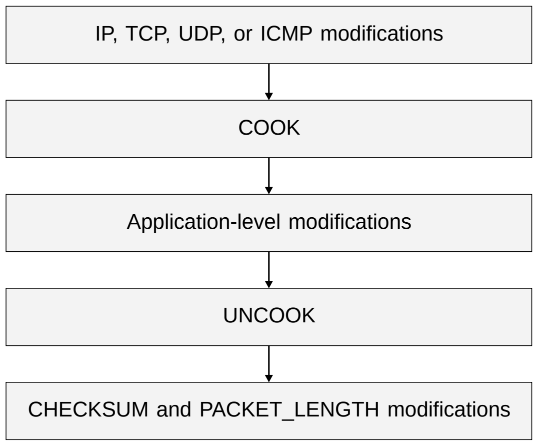
IP, TCP, UDP, or ICMP modifications
COOK
Application-level modifications
UNCOOK
CHECKSUM and PACKET_LENGTH modifications

Figure 2. The order of functions after applying reordering

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 think and add to AAPI a new anonymization function for IP addresses that hashes the first 8 bits and randomize the rest. Moreover, we provide a callback functionality, meaning that the user can specify a function that is called for each packet, therefore get raw access to packets.

In our current implementation the only 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 the content from emails.

Finally, it is straight forward to add support of different input sources. For example, snort alert logs [20] can be supported by simply adding a new decoder that reads such records and provide reference to each field. Since our framework is generic in the sense that 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 relative fields. This way the framework can anonymize snort alert logs, without any other change, using the same notation and anonymization functions described in this paper.

3.5 Input and Output Functionality

Our AAPI works both offline with traces as well as online with real traffic. The framework natively supports live traffic from standard Ethernet interfaces as well as DAG cards [10]. In case of offline traces, we support the `tcpdump` and DAG format traces. 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 anonymization output 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 output traces by 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.

3.6 Integration with Passive Monitoring

AAPI has been integrated with a real-time generic passive monitoring framework. In this section we give a brief description of the passive monitoring framework and we list the reasons that lead us to integrate it with AAPI.

The Monitoring API (MAPI) [2] was designed by the IST project SCAMPI [1], and is presented in [18]. 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.

AAPI was initially designed as a stand-alone framework. We did decide 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 to a distributed monitoring infrastructure, so the need for privacy is more than necessary. Using the AAPI integrated in MAPI users can be sure that no sensitive data is revealed to others. Also the administrator can enforce certain anonymization policies to all users using the monitoring sensor.

Also the AAPI can gain from the advantages of MAPI. MAPI supports the collection of data from additional hardware interfaces -such as the SCAMPI card [7]. Also, some of the basic anonymization procedures could be implemented by hardware in the near future, so the real-time anonymization at very high speeds will be achievable.

4 Performance Evaluation

In order to measure the performance of the framework, we have implemented some simple anonymization policies both as AAPI applications, as well as `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*". Also in order to check 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.

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

Table 1. Performance comparison between tcpdpriv and AAPI-based anonymization

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

Table 2. Performance comparison between Bro and AAPI-based anonymization

The AAPI allows us to implement the anonymization functionality in less than 40 lines of code. In the experiment we used a P4 at 3.0 GHz with 512 MB main memory. We used a 2 GB tcpdump trace as traffic input for anonymization, consisted exclusively of TCP packets from a web portal mirroring. In Table 1 we show the user time in seconds for both the AAPI-based and the `tcpdpriv` applications. As it can be observed, that 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 for the performance of the tool since `tcpdpriv` is a highly specialized tool for simple anonymization policies without offering all the functionality supported by AAPI.

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 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; more specifically the policy is: "*replace the URL in HTTP packets with the string SAMPLE_URL*". In Table 2 we present the measured user time to complete the 2 GB trace anonymization in seconds.

Due to the limitations of `Bro` discussed in section 2, even for a simple action like simply changing the IP address in a packet, it had to reassemble the trace up to the

HTTP level before it could anonymize the IP addresses. In the case of mapping IP addresses, our approach is up to 30 times faster than `Bro` system, as we do not have to do stream reassembly. In the case of URL replacement, where both tools require to perform cooking, the AAPI application needs about half the time required by the `Bro` application. AAPI is faster because it is a framework specially designed for anonymization in contrast with `Bro` that is an IDS system and therefore functionality useless to anonymization introduces this overhead.

4.1 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 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 stored on disks this performance complexity may not be an issue, 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 GBps links.

Infrastructures that work with live traffic from diverse administrative domains, such as zero-day worm detection systems [4], [21], emerge 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 to the the reader an insight of what is the performance penalty of each one of them. All experiments were performed on a PC with P4 processor at 3.0 GHz, with 512 MB main memory. As input source we used a 2 GB tcpdump trace consisting of HTTP traffic.

Our results are summarized in Table 3. The metric we use is user time, measured with the `time` command-line utility. In our effort to express the results in terms of MBps, we replayed the trace on a Gigabit speed link. The actual transmission rate reached 630 MBps, a speed at which all functions, except prefix-preserving, could handle the traffic without inducing any packet loss.

The prefix-preserving function is based on the Crypto-PAn package, while prefix-preserving-map is a much simpler algorithm for prefix-preserving anonymization without using cryptographic methods. The prefix-preserving function based on the Crypto-PAn package presents low performance. The optimization of this function is left for future work.

Function	*User time*
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 3. Cost of basic anonymization functions

4.2 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 the other hand, the information we delete from 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 the minimum needed information. The policy we want to apply 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 we use is the number of alerts that were generated by the Snort [20] intrusion detection system. Our input was a 400 MB trace, which was collected during the DARPA evaluation test [14].

We anonymized the trace with both `tcpdpriv` and with a simple application based on AAPI. We passed both the output traces to Snort. We ran Snort with two different sets of rules: one that contains header-only rules, i.e., rules that do not need access to the packet payload, and one with content rules, i.e., rules that require access to both headers and payload. The results are presented in Table 4. The last row denotes the sum of the two cases.

	tcpdpriv	*AAPI*	*Plain*
header-only rules	45	45	45
content rules	0	527	1892
complete snort ruleset	45	572	1937

Table 4. Number of alerts produced by Snort IDS for web trace

As we can see, the `tcpdpriv` approach preserves only a small percentage on the initial alerts, derived solely from header rules. AAPI on the other hand, uses a less strict approach, creates a much more useful trace.

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 does not. This way, alerts will be preserved in the output trace and on the other hand 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 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.

5 Summary and Concluding Remarks

In this paper we have presented the design and effectiveness of a generic anonymization framework. The key point is configurability where the user can define any anonymization policy as a series of functions that are applied on packets. Our 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. The usefulness 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 parallel trying to optimize the necessary functions. All in all this work constitutes currently the most complete framework for anonymization of realtime traffic and offline traces. Furthermore, the framework is implemented in a modular way so it is fully extensible in terms of functionality, protocols and new traffic sources. Finally, we measured 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.

We intend to enhance the performance of the framework in the future by parallelism of the existing algorithms, and hardware support.

6 Availability

An implementation of AAPI, along with a fully functional application based on it, can be found at `http://www.ics.forth.gr/~koukis/aapi.html`

Acknowledgments

This work was supported in part by the IST project LOBSTER funded by the European Union under contract number 004336, and in part by the project CyberScope funded by the Greek General Secretariat for Research and Development under contract number PENED 03ED440. Dimitris Koukis, Spiros Antonatos, Demetres Antoniades, and Evangelos P. Markatos are also with the University of Crete. This work was done while Panagiotis Trimintzios was at ICS-FORTH, and does not express the opinion of his current employer. We would like to thank Kostas Anagnostakis and the members of the DCS Lab at FORTH-ICS for their valuable assistance and support.

References

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

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

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

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

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

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

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

[11] Michael J. Freedman, Emil Sit, Josh Cates, and Robert 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.

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

[13] 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`.

[14] MIT Lincoln Laboratory. Darpa intrusion detection evaluation data sets. `http://www.ll.mit.edu/IST/ideval/data/data_index.html`.

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

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

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

[18] M. Polychronakis, K. G. Anagnostakis, E. P. Markatos, and Arne 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.

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

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

[21] 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*, December 2004.

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

[23] Adam Slagell Jun Wang and William Yurcik. Network log anonymization: Application of crypto-pan to cisco netflows. *NSF/AFRL Workshop on Secure Knowledge Management (SKM)*, 2004.

[24] Jun Xu, Jinliang Fan, Mostafa Ammar, and Sue 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.

[25] Jun Xu, Jinliang Fan, Mostafa Ammar, and Sue B. Moon. Prefix-preserving ip address anonymization: Measurement-based security evaluation and a new cryptography-based scheme. *ICNP 2002*, 2002.