Real-time anonymization in passive network monitoring

Sven Ubik, Petr Žejdl, Jiří Halák
CESNET, Czech Republic

Abstract—**Passive network monitoring that observes user traffic has many advantages over active monitoring that uses test packets. It can provide characteristics of real user traffic, that cannot be detected actively.**

However, when processing user traffic, we must guarantee user privacy. This is a task of packet header anonymization that removes sensitive information, while keeping as much as possible of the original traffic properties.

In this paper we present design and implementation of an FPGA-based packet header anonymization that unlike previous approaches operates in real time and prevents sensitive information from getting to the monitoring PC and beyond.

Keywords: passive network monitoring, packet header anonymization

I. Purpose of packet anonymization

In passive network monitoring we directly process real user traffic, as oposed to active monitoring, when we use injected test traffic. Passive monitoring allows to detect properties of real traffic, such as security attacks, traffic dynamics or real packet loss rate. Packet traces are also useful resource for networking research.

When processing user traffic, we have to secure user privacy. Particularly, we need to remove packet payload (or most of it, just except indicators of possible security attacks) and we need to modify packets headers.

The purpose of packet anonymization is to remove all sensitive information from the monitored traffic to protect user privacy, while keeping as much as possible of the original traffic characteristics so that the monitoring applications can achieve their mission.

In this paper we describe how anonymization is implemented in the LOBSTER passive monitoring architecture, with particular attention to the first-tier real-time hardware-supported anonymization.

In section II we mention the most important related work. In section III we state the requirements set for our solution. In section IV we describe design and implementation of our system. Then we present an example of use in section V and we summarize performance characteristics and required resources in session VI.

II. Related work

Anonymization can be done on a reconstructed stream, which is then again split into packets [1]. This type of anonymization can do advanced processing of higher-layer protocols, such HTTP. However, this approach is compute-intensive, it can not be used at gigabit speeds and it looses many of the original traffic dynamics (packet spaces, bad CRCs, IP options, etc.).

A table-based approach to IP address anonymization was implemented in TCPdpriv [2]. A cryptography-based scheme to provide consistency of anonymization across different traces using the same cryptography key was described in [3]. However, software implementation is slow for gigabit speeds and it requires that sensitive information is temporarily stored in a monitoring PC.

Hardware implementation on a network processor using precomputed mapping trees [4] can remove sensitive information in the monitoring hardware, but it still requires a lot of instructions and several memory accesses per packet. The measured throughput for 40-byte UDP datagrams was 65000 packets per second, which is approximately 50 Mb/s at a wire level.

III. Requirements

We designed and implemented hardware anonymization as part of the LOBSTER [5] project. The goal of LOBSTER is to enhance passive network monitoring architecture developed by the preceeding SCAMPI [6] project and to deploy it in a European scale.

As part of the LOBSTER project, we conducted a review [7] of user requirements on packet anonymization. Most people are only willing to share data about their network traffic in the form of statistics or after proper anonymization. This was in most cases expressed as payload removal and IP address hashing.

We defined the following set of requirements to be fulfilled by our implementation of packet anonymization:

- Anonymization must provide a wide range of easily configurable possibilities of removal of sensitive information.
- Anonymization up to TCP and UDP headers must be implemented in a hardware monitoring adapter for high speed and to remove sensitive information before it gets to the host PC.
- IP address mapping must be consistent among traces - the same real IP address in two traces must be mapped into the same anonymized IP address. Multicast addresses must be mapped into anonymized multicast addresses.
- IP address mapping must be also prefix-preserving. Two real IP addresses that share a common prefix must be mapped to two anonymized IP addresses that also share a common prefix of the same length.
- The architecture must allow follow-up software anonymization of higher-layer protocols.

IV. IMPLEMENTATION

Packet processing in MAPI (Monitoring Application Programmable Interface), which is the central part of the LOBSTER architecture, is conceptually based on flows and monitoring functions. An application opens one ore more *flows*, which are initially all packets coming to one or more specified network interfaces (in case of multiple interfaces it is called a *scope*). The application then applies a sequence of *monitoring functions* on each flow. The selection and order of monitoring functions determine the resulting functionality.

A. Two-layer anonymization

MAPI can run on top of different monitoring adapters, with different hardware functionality. Implementation of monitoring functions for each adapter is in a separate library. Each function includes device ID, which defines what adapters this function can run on. Functions in stdflib library do not use any hardware acceleration and run on all adapters. At the application start-up, MAPI selects implementations of all applied monitoring functions depending on adapters used and on the order of monitoring functions. (hardware-supported versions can usually be used only when they appear in certain order in the application).

Anonymization is implemented by the ANONYMIZE monitoring function. The anonymization policy is determined by a sequence of ANONYMIZE functions applied to a flow, for example, the following functions apply prefix-preserving mapping to source IP addresses, map destination TCP port to a constant and strips the URI in the HTTP header:

```
mapi_apply_function(fd, "ANONYMIZE",
   "IP, SRC_IP, PREFIX_PRESERVING);
mapi_apply_function(fd, "ANONYMIZE",
   "TCP, DST_PORT, MAP, 0x2694);
mapi_apply_function(fd, "ANONYMIZE",
   "HTTP, URI, STRIP);
```

First-layer anonymization functions that do not require reconstruction of the original data stream are performed in high speed in hardware when MAPI runs on top of programmable COMBO card.

Second-layer anonymization functions that operate on a reconstructed data stream (e.g., HTTP anonymization) are implemented in software, as well as all other functions when MAPI runs on a network adapter that does not include hardware support for anonymization.

The selection of hardware or software implementation for each anonymization function is done by MAPI transparently to the application.

B. Overview of hardware anonymization

The family of COMBO cards was developed in Liberouter [8] and SCAMPI [6] projects and as such they allow us to make modifications to their firmware. COMBO cards are PCI cards for PC equipped with Gigabit Ethernet ports and the Virtex II FPGA circuit to process packets arriving from the network. The SCAMPI firmware implements advanced packet processing including packet classification, sampling and statistics.

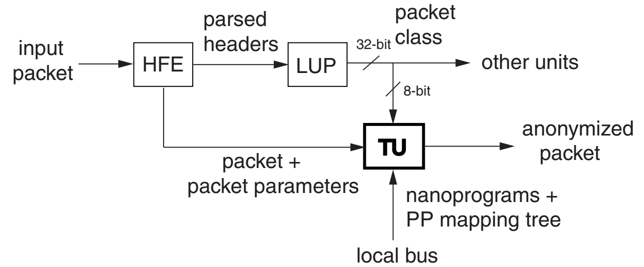

Fig. 1. Position of anonymization in packet processing

Anonymization is a kind of data transformation. Therefore, we designed a universal *Transformation Unit (TU)* for general packet transformations, which can be used to anonymize selected information inside packets. The TU unit can be used as a standalone IP core when we provide required data to its input signals.

The SCAMPI firmware consists of several units. The position of the TU unit in the SCAMPI firmware is illustrated in Fig. 1, which includes only units important to our discussion.

Input packets go to the Header Field Extractor (HFE), which parses packet headers, stores them in internal data structures and prepares *packet parameters*, which are pointers to important sections of the packet, such as positions of selected header fields. A table of these packet parameters is added in front of each packet. HFE operation is programmable and we modified it such that the packet parameters include pointers to all important sections of the packet that we want to anonymize.

Parsed headers come to the Lookup Processor (LUP), which sorts packets into 256 classes based on header fields and assigns each packet a 32-bit control word. This control word indicates which Sampling Units (SAU), Statistical Units (STU) and which parts of the Payload Checker (PCK) should further process the packet. Due to limitations in hardware design we could not use a wider control word. Therefore, we reused an 8-bit part of the control word indicating the STU number to also indicate a class of packets for anonymization purposes. Different anonymization can be applied to packets in different classes. Packets which have been assigned some non-zero STU number are passed to the TU unit. Packets which have been assigned zero STU number do not go to the TU unit and are not transformed.

C. Transformation Unit (TU)

The TU unit is designed as a small processor interpreting programs in a simple instruction set. These programs describe what anonymization should be performed for each header field for packets in each class.

The TU unit can do the following types of header field transformations:

- set to a specified constant
- set to a pseudorandom number
- xor with a specified constant
- table-based hashing
- prefix-preserving mapping

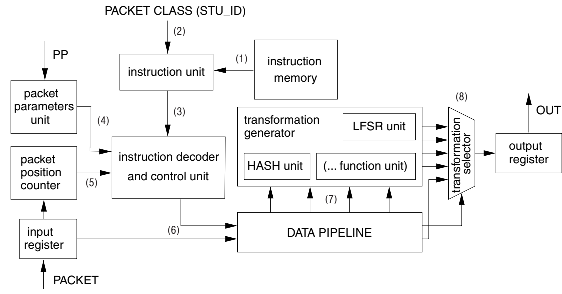

Fig. 2. Transformation Unit (TU)

- any combination of the above (e.g., first half of IP address can be set to a constant and second half randomized)

Each of the above types of anonymization can be applied to any 16-bit header field in the packet. Two flags in the TU instruction (see below) can be used to mask out left or right half of the 16-bit header field and request that the anonymization function applies to an 8-bit header field only. Each header field can be identified by an offset (in 16-bit words) from one of the following packet parameters:

- PP_ETHER - Ethernet header
- PP_ARP - ARP header
- PP_ICMP - ICMP header
- PP_ICMPv6 - ICMPv6 header
- PP_IPv4 - IPv4 header
- PP_IPv6 - IPv6 header
- PP_UDP - UDP header
- PP_TCP - TCP header

For example, you can identify source IPv4 address by using PP_IPv4 packet parameter and offsets 6 and 7.

D. TU operation

The TU unit structure is shown in Fig. 2. Anonymization programs are stored in the instruction memory (1). A packet class assigned by LUP is used to select a corresponding anonymization program (2). Each instruction is then decoded (3), the required packet parameter (offset to some key header field) is retrieved (4), internally added with direct offset specified in the instruction and compared with the current packet position (5). The packet itself arrives through the input register in 16-bit chunks. These chunks are associated with the transformation description (6) and passed to the data pipeline. Different anonymization functions (7) require different number of clock cycles to complete. Therefore, the input data is passed to each function from the different stage of the pipeline such that all results arrive at the right time to the multiplexor (8), which selects the function that should be applied to the current header field.

E. Prefix-preserving IP address mapping

The Prefix-Preserving Mapping Unit was designed for IP addresses, but it can operate on any 32-bit chunks of data.

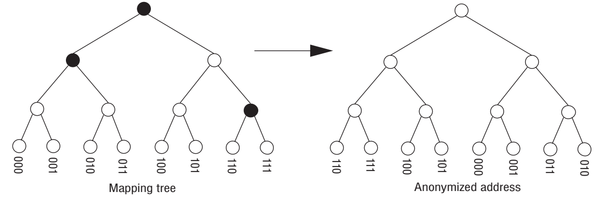

Fig. 3. Prefix-preserving mapping by swapping tree nodes

Original address	XOR pattern	Anonymized address
000	110	110
001	110	111
010	110	100
011	110	101
100	100	000
101	100	001
110	101	011
111	101	010

TABLE I
PREFIX-PRESERVING MAPPING BY XOR OPERATION

The mapping between the original and anonymized IP address is based on swapping subtrees of a tree representation of IP addresses [3]. The method is illustrated in Fig. 3 for 3-bit numbers. The set of all 3-bit numbers can be represented by leaf nodes or by paths going from the root node to the leaf nodes in a 3-level binary tree. When we take bits of a binary number from left to right, then the path from the root node goes left for 0 bit and goes right for 1 bit.

Now we can mark some nodes — in our example by black color. When we swap bits in original IP addresses corresponding to the marked nodes, we get anonymized IP addresses and the mapping between original and anonymized IP addresses is prefix-preserving. We can mark any nodes in a tree to get prefix-preserving mapping, but certainly some markings produce low-quality anonymization, such as when very few nodes are marked or not marked. A common method is to mark nodes pseudorandomly or as a result of cryptographic function applied to a key.

The flip or non flip action can be implemented in hardware by XOR operation of the original bit with 1 or 0 bit, respectively. The mapping and XOR patterns for our example are shown in Table I. Note that the property of this method is that the first bit of an IP address is always either swapped or not swapped (it does not depend on the IP address) and that the last bit of an IP address is not used to select mapping.

F. Memory organisation

To store marks of the whole tree, we need at least one bit for each non-leaf node. That is for 32-bit IP addresses we need at least $2^{32} - 1$ bits = approx. 512 MB of memory. For practical implementation it is more convenient to store each path from the root to a leaf as a separate 32-bit word which can be directly XORed with the original IP address to get the anonymized address.

In this representation we would need 16 GB of memory.

The volume of required memory can be reduced by storing only a part of the mapping tree and replicating it. This method was first proposed in [4]. In real packet traces we normally do not have all 2^{32} IP addresses. The number of different IP addresses present is much smaller. Therefore, if we store only a subset of the mapping tree and replicate it, chances are that we do not use many duplicates.

In order to read a mapping path for the whole IP address as fast as possible and to allow configuration of the size of the stored tree subset, we divided the 32-bit tree into 8-bit subtrees and store these subtrees inside FPGA as illustrated in Fig. 4. We use two dual-port BRAMs, which act as four independent memories. The data width of each BRAM is 8 bits and the address width is configurable and it is at least 11 bits. There are three configuration options.

Fig. 4 a) shows the case when all BRAMs have address width of 11-bits. Each byte of the original IP address is used as address to retrieve mapping from one of the BRAMs. Only seven bits from each byte are used to direct anonymization (see the description of the mapping tree above). Therefore, there are some spare bits in the BRAM's address space. These spare bits are connected to some of the bits of the previous byte in the IP address. In this way anonymization of one byte is influenced at least by a part of the value of the previous byte. Spare bits of the first BRAM are connected to fixed values. Each additional available address bit doubles the stored subset of the whole mapping tree

Fig. 4 b) shows the case when the address space of the BRAMs for the two lower levels is larger than for the two upper levels. This is desirable because the lower levels need more stored subtrees if we do not want to increase replication. More bits from the second and third bytes of the IP address are used to influence anonymization of next bytes. In practical implementation we use at least 2 BRAMs for the upper levels and 6 BRAMs for the lower layers.

A consequence of using separate BRAMs is that the part of the first BRAM is not used and subtrees for each level can be replicated from only one BRAM. An alternative solution shown in Fig. 4 c) is to use one larger BRAM. We can use the whole address space of this BRAM to select subtrees for all four levels, but we need to read them in four clock cycles one after another. As speed was our primary goal we use two separate dual-port BRAMs at the cost of replicating subtrees only within each two levels (option b).

The mapping tree must be loaded into BRAMs before enabling the TU unit. We connected BRAMs to the local bus in the SCAMPI design. The data width of the local bus is 16 bits. Lower 8 bits are sent to the first BRAM and the upper 8 bits are sent to the second BRAM.

V. Example of Use

Suppose that we want to do the following anonymization:

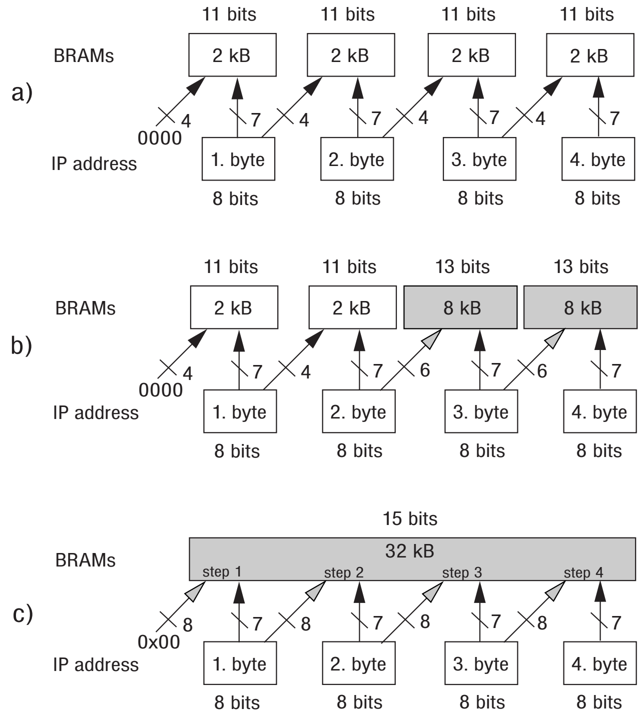

Fig. 4. Storing mapping trees in BRAMs inside FPGA

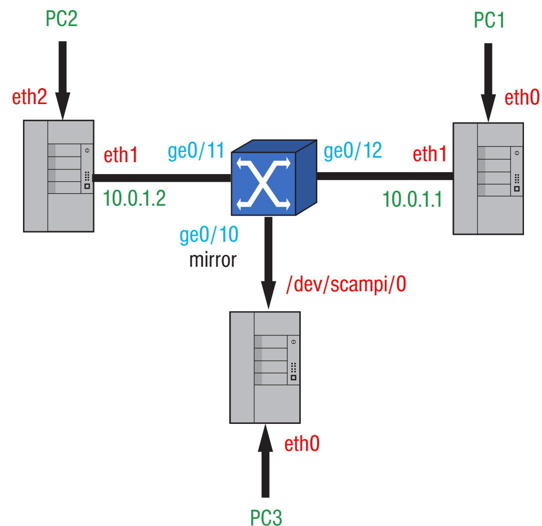

Fig. 5. Setup to test packet header anonymization

- Hash the source IP address
- Keep the first part of the destination IP address
- Randomize the second part of the destination IP address
- Set the source port to constant 9876
- XOR the destination port with constant 0x1234

We used the setup shown in Fig. 5 to test the configuration. PC1 sends UDP packets to PC2. These packets are captured by PC3. Before we enabled anonymization, the packet capture looked as follows:

```
IP 10.0.1.2.2000 > 10.0.1.1.2000: UDP
IP 10.0.1.2.2000 > 10.0.1.1.2000: UDP
IP 10.0.1.2.2000 > 10.0.1.1.2000: UDP
IP 10.0.1.2.2000 > 10.0.1.1.2000: UDP
IP 10.0.1.2.2000 > 10.0.1.1.2000: UDP
```

After we enabled anonymization, the capture of the same packets looked as follows:

```
IP 0.32.48.96.9876 > 10.0.68.143.13250: UDP
IP 0.32.48.96.9876 > 10.0.200.43.13250: UDP
IP 0.32.48.96.9876 > 10.0.42.77.13250: UDP
IP 0.32.48.96.9876 > 10.0.131.131.13250: UDP
IP 0.32.48.96.9876 > 10.0.95.134.13250: UDP
```

VI. Performance and resources

Packets are passed through the TU unit at the rate of 16 bits per clock cycle. We need additional 20 clock cycles per packet (9 cycles to retrieve packet parameters, 9 cycles to fill the data pipeline and 1 cycle to initialize the instruction pipeline). Throughput depends on the packets size and it can be computed by the following formula:

$$throughput = \frac{length + gap}{length + gap + 2 * delay} * (16 * clockrate)$$

where:

- *throughput* in bits/s is measured at the wire level
- *length* of packet in bytes includes Ethernet header and CRC
- *gap* includes the interframe gap, the preambule and the start-of-frame delimiter and it is equivalent to 20 bytes
- *delay* corresponds to the 20 clock cycles of overhead per packet; as 2 bytes are normally processed in one clock cycle, we multiple it by two and use the result of 40 bytes to represent the delay in the formula as the equivalent number of bytes

The TU unit was synthesized including all constraints at 100 MHz clock rate. Computed throughput depending on the packet size for this clock rate is shown in Fig. 6. Throughput for the worst case of 64-byte packets is 1.08 Gb/s, which is sufficient to process Gigabit Ethernet traffic at line rate.

We tested the TU unit with the COMBO card consisting of the COMBO6 mainboard and the COMBO4MTX interface card. We used SCAMPI phase 1 design version 1_02_07 as the basis for our modifications. We changed HFE program to produce more packet parameters and we integrated our TU unit. The rest of the used SCAMPI design can run at 50 MHz and some units adds more per-packet overhead. Therefore, throughput of the whole modified SCAMPI design, which we measured by a hardware packet generator was lower than the computed throughput of the TU unit alone and is also indicated in Fig. 6.

The consumption of FPGA resources for the TU unit alone and for the whole SCAMPI design with the integrated TU unit is shown in Tab. II.

VII. Conclusion

We implemented easily configurable FPGA-based packet header anonymization that removes sensitive information from packet headers in real time on the monitoring adapter before it can get to the host PC. Anonymization functions can be different for various classes of packets and can include prefix-preserving IP address mapping, which preserves original dynamics in IP address space. The implemented TU unit to perform anonymization can run at full Gigabit Ethernet speed. The whole modified SCAMPI design currently runs at lower speed. We plan to integrate our TU unit in the newer version of design for the COMBO card, which will permit to operate at full Gigabit Ethernet speed.

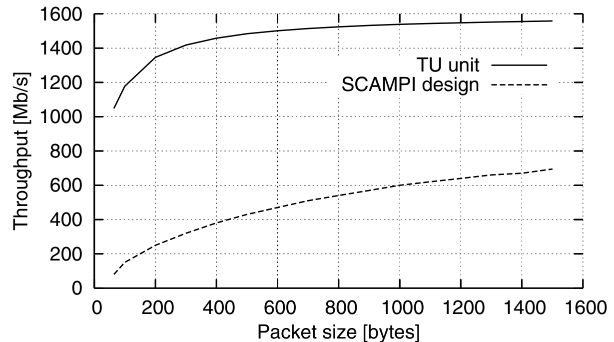

Fig. 6. Throughput of the TU unit and the modified SCAMPI design

	TU unit		Whole design	
	Used	Percentage	Used	Percentage
Slices	956	7%	6238	44%
Flip-flops	1116	4%	6496	23%
4-input LUTs	987	4%	8001	28%
BRAMs	13	14%	46	48%

TABLE II
CONSUMPTION OF FPGA RESOURCES

References

[1] Ruoming Pang, Vern Paxson. *A High-Level Programming Environment for Packet Trace Anonymization and Transformation*, SIGCOMM 2003, August 25-29, 2003, Karlsruhe, Germany.
[2] Greg Minshall. *TCPdpriv*, http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html.
[3] Jinliang Fan, Jun Xu, Mostafa H. Ammar. *Crypto-PAn: Cryptography-based Prefix-preserving Anonymization*, http://www.cc.gatech.edu/computing/Telecomm/cryptopan.
[4] Ramaswamy Ramaswamy, Ning Weng, Tilman Wolf. *An IXA-Based Network Measurement Node*. Proc. of Intel IXA University Summit, 2004. http://www.ecs.umass.edu/ece/wolf/pubs/2004/ixa2004.pdf
[5] LOBSTER project, www.ist-lobster.org.
[6] SCAMPI project (Scaleable Monitoring Architecture for the Internet), http://www.ist-scampi.org.
[7] D0.1 - Requirement Collection and Analysis, LOBSTER project deliverable, http://www.ist-lobster.org/publications/deliverables/D0.1.pdf.
[8] Liberouter project, http://www.liberouter.org.