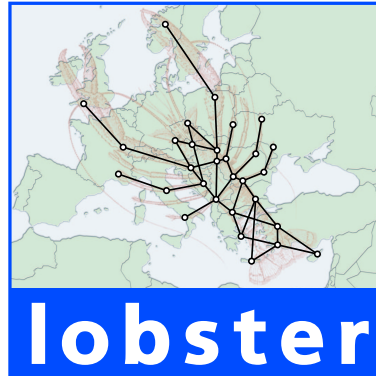


INFORMATION SOCIETY TECHNOLOGIES (IST) PROGRAMME



Large Scale Monitoring Broadband Internet Infrastructure
Contract No. 004336

D1.3 “First-Tier Encryption Definition”

Abstract: In this deliverable we propose architecture for tier-1 hardware anonymization of selected packet header fields.

Contractual Date of Delivery	30 June 2005
Actual Date of Delivery	21 July 2005
Delivarable Security Class	Public
Editor	Sven Ubik
Contributors	CESNET, FORTH, Endace

The LOBSTER Consortium consists of:

FORTH-ICS	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 - two tier anonymization	3
2	Transformation unit	3
3	TU configuration	5
4	TU nanoprocessor	8
5	Prefix-preserving IP address mapping	9
6	Supported anonymizing functions and header fields	11
7	Programming interface	13
8	Requirements on IP address anonymization	14

1 Introduction - two tier anonymization

Lobster architecture will provide anonymization [1] of selected packet parts in a configurable way in order to comply with privacy requirements of data owners and to provide packet traces for researchers.

It is desirable to perform certain anonymization tasks directly in a hardware monitoring adapter used. There are two reasons for that. First, anonymization tasks that apply to most packets, such as IP address mapping, can take a lot of computing resources and doing them in hardware can alleviate host CPU load. Second, when some data is anonymized directly in a hardware monitoring adapter, this information does not get in the host computer at all and thus cannot be compromised by a possible intruder.

However, certain anonymizing tasks are rather complex, such as those operating on reconstructed application-level streams (e.g., replacing HTTP header fields) and cannot be implemented in limited resources of programmable hardware cards.

Therefore, we propose to implement anonymization in two layers (or tiers) - first tier anonymization implemented in firmware of the hardware monitoring card and second tier anonymization implemented in software.

We can also divide anonymization tasks into another two categories (orthogonal to dividing between hardware and software) — packet-level anonymization without regard to flows and flow-based anonymization, with respect to individual flows.

First-tier hardware anonymization will work at packet level. However, certain complex packet-level anonymization tasks, such as certain hashing or encryption algorithms will have to be done in software. On the other hand, SCAMPI firmware units permit classification of incoming packets into 256 flows and thus we can do different packet-level anonymization, such as different treatment of IP addresses for different flows. Tasks on upper-layer protocols (e.g., HTTP) will have to be done in software after reconstructing the stream from packets. In future hardware can look at packet payloads and do certain tasks on upper-layer protocols.

2 Transformation unit

Existing SCAMPI firmware for COMBO hardware card consists of several units. For instance, there is LUP (Lookup Processor) for packet classification, SAU (Sampling Unit) for packet sampling or STU (Statistical Unit) for calculating statistics.

We propose to add Transformation Unit (TU) to the SCAMPI firmware to do general packet transformations, which will be used to anonymize sensitive information by transforming specified packet fields according to configuration. TU will be located after LUP, that is anonymization will start after packet classification and filtering. There are several units after LUP working in parallel in order to minimize time needed to process each packet, namely STU (Statistics Unit), SAU (Sampling Unit), PCK (Payload Checker, only in scampi_ph2 design for 10 Gb/s COMBO card) and the newly added TU. Interrelations of units is illustrated in Fig. 1.

As a result of classification, LUP assigns to each packet a 32-bit wide control word. This control word is divided into three parts going from bit 0 to bit 31:

- 16-bit SAU mask, which specifies which SAU units will process the packet (any combination of 16 SAU units can be selected by setting corresponding bits to 1)

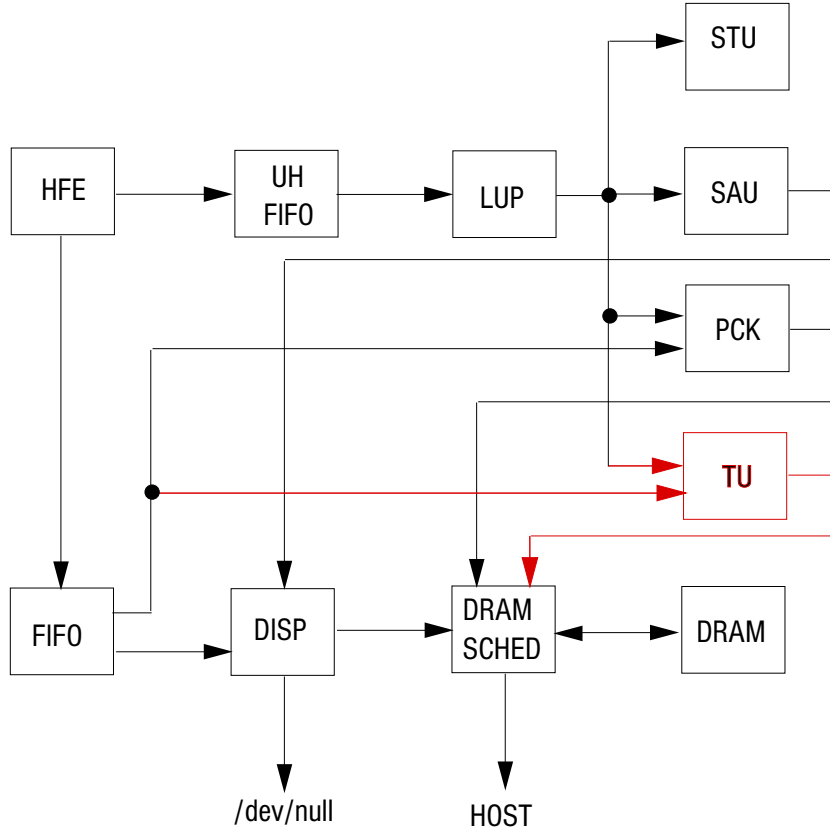


Figure 1: Place of Transformation Unit (TU) in LOBSTER firmware

- 8-bit STU ID, which specified which STU unit will process the packet (one of 256 STU units can be selected)
- 8-bit PCK mask, which is prepended to generated patterns for payload searching, thereby selecting a portion of PCK CAM to be used for matching

All packets that match some filtration rule and thus pass through LUP go to selected STU and SAU units. If we want to fetch the whole packet or its prefix part from the card to the host computer, we must send the packet to at least one SAU. Packets that do not go to any SAU are not sent to the host computer and we can only read statistics about such packets from STU units. On the other hand, packets that go to at least one SAU must be fetched from the card to the host computer, otherwise the buffer on the card will overflow. If we want to get all packets, that is if we do not want to sample them, we must configure the corresponding SAU unit to threshold of one packet and the deterministic mode.

Packets which have non-zero PCK mask are also passed to PCK unit for payload searching (packet which have zero PCK mask do not go to PCK unit). PCK unit is another step of filtration. Packet whose payload does not match any pattern in PCK are discarded. Similarly, packets which have non-zero STU ID are also passed to TU unit for transformations (packet which have zero STU ID do not go to TU unit and are not transformed). Note that even if a packet passes PCK unit and TU unit, it still must also pass at least one SAU unit to be sent to the host computer.

It can be useful if we can apply different anonymization to packets classified by different filters in LUP (Lookup Processor). It would require a lot of resources in FPGA to implement a separate TU unit for different packet classes. Therefore, we will implement only one physical TU unit (a piece of FPGA), which will serve multiple virtual TU units. Each virtual TU unit is a separate data structure specifying packet transformations. The physical TU unit will use a data structure corresponding to the class of each packet. The same concept has already been used for STU unit.

There is no more space in the control word to select virtual TU unit for a packet and we cannot make the control word longer because of limitations in FPGA resources. Therefore, we decided to reuse STU ID part to also select one of 256 virtual TU units.

The structure of the physical TU unit is illustrated in Fig. 2. TU_CORE fetches nanoinstructions from BlockRAM, interprets them and communicates with other blocks using external registers.

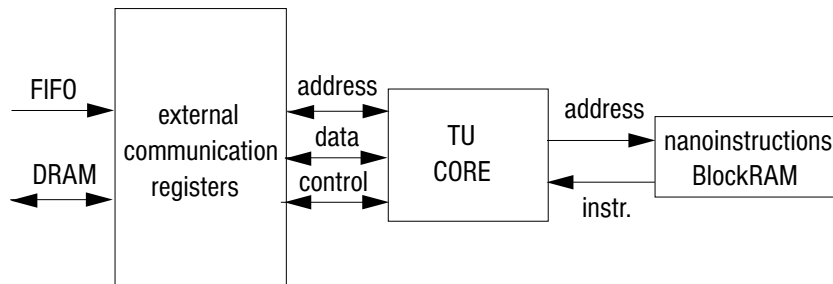


Figure 2: Physical TU unit

3 TU configuration

Transformations performed by TU unit must be configurable. The task that each virtual TU unit should perform can be specified by a sequence of transformation triples:

`<field offset, field length, transformation function>`

The supported packet fields and transformation functions are described in section 6. Some transformation functions are relatively complex. We found that it is not practical to implement such complex functions hardwired in FPGA, because any modification to such a complex function then requires all phases of VHDL redesign. Instead, we will design TU unit as a “nanoprocessor”, which is a simple processor implementing a specialized set of nanoinstructions. Its operation can then be programmed by supplying it with different nanoprograms stored in BlockRAM on the COMBO card. We can compare the solution with hardwired function to the solution with nanoprocessor as follows:

Hardwired functions - advantages

- Higher chance to optimize for speed
- Probably consumes less FPGA resources

Hardwired functions - disadvantages

- Complex FPGA design
- Any function change requires complete sequence of VHDL redesign

Nanoprocessor - advantages

- Easy function modification without VHDL redesign
- Functions can be programmed by software developers, who do not need to know hardware details

Nanoprocessor - disadvantages

- More difficult to optimize for speed
- Probably consumes more FPGA resources

Each transformation function requested in any of the transformation triples for any virtual TU unit will be translated into the corresponding nanoprogram and stored in BlockRAM on the COMBO card. If more transformation triples in different virtual TU units use the same transformation function applied on the same packet field, then only one common nanoprogram for this transformation function will be created and stored in BlockRAM.

We may want to perform several transformations on one packet, for example, hashing source and destination IP addresses and changing TCP port numbers. In such a case, several nanoprograms will need to be performed in sequence for one packet. The physical TU unit receives two inputs for each packet.

The first input is illustrated in Fig. 3. Incoming packets are stored in DRAM. Header Field Extractor (HFE) parses packet headers step-by-step and determines offsets of interesting header fields from the beginning of the packet. The table of these offsets is added in front of each packet. SFIFO unit, which is a queue between LUP and TU, then provides addresses of beginnings of offset tables followed by packets themselves. Computing header field offsets is actually a by-product of HFE operation. Its main goal is to extract data from packet headers and store them in data structure called Uniform Header (UH) for processing by further units.

The second input is illustrated in Fig. 4. STU ID part of control word assigned to a packet by LUP is used as index to select a virtual TU unit to be used for this packet. Each STU ID value corresponds to a fixed-size piece of BlockRAM. This piece of BlockRAM includes a sequence of starting addresses of nanoprograms which should be performed for this virtual TU unit.

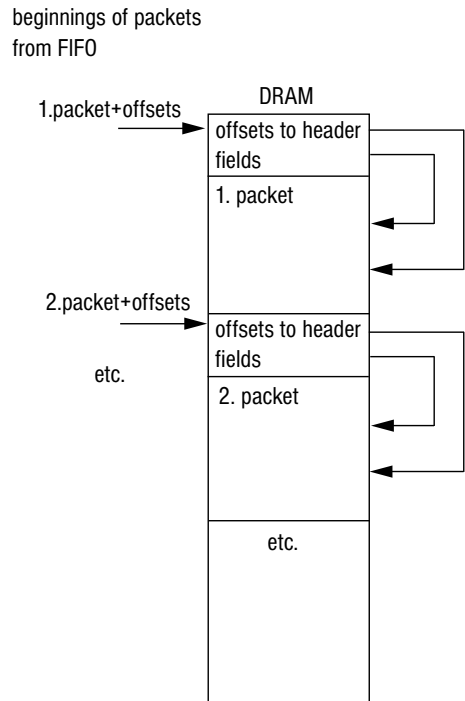


Figure 3: Header field offsets passed to TU

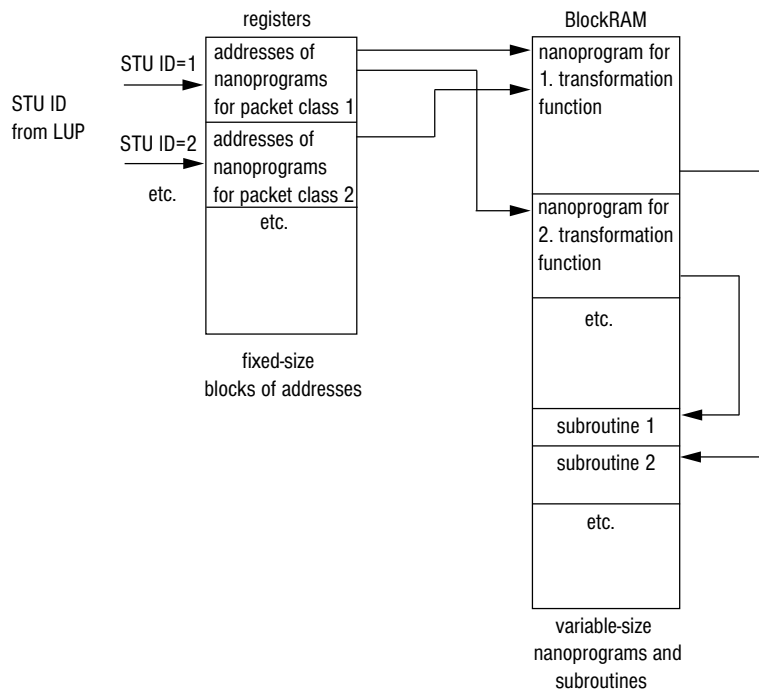


Figure 4: TU nanoprogram addresses

Note that header fields offsets can be different for each packet, whereas nanoprogram addresses can be different for different packet classes, but are the same for all packets in the same class. However, we can use the same nanoprogram for different packet classes to implement a particular transformation function on particular packet fields. The nanoprogram reads packet address from SFIFO and packet field offsets from the structure in front of the packet in DRAM.

If the same anonymization function is applied to several packet fields, for example, to source IP address and destination IP address, the nanoprogams implementing these anonymization functions can share a part of code using subroutines.

TU nanoprogams are generated and stored to BlockRAM by a software utility. This utility takes as input the sequence of transformation triples and produces as output exactly those nanoprogams, that are needed for particular anonymization functions and packet fields, trying to use subroutines to reduce the volume of code, because the space in BlockRAM is limited. The current HFE design uses two BlockRAM units each organized as 1024 words of 18 bits. Therefore there can be a maximum of 2048 nanoinstructions. The software utility then calls `nsim` to compile nanoprogams, loads them into BlockRAM on the card along with the table of pointers to the nanoprogams used by TU.

4 TU nanoprocessor

TU nanoprocessor will implement a set of nanoinstructions that will do elementary packet transformations and data manipulation. Nanoinstructions look similar to assembler instructions. They have similar mnemonics (`movc`, `jmpz`, ...) and use operands of similar types (direct constant, register name, direct memory address, memory address in register, etc.). A nanoprogram can also include jump labels.

When we want to use some nanoinstruction, we have to first specify its operating code and structure, that is the width and position of operands in the binary form of the nanoinstruction.

Nanoprogram together with the specification of used nanoinstructions is then compiled by `nsim` tool into a binary form understandable by hardware nanoprocessor.

Optionally, specification of nanoinstructions can also include executable description of their semantics in C code. `nsim` tool can then compile a simulation program, that can be used to test the nanoprogram. For example, the specification of `movr` nanoinstruction, which moves data from one register into another can look as follows:

```
#define MOVR $reg1,$reg2 : 1011 $reg1[4] $reg2[4] {\
memory[ (code1 &0xF0)>>4 ] = memory[ code1&0x0F ]; \
ip++; \
}
```

The first line specifies instruction name, list of arguments, operating code, and the order and width of operands. This is sufficient for `nsim` to compile nanoprogram for hardware unit. The remaining lines describe nanoinstruction semantics in executable C code, which can be used for simulation. `code1` includes the whole nanoinstruction including operands and `ip` is instruction pointer.

If we know the number of clocks required by hardware unit for individual nanoinstructions (most instructions require one clock), we can also use simulation with `nsim` to calculate the number of clocks required to process different packet types and thus estimate the unit throughput.

There are already two units in current liberouter firmware code, which were designed for similar purposes, that is packet modification. The first unit is EE (Editing Engine). However, this unit was used only for experiments and was never integrated into the running liberouter firmware. Its nanoinstructions perform complete simple packet modifications, such as adding a constant to a packet byte in memory. The second unit is OPE (Output Packet Editor). This unit should replace EE and is currently being designed. Its nanoinstruction set is not yet defined. The OPE unit should be used to modify packets leaving router ports, such as decrementing TTL field. The OPE unit will not be used for packets passing to the host computer for software processing. In our monitoring firmware, we will need to apply TU unit to the packets going to the host computer.

Some of our anonymization functions will be rather complex, such as prefix-preserving IP address mapping. Therefore we decided to use a low level of abstraction of TU nanoinstruction, which will allow us flexible programming of required transformations. We will base TU nanoinstruction set on HFE nanoinstructions, which use a similar level of abstraction.

The first cut of TU nanoinstructions is based on HFE nanoinstructions [2] and is shown in Table 1. We can divide nanoinstructions into several groups:

- Data copying - between registers, from and to DRAM, loading a direct constant
- Arithmetic and logical operations - add, sub, and, or, xor, not, shift (shl, shr)
- Bit operations - set, reset, invert and test bit value
- Control instructions - jump, subroutine call, etc.
- Special instructions - data output (sending data to DISP unit)

In addition to providing TU, we will also need to modify HFE nanoprogram to provide pointers to all header fields that will be anonymized. We will need to operate on much more header fields than what is sufficient for a router, for which the offsets are provided by the current version of HFE nanoprogram. See section 6.

In the latter phase, we could dynamically create an HFE nanoprogram, which will provide offsets just to those header fields, which are currently referred to by some anonymization function applied to some packet class. In this way we could reduce the time needed to parse packet headers and thus increase throughput.

5 Prefix-preserving IP address mapping

One of the requirements on prefix-preserving IP address mapping was that it should also be cryptographic - it should provide a high level of secrecy of original IP addresses. We can generate prefix-preserving mapping of original IP addresses by applying a cryptographic algorithm to them bit-by-bit from the left to the right. The two original addresses with the same prefix will be mapped into encrypted IP addresses with the same prefix. Doing 32 (for IPv4) or 128 (for IPv6) encryptions in series is a time consuming task and it would be difficult to implement it fast enough for packets coming at 1 Gb/s or more.

A possible solution is to precompute encrypted IP addresses for different plain-text IP addresses. There can be up to 2^{32} (for IPv4) or 2^{128} (for IPv6) different possible IP addresses. In reality some bit combinations do not form valid IP addresses and especially IPv6 address space

Mnemonic, ops	Description
MOVC const16	Move 16 bit constant into acc
MOVC4 n,const4	Move 4 bit constant into acc on nibble n
MOVC8 b,const8	Move 8 bit constant into acc on on byte b
MOVR dst,src	Move src register to dst register
IMOVI reg	Move reg to addr pointed by INDD and then increase INDD
IMOVD reg	Move reg to addr pointed by INDD and then decrease INDD
CMPC4 n,const4	Compare nibble n of acc with 4 bit constant const4, set ZERO flag
CMPC8 b,const8	Compare byte b of acc with 8 bit constant const8, set ZERO flag
OUT addr,reg	Data output - copy reg to DOUT or DRAMDOUT(***), addr to DOUT_ADDR register and set the "output data valid" signal
OUTI reg	Address increment & data output Increment value stored in DADDR and output data (see OUT instr.)(***)
SWP reg	Swap high and low byte of reg
ADD reg	Addition; acc + reg => acc;
SUB reg	Subtraction; acc - reg => acc;
INC reg	Increment reg
DEC reg	Decrement reg
SHL reg	Logical shift reg left
SHR reg	Logical shift reg right
AND reg	Logical AND; acc AND reg => acc
OR reg	Logical OR; acc OR reg => acc
XOR reg	Logical XOR; acc XOR reg => acc
NOT reg	Logical NOT (1's complement); NOT reg => reg
CLR reg	Clear register
CMP reg	Compare acc and reg; set ZERO flag
JMP iadr	Jump to iadr
JMPZ iadr	Jump to iadr if ZERO flag set
JMPNZ iadr	Jump to iadr if ZERO flag not set
REPI const8	Repeat next instruction const8-times
REPILC	Repeat next instr. loop_cntr-times
BTST b,r	Test bit b of register r, set ZERO flag
BST b,r	Set bit b of register r
BCL b,r	Clear bit b of register r
BINV	Invert bit b of register r
NOP	No operation

Table 1: Initial TU nanoinstruction set

is used very sparcely now. But the number of valid IP addresses that can be expected is still at the order or 10^9 , which is too much to be stored in memory.

Therefore we decided to split IP address width in two parts. Encrypted mappings of prefixes that form the first part will be precomputed and stored in memory. Encryption of postfixes that form the second part will be done in real time in our hardware. As there is a high probability of repeated occurrence of same IP addresses in a short time - in packets that form a flow - there can be a small chache of recently encrypted IP addresses for fast access without doing encryption again. The configuration for IPv4 is illustrated in Fig. 5. The numbers indicate steps to be taken during IP address anonymization:

1. A full plain-text IP address is compared by CAM to find out if its encrypted mapping is in the cache.
2. If there is a match in CAM, the index of matched line is used as address into the cache
3. The encrypted IP address is fetched from the cache
4. If there was no match in CAM, the first part of the original plain-text IP address is used as address into memory which stores precomputed prefixes
5. A precomputed prefix and the second part of the original plain-text IP address are supplied as input to encryption algorithm.
6. The encrypted IP address is obtained and stored in cache along with the original plain-text IP address in CAM

There are several points to note about this configuration:

- With COMBO card, the use of CAM as index to the cash of recently used encrypted IP addresses will require to share one physical CAM available on the card with LUP (Lookup Processor), which uses CAM for header filtering. The possibility of this sharing will have to be investigated. If it turns out to be too difficult, we will skip the cache and use only a memory of precomputed encrypted prefixes.
- We have chosen AES (Advanced Encryption Standard) because it is less compute-intensive than some other encryption schemes.

6 Supported anonymizing functions and header fields

We envisage to implement in hardware the following anonymization functions applied to the following packet fields. All other combinations must be implemented in software.

MAP All IP, TCP, UDP and ICMP header fields except OPTIONS and TCP_OPTIONS.

One optional argument as 8, 16 or 32 bit number to which the field is to be mapped. If it is not specified some compiler-time constant for each field length will be used (other than zero).

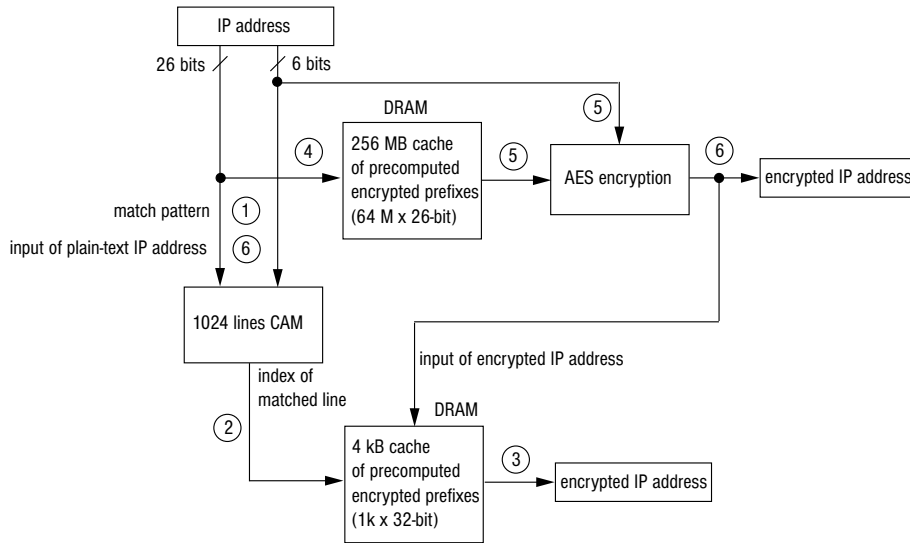


Figure 5: Implementation of prefix-preserving IP address mapping

MAP_DISTRIBUTION All IP, TCP, UDP and ICMP header fields except OPTIONS and TCP_OPTIONS.

Arguments can specify <UNIFORM, min, max> or <GAUSSIAN, median, standard deviation>. For UNIFORM a counter starting from min, going to max and wrapping back to min will be used in implementation. For GAUSSIAN a precomputed array of values will be used. The array will be indexed by a round-robin counter similar to that used for UNIFORM. There will be some implementation limitation on the range of possible values.

STRIP It *could* be applied to IP, TCP, UDP or ICMP PAYLOAD and to OPTIONS and TCP_OPTIONS (the last two would imply also stripping of IP or TCP payload) provided that we will be able to configure the length of prefix of each packet to be transferred from the monitoring card to the host PC.

Currently this configuration requires reloading a device driver for COMBO card or using an external utility for DAG card:

```
modprobe combo6-sc6ph1.o packet_size=86    (includes 32-byte hardware
                                             header and Ethernet header)

dagload snaplen=54                          (includes Ethernet header)
```

Striping TCP PAYLOAD, UDP PAYLOAD and TCP_OPTIONS works properly only when the IP header does not include options. If it includes options, they will be passed to the host in place of the TCP header.

RANDOM Only IP, TCP, UDP and ICMP header fields except OPTIONS and TCP_OPTIONS (cannot be applied to PAYLOAD).

We will need 8, 16 and 32 bit random generators implemented in hardware.

HASHED Only IP, TCP, UDP and ICMP header fields except OPTIONS and TCP_OPTIONS (cannot be applied to PAYLOAD).

Only CRC32 function can be used and it implies PAD_WITH_ZERO parameter.

PATTERN_FILL Only IP, TCP, UDP and ICMP header fields except OPTIONS and TCP_OPTIONS (cannot be applied to PAYLOAD).

The length of specified pattern must be less or equal to the length of the field to which it applies. This function will be implemented by MAP function to which specified pattern repeated to the length of the field will be supplied as argument.

ZERO Only IP, TCP, UDP and ICMP header fields except OPTIONS and TCP_OPTIONS (cannot be applied to PAYLOAD).

This function will be implemented by MAP function to which zero will be supplied.

PREFIX_PRESERVING SRC_IP and DST_IP.

This is the most difficult anonymization function to implement in hardware, see section 8.

CHECKSUM_ADJUST Possibly IP, TCP, UDP and ICMP checksum, recomputing possibilities in hardware will have to be investigated.

7 Programming interface

A programming interface for the use of TU by upper layer software will be provided in the form of a function library.

The library will translate tuples <8-bit TU ID, protocol, header field, anonymization function, parameters> to the configuration of corresponding virtual TUs as tuples <field offset in packet, field length, what to do with this field>. In the second step, the library will also produce, if necessary, the TU nanoprogram and the HFE nanoprogram (requested anonymizing functions and packet used fields may allow to use a fixed prewritten TU and HFE nanoprograms).

MAPI decides whether to use hardware or software implementation of each function applied to each flow. Hardware implementations can be used only when the functions are applied to a flow in the same sequence as is the order of units in the SCAMPI firmware. That is, the following sequence of MAPI functions can be fully implemented in hardware:

```
BPF_FILTER, PKT_COUNTER, SAMPLE, [ STR_SEARCH, ] ANONIMIZE
```

The programming interface to anonymization will use the following functions. Only the first function (`scampi_compile_transformation()`) is unique for anonymization. Other functions (`scampi_reset_classification()` and `scampi_set_filters()`) are part of the libfilter library for configuring SCAMPI firmware units, which is now being developed.

```
int scampi_compile_transformation(int filter_id, int protocol, int field,
int function, ...);
```

This function will do the tuple transformation described above for one transformation request. After all transformation requests as well as all filtering, sampling and payload searching requests (as provided by libfilter library) are compiled, the complete generated data structure are loaded into the card by `scampi_set_filters()` function.

- **filter_id** This argument works differently depending on libfilter library mode. In DIRECT mode, transformation request will be written directly do TU, whose number is specified by this argument, which can be in the range $\langle 0, 255 \rangle$. In COMPLEX mode, this argument is set to the value obtained from `scampi_compile_filter()` function.
- **protocol** This argument can be set to IP, TCP, UDP or ICMP depending on the protocol whose field is to be transformed.
- **field** This argument specifies the protocol field to be modified. There are many possible protocol fields, see Appendix A in [1].
- **function** This argument specifies the transformation function to be applied to the given protocol field. See section 6 for possible functions names.

Function returns 0 if the transformation request was successfully compiled and -1 if error occurred.

```
int scampi_reset_classification(int mode);
```

This is a related function from the libfilter library that sets mode of dealing with multiple requests for hardware monitoring functions, see argument `filter_id` of `scampi_compile_transformation` above. In DIRECT mode, multiple requests are not correlated one to another in any way, they are just compiled and downloaded to specified CAM lines and other firmware units. In COMPLEX mode, each new request is correlated with all existing requests and all requests are divided into disjunct primitives. The card can then correctly process and pass packets belonging to more than one class - typically to different applications.

```
int scampi_set_filters();
```

This is a related function from the libfilter library that loads all compiled monitoring functions in the COMBO hardware card.

8 Requirements on IP address anonymization

The most important anonymization function will be prefix-preserving anonymization of source and destination IP addresses. For the implementation of the prefix-preserving anonymization by TU we plan to use precompiled translating binary trees introduced in [5]. In this section we summarize pragmatic requirements, that ideally should be all provided by hardware implementation. We also discuss certain options and point out certain properties of IP address anonymization to keep in mind.

- Mapping of real IP addresses into their anonymized counterparts should be one-to-one, that is the same real IP address in two places in the trace should map into the same anonymized IP address.
- Mapping should be consistent among traces. That is the same real IP address in two traces should map into the same anonymized IP address. This consistency can be provided by applying the same cryptographic key as input to the algorithm when processing different traces.

- Mapping should be prefix preserving. If several real IP addresses have the same prefix of some length, they should be mapped into anonymized IP addresses which will also have the same prefix (to each other, but of course different from the real prefix). This property holds for prefixes of any lengths.
- Packet checksum should be considered as sensitive, because if we know checksum and all fields other than IP addresses we can find IP addresses by brute force.
- By using anonymized IP addresses instead of real IP addresses we lose information about real geographic places of communicating parties. By using prefix-preserving IP address anonymization, we keep information about flows of data (volume and time) between different subnets of different sizes. We can approximately compute how much traffic was local and how much traffic was travelling between remote places, based on anonymized IP address prefixes. But if two remote places share subnets, which have the same prefix of some length, we cannot find that they are remote.
- IP addresses of popular public servers (www servers, etc.) can be inferred from their high occurrence in the anonymized trace. IP addresses of DNS servers can be inferred from the hierarchy between them.
- IP addresses that are much more frequently used as source addresses rather than destination addresses are probably public servers, particularly if destination addresses are highly varying. This is because servers are more used for download than for upload.
- We should probably preserve all-ones broadcast postfixes and all-zeros network postfixes.
- We may need (requested by parameter) that real multicast addresses are mapped to anonymized addresses in some special range to keep the information that they are multicast packets.
- We may need (requested by parameter) to preserve IP address class (real IP address of class A maps to anonymized IP address of class A, etc.).

References

- [1] *D1.1a: Anonymization Framework Definition*, LOBSTER project deliverable, June 2005.
- [2] Ivo Hazmuk. *HFE Instruction Set Summary*, January 2005, `liberouter/vhdl_design/units/hfe/doc/Instruction-set-summary.txt` in `cv.s.liberouter.org`.
- [3] *Email correspondation with Panos Trimintzios from FORTH*.
- [4] *Nsim and nanoprogramming*, <http://www.liberouter.org/nsim>.
- [5] R.Ramaswamy, N.Weng, T.Wolf. *An IXA-based network measurement node*, Proc. of Intel IXA University Summit, Hudson, MA, Sep. 2004.