Prefix- and Lexicographical-order-preserving IP Address Anonymization

Matúš Harvan, Jürgen Schönwälder
International University Bremen
Campus Ring 1
28759 Bremen, Germany

***Abstract*— The anonymization of SNMP traffic traces requires an IP address anonymization scheme which is prefix-preserving and lexicographical-order-preserving. We present an anonymization scheme satisfying these two requirements which has been derived from the prefix-preserving cryptography-based scheme Crypto-PAn. We formally prove the correctness of the scheme and we describe an embeddable implementation. Limits of the proposed anonymization scheme and some security aspects are discussed as well.**

I. Introduction

The Simple Network Management Protocol (SNMP) [1] is widely deployed for network monitoring, event notification, and to some lesser extent for device configuration and control. In order to utilize more standard software components and to simplify the integration of management interfaces, there is currently a trend to introduce new XML-based protocols for network management [2].

While designing new network management protocols or improvements of the SNMP standards, it is important to understand the impact on the network in terms of bandwidth and latency as well as the impact on devices in terms of CPU and memory requirements. Comparative studies such as [3] will greatly benefit from accurate models how SNMP is used in real-world production networks. In order to develop such models, it is required to obtain and analyze SNMP traffic traces from several different production networks.

By its design, SNMP typically requires multiple protocol operations to achieve a single logical operation (e.g., retrieving a projection of a table). To reconstruct logical operations, it is necessary to analyze sequences of SNMP interactions between two SNMP engines. Furthermore, it is necessary to analyze the payload contained in SNMP messages and not just the message headers since the headers alone are not sufficient to determine where logical operations (e.g., a table retrieval) start and end.

Since the payload of SNMP messages contains large amounts of sensitive information, it is highly desirable to anonymize traces before they are given to researchers for further analysis. Existing standard techniques to anonymize network and transport layer packet headers cannot be applied directly since it is required to anonymize the payload of SNMP messages, including the variable names and their values.

The SNMP protocol operations operate on a lexicographically ordered list of variables. To be able to infer from a traffic trace where a logical operation starts and where it ends, it is necessary to look at the lexicographic order of the variables in the payload because the variable names of tabular objects are constructed from the values of so called index columns. As a consequence, anonymization functions applied to traffic traces must preserve SNMP's lexicographical-ordering property. In particular, all data encoded in variable names must be anonymized in such a way that the lexicographical-ordering of the variable names does not change.

The requirement to preserve lexicographical-ordering sets a clear constraint on the anonymization functions that can be applied to data types that are used as table index components. For IP addresses, it is in addition important to preserve the prefix relationships since IP forwarding relies on a longest-prefix-match. There are several widely deployed MIB modules where IP addresses are used as part of variable names and where lexicographical-ordering must be preserved in order to understand how traffic would be forwarded in a given network (e.g., `ipCidrRouteTable` of the `IP-FORWARD-MIB` [4], `tcpConnTable` of the `TCP-MIB` [5], `ipAddrTable` of the `IP-MIB` [6]).

This paper investigates on the existence and feasibility of a prefix-preserving and lexicographical-order-preserving IP address anonymization scheme that can be applied to SNMP traces. The rest of the paper is structured as follows. Section II reviews a cryptography-based prefix-preserving address anonymization scheme which forms the basis of the work described in this paper. Section III introduces the prefix- and lexicographical-order-preserving IP address anonymization scheme and formally proves its correctness. Some security aspects are discussed in Section IV. An implementation in form of a C library called `libanon` is described in Section V while Section VI outlines how the anonymization functions are used in a tool-set to analyze SNMP traffic traces. Related work is discussed in Section VII before the paper concludes in Section VIII.

II. Prefix-preserving IP Address Anonymization

Jun Xu, Jinliang Fan and Mostafa H. Ammar have shown that prefix-preserving IP address anonymization functions always follow a canonical form [7], [8]. In this paper, we follow the notation introduced by these authors. This section formally defines prefix-preserving anonymization and is adapted from [7].

Definition 1 (Prefix-preserving anonymization)**.** *Two IP addresses $a = a_1a_2 \ldots a_n$ and $b = b_1b_2 \ldots b_n$ share a k-bit prefix ($0 \leq k \leq n$) if $a_1a_2 \ldots a_k = b_1b_2 \ldots b_k$ and $a_{k+1} \neq b_{k+1}$ when $k < n$. An anonymization function F is defined as one-to-one function from $\{0,1\}^n$ to $\{0,1\}^n$. An anonymization function F is prefix-preserving if given two IP addresses a and b that share a k-bit prefix, $F(a)$ and $F(b)$ share a k-bit prefix as well.*

Let us consider a geometric interpretation of the prefix-preserving anonymization. Please note that the full IP address space can be represented by a complete binary tree. For IPv4 addresses this tree would have height 32, while for IPv6 it would be of height 128. Each IP address is then represented by a leaf node. Furthermore, each node corresponds to a bit position (indicated by the height of the node) and a bit value (indicated by the branch direction from its parent node). Addresses present in the unanonymized traffic trace are then represented by a subtree of the complete binary tree. Let's call this subtree the *original address tree*. Let us consider an example with 4-bit addresses for simplicity. Figure 1(a) shows the original address tree (only addresses from the trace).

A prefix-preserving function then specifies a binary variable for each non-leaf node (including the root node). This variable decides if the corresponding bit gets "flipped" during anonymization or not. The anonymization function then rearranges the original address tree into an *anonymized address tree*. The anonymization function with its variables deciding the flipping is shown in Figure 1(b) and an anonymized address tree is shown in Figure 1(c). It should be clear that the described anonymization function is prefix-preserving.

Theorem 1 (Canonical Form Theorem)**.** *Let f_i be a function from $\{0,1\}^i$ to $\{0,1\}$ for $i = 1,2,\ldots,n-1$ and f_0 be a constant function. Let F be a function from $\{0,1\}^n$ to $\{0,1\}^n$ defined as follows. Given $a = a_1a_2 \ldots a_n$, let*

$$F(a) := a'_1a'_2 \ldots a'_n \tag{1}$$

where $a'_i = a_i \oplus f_{i-1}(a_1, a_2, \ldots, a_{i-1})$ and $\oplus$ is the exclusive-or operation, for $i = 1,2,\ldots,n$. Then F is a prefix-preserving anonymization function and every prefix-preserving anonymization function necessarily takes this form.

Please note that there is a natural one-to-one mapping between the canonical form of the anonymization function and its graphical representation. Each node in the anonymization tree, corresponding to a prefix $a_1a_2 \ldots a_k$, will be labeled "flip" or "no flip" when $f_k(a_1a_2 \ldots a_k) = 1$ or 0, respectively.

The proof of Theorem 1 can be found in [7]. An implementation of the anonymization scheme using the Rijndael cipher is available under the name Crypto-PAn [9].

III. Prefix- and lexicographical-order-preserving IP Address Anonymization

A prefix-preserving and lexicographical-order-preserving anonymization function clearly has to be of the canonical form described by Theorem 1. In addition, it has to take into account the lexicographical order.

Definition 2 (Lexicographical order on IP addresses)**.** *Let $a = a_1a_2 \ldots a_n$ and $b = b_1b_2 \ldots b_n$ be two IP addresses (of the same length) where a_i's and b_i's are bits. Then a lexicographical order $<^l$ is defined by*

$$a <^l b \Leftrightarrow (\exists m > 0)(\forall i < m)(a_i = b_i) \wedge (a_m < b_m) \tag{2}$$

Note that the definition above only covers the cases where both IP addresses are of the same length (e.g., comparing two IPv4 or two IPv6 addresses, but not comparing an IPv4 with an IPv6 address).

Definition 3 (Lexicographical-order-preserving anonymization)**.** *An anonymization function F is a one-to-one function from $\{0,1\}^n$ to $\{0,1\}^n$. F is lexicographical-order-preserving if given two IP addresses a and b we have*

$$a <^l b \Rightarrow F(a) <^l F(b)$$

To preserve the lexicographical order of the anonymized IP addresses, we have to look at how the address space is used by addresses in the trace.

Definition 4 ($used_i$)**.** *Let $used_i$ be a function from $\{0,1\}^i$ to $\{0,1\}$ for $i = 1,2,\ldots,n$. The function $used_i$ is defined recursively*

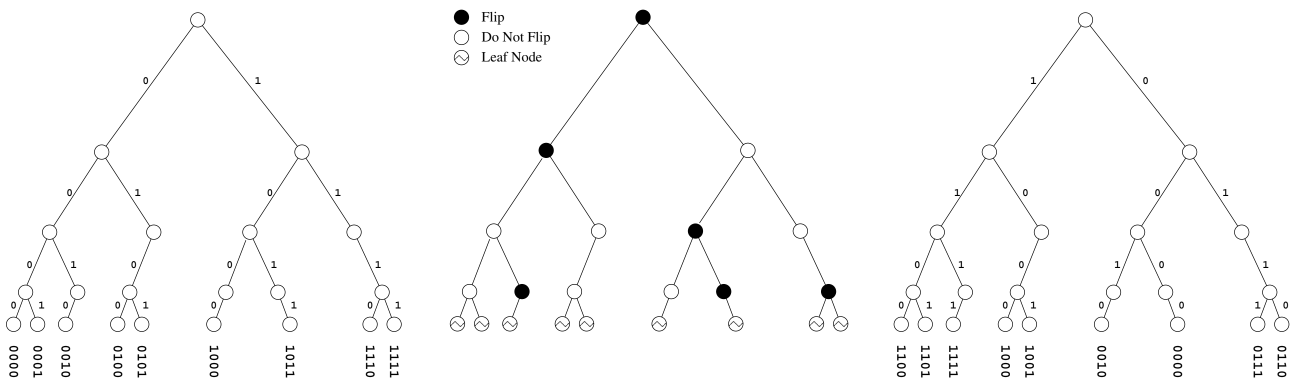
$$\begin{aligned} used_i(a_1a_2 \ldots a_i) = (\quad & used_{i+1}(a_1a_2 \ldots a_i0) \\ & \vee used_{i+1}(a_1a_2 \ldots a_i1)\,) \end{aligned} \tag{3}$$

where $used_n(a_1a_2 \ldots a_n)$ is 1 if the IP address $a_1a_2 \ldots a_n$ is in the traffic trace and 0 otherwise. The function $used_i$ determines if any IP addresses in the subtree below the a_i bit are used.

Obviously, in order to determine the values for $used_i$, we need to know all the IP addresses that should be anonymized.

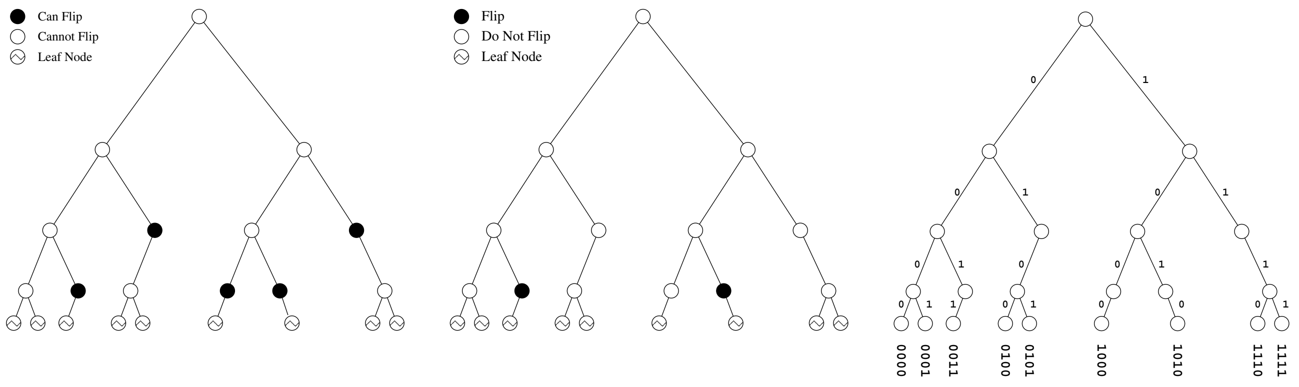
We can extend the example from Figure 1 with $used_i$. Let the addresses and original address tree be the same as in the previous example. Clearly, all nodes in the original address tree have $used_i() = 1$. Obviously, flipping a bit for which both child nodes have $used_i = 1$ (each subtree under child nodes contains at least one IP address) breaks the lexicographical order. However, if one of the child nodes has $used_i = 0$ (there is no IP address from that particular subtree present in the trace), flipping the corresponding bit does not break lexicographical order. Figure 2(a) shows which bits can be flipped by the anonymization function based on the values of $used_i$. We can now combine the previous anonymization function with $used_i$ (information on which bits can be flipped) to obtain a restricted version of the previous anonymization function — not all of the bits can be flipped any more. This restricted anonymization function is shown in Figure 2(b) and the anonymized address tree (using the restricted anonymization function) is shown in Figure 2(c).

Theorem 2 (Prefix-preserving and lexicographical-order-preserving Anonymization)**.** *Let f_i, f'_i be functions from $\{0,1\}^i$ to $\{0,1\}$ for $i = 1,2,\ldots,n-1$ and f_0,f'_0 be constant*



(a) original address tree (b) anonymization function f_i (c) anonymized address tree

Fig. 1. Geometric interpretation of the prefix-preserving anonymization function as defined in [7]. The left part (a) represents nine addresses taken from a 4-bit address space as a binary tree. The middle part (b) shows a randomly chosen anonymization function, i.e., a set of nodes in the binary tree that are flipped to obtain anonymized addresses. The right part (c) shows the resulting 4-bit addresses produced by applying the anonymization function f_i from (b).



(a) bits that can be flipped (b) anonymization function f'_i (c) anonymized address tree

Fig. 2. Geometric construction of the prefix- and lexicographical-order-preserving anonymization function. The left part (a) shows the nodes that can be flipped to obtain anonymized addresses that are lexicographic-order-preserving. The middle part (b) shows the combination with the nodes that should be flipped according to the prefix-preserving anonymization function shown in Figure 1 (b). The right part (c) shows the resulting 4-bit addresses produced by applying the anonymization function f'_i from (b).

functions. Let F be a function from $\{0,1\}^n$ to $\{0,1\}^n$ defined as follows. Given $a = a_1 a_2 \dots a_n$, let

$$F(a) := a'_1 a'_2 \dots a'_n \tag{4}$$

where

$$a'_i = a_i \oplus f'_{i-1}(a_1, a_2, \dots, a_{i-1}) \tag{5}$$

for $i = 1, 2, \dots, n$ and

$$f'_j(a_1, \dots, a_j) = f_j(a_1, \dots, a_j) \wedge \\ \neg\,(used_{j+1}(a_1, \dots, a_j, 0) \wedge used_{j+1}(a_1, \dots, a_j, 1)) \tag{6}$$

for $j = 1, \dots, n-1$. For the case $j = 0$ ($i = 1$) we have

$$f'_0 = f_0 \wedge \neg\,(used_1(0) \wedge used_1(1)) \tag{7}$$

Then we claim F is a prefix-preserving and lexicographical-order-preserving anonymization function.

f'_i is similar to f_i except that it takes into account which parts of the address space are used. As we will see later, this is sufficient for the lexicographical-order-preserving property. Please note that the lexicographical-order-preserving property holds only for IP addresses used in the trace, so all IP addresses need to be known beforehand. Trying to anonymize an IP address not in the trace when $used_i$ was generated might break the lexicographical order.

Proof. To show that F is prefix-preserving, it is sufficient to observe that $a'_i = a_i \oplus f'_{i-1}(a_1, a_2, \dots, a_{i-1})$ is of the form as required by Theorem 1 and hence F is prefix-preserving.

To show that F is lexicographical-order-preserving, let a,b be two IP addresses of length n-bits, sharing a k-bit prefix ($a_i = b_i$ for $i \le k$ and $a_{k+1} \ne b_{k+1} = \neg a_{k+1}$ if $k < n$).

For $i \le k$ we have

$$\begin{aligned} a'_i &= a_i \oplus (f_{i-1}(a_1, \dots, a_{i-1}) \wedge \\ &\quad \neg\,(used_i(a_1, \dots, a_{i-1}, 0) \wedge used_i(a_1, \dots, a_{i-1}, 1))) \\ &= b_i \oplus (f_{i-1}(b_1, \dots, b_{i-1}) \wedge \\ &\quad \neg\,(used_i(b_1, \dots, b_{i-1}, 0) \wedge used_i(b_1, \dots, b_{i-1}, 1))) \end{aligned}$$

$= b'_i$

If $k = n$ then $a = b$ and hence also $F(a) = F(b)$, so the lexicographical order is preserved. Let's consider the case where $k < n$ and hence $a \neq b$. Without loss of generality assume $a <^l b$. It follows that $a_{k+1} < b_{k+1} \Rightarrow a_{k+1} = 0, b_{k+1} = 1$. Then for $k+1$ we have (from equations 5 and 6)

$$a'_{k+1} = a_{k+1} \oplus (f_k(a_1, \ldots, a_k) \wedge \neg (used_{k+1}(a_1, \ldots, a_k, 0) \wedge used_{k+1}(a_1, \ldots, a_k, 1))) \quad (8)$$

If $f_k(a_1, \ldots, a_k) = 0$, then from equation 8 we have $a'_{k+1} = a_{k+1} \oplus 0 = a_{k+1}$ and $b_{k+1} = b'_{k+1}$. Hence $a <^l b \Rightarrow F(a) <^l F(b)$.

If $f_k(a_1, \ldots, a_k) = 1$, then we have to consider four possible cases for the values of $used_{k+1}$ in equation 8. Please note that since $a_1 \ldots a_k = b_1 \ldots b_k$ we have

$$used_{k+1}(a_1, \ldots, a_k, 0) = used_{k+1}(b_1, \ldots, b_k, 0)$$

and

$$used_{k+1}(a_1, \ldots, a_k, 1) = used_{k+1}(b_1, \ldots, b_k, 1)\,.$$

First, consider the case $used_{k+1}(a_1, \ldots, a_k, 0) = 0$ and $used_{k+1}(a_1, \ldots, a_k, 1) = 0$. The values of $used_i$ indicate that no IP address from the subtree below $a_1 \ldots a_k$ is present in the trace. Therefore, preserving lexicographical-ordering between a and b (both unused in the trace) is not necessary and the a_{k+1}, b_{k+1} bits may be flipped.

Next, consider the case $used_{k+1}(a_1, \ldots, a_k, 0) = 0$ and $used_{k+1}(a_1, \ldots, a_k, 1) = 1$. This implies that one of the IP addresses may be present in the trace (because at least one IP address from its subtree is used) while the other one for sure is not in the trace (as no IP address from its subtree is used). Therefore, preserving lexicographical-ordering between a and b is not necessary and the a_{k+1}, b_{k+1} bits may be flipped.

Next, consider the case $used_{k+1}(a_1, \ldots, a_k, 0) = 1$ and $used_{k+1}(a_1, \ldots, a_k, 1) = 0$. This is is similar to the previous case - only one of the IP addresses can be in the trace. Therefore, preserving lexicographical-ordering between a and b is not necessary and the a_{k+1}, b_{k+1} bits may be flipped.

Finally, consider the case $used_{k+1}(a_1, \ldots, a_k, 0) = 1$ and $used_{k+1}(a_1, \ldots, a_k, 1) = 1$. This implies that both IP addresses a and b may be in the trace and hence their lexicographical-ordering has to be preserved. $a'_{k+1} = a_{k+1} \oplus 1 \wedge \neg(1 \wedge 1) = a_{k+1} \oplus 0 = a_{k+1}$ and $b'_{k+1} = b_{k+1} \oplus 0 = b_{k+1}$. It follows that $a <^l b \Rightarrow a_i = b_i$ for $i \leq k$ and $a_{k+1} < b_{k+1} \Rightarrow a'_i = b'_i$ for $i \leq k$ and $a'_{k+1} < b'_{k+1} \Rightarrow F(a) <^l F(b)$. □

IV. Security Considerations

In this section, we examine how feasible it is for an attacker to recover the original addresses from an anonymized trace. Since our scheme is prefix-preserving, the limitations and security weaknesses of *Crypto-PAn* apply to it as well. In particular, the prefix-preserving property implies that if an address is compromised, so is its prefix and hence prefixes of other addresses will be revealed.

Clearly, the requirement to preserve lexicographical-ordering poses further limitations on the anonymization. The more IP addresses are used in the trace, the less bits can be flipped by the anonymization function. In the extreme case where the whole address space is used, we cannot anonymize any IP address. In case a complete subnet is used, it turns out that we only can anonymize the prefix for that subnet, but the last part of the IP address (suffix or host part) would have to remain unchanged. However, if one of the addresses is revealed, the prefix for the other addresses will be known as well. Therefore, the origin of the anonymized trace should be kept secret as its knowledge might allow an attacker to guess the prefix of addresses in the trace.

It must be noted that the 32-bit IPv4 address space is rather small and hence there are only a few bits left for anonymization in many practical applications. Since the IPv6 address space is significantly larger, a more robust anonymization of IPv6 traces can be achieved compared to IPv4 traces. Furthermore, some implementations randomize the host portion of an (auto-configured) IPv6 address and hence revealing it might be of much lower value for an attacker.

With respect to SNMP trace anonymization, we have to bear in mind that *Crypto-PAn* was evaluated with respect to traces of application traffic containing IP addresses from various networks with different prefixes. In the case of SNMP, the IP addresses to be anonymized would come from the management traffic and hence might contain different sets of addresses. For example, attack techniques based on well-known frequently accessed servers like DNS root servers or frequently visited web servers might be less feasible.

The number of bits actually flipped also depends on the choice of the anonymization key. However, choosing the key in such a way that more bits get flipped does not make the anonymization more secure. Such an approach only results in some keys being more probable to be chosen, which in turn could be exploited by an attacker to find the key much faster.

We can, however, use this idea to define a metric q indicating how many bits can be flipped. Let q be the fraction of the number of times when a bit can be flipped over the size of the address space. In the following, we consider addresses with a length of n bits. Since the $used_i$ function indicates which bits can be flipped, we can define q as follows:

$$q = \frac{\text{number of times}\neg(used_i(...0) \wedge used_i(...1))}{2^n} \quad (9)$$

Theorem 3. *The number of times a bit cannot be flipped, i.e., the number of times $\neg(used_i(...0) \wedge used_i(...1)) = 0$ (white nodes in Figure 2(a)) is the number of distinct addresses in the trace -1 (in case there is at least one IP address already in the trace).*

Proof. In the case of a trace with just one IP address, we can obviously flip any bit we want. For a trace with two

```
/*
 * IPv4 address anonymization API.
 */

typedef struct _anon_ipv4 anon_ipv4_t;

anon_ipv4_t* anon_ipv4_new();
void         anon_ipv4_set_key(anon_ipv4_t *a, const uint8_t *key);
int          anon_ipv4_set_used(anon_ipv4_t *a, in_addr_t ip, int prefixlen);
int          anon_ipv4_map_pref(anon_ipv4_t *a, const in_addr_t ip, in_addr_t *aip);
int          anon_ipv4_map_pref_lex(anon_ipv4_t *a, const in_addr_t ip, in_addr_t *aip);
void         anon_ipv4_delete(anon_ipv4_t *a);

/*
 * IPv6 address anonymization API.
 */

typedef struct _anon_ipv6 anon_ipv6_t;
typedef struct in6_addr in6_addr_t;

anon_ipv6_t* anon_ipv6_new();
void         anon_ipv6_set_key(anon_ipv6_t *a, const uint8_t *key);
int          anon_ipv6_set_used(anon_ipv6_t *a, const in6_addr_t ip, int prefixlen);
int          anon_ipv6_map_pref(anon_ipv6_t *a, const in6_addr_t ip, in6_addr_t *aip);
int          anon_ipv6_map_pref_lex(anon_ipv6_t *a, const in6_addr_t ip, in6_addr_t *aip);
void         anon_ipv6_delete(anon_ipv6_t *a);
```

Fig. 3. IP address anonymization API provided by `libanon`. First, an anonymization object (type `anon_ipv4_t` or `anon_ipv6_t`) must be created. The `anon_ipv4_map_pref()` and `anon_ipv6_map_pref()` functions implement prefix-preserving anonymization while the functions `anon_ipv4_map_pref_lex()` and `anon_ipv6_map_pref_lex()` implement prefix-preserving and lexicographical-order-preserving anonymization. Note that these last two functions require that addresses are first marked as used by calling `anon_ipv4_set_used()` or `anon_ipv6_set_used()`.

distinct addresses (sharing a k-bit prefix), only one bit cannot be flipped. This is the $k+1$-th bit.

Assume the claim holds for a trace with n addresses. Then let a be an IP address not yet in the trace. Let b be an address from the trace with n addresses such that b has the longest common prefix with a from all addresses in the trace. Let the length of this prefix be k. Clearly $used_{k+1}(a_1 \dots a_k b_{k+1}) = 1$ as b is in the trace and $used_{k+1}(a_1 \dots a_k a_{k+1}) = 0$ as a is not in the trace. $\neg(used_{k+1}(a_1 \dots a_k 0) \wedge used_{k+1}(a_1 \dots a_k 1)) = 1$ and hence the k-th bit can be flipped.

However, after adding the address a to the trace we get $used_{k+1}(a_1 \dots a_k a_{k+1}) = 1$, so $\neg(used_{k+1}(a_1 \dots a_k 0) \wedge used_{k+1}(a_1 \dots a_k 1)) = 0$ and $f'_k(a_1 \dots a_k) = 0$. Hence the k-th bit can no longer be flipped. By adding a to the trace, we get a trace with $n+1$ distinct addresses and n non-flippable bits, so the claim holds. $\square$

The theorem essentially says that every new address added to the trace makes one more bit in the tree not flippable. This allows us to calculate the q metric directly from the number of distinct IP addresses in the trace:

$$q = \frac{\text{number of distinct addresses} - 1}{\text{size of address space}}$$

The number of bits that can be flipped only depends on the number of distinct addresses in the trace rather than on the prefix-relationships between the addresses.

Finally, it should be noted that the strength of the IP address anonymization scheme may be reduced in cases where IP addresses contain other identifiers. Examples are IPv6 addresses containing embedded IEEE 802 MAC addresses, constructed according to the Modified EUI-64 format [10]. If it is possible to obtain knowledge about the MAC addresses, one can exploit that knowledge to restrict the search space on brute force attacks. In fact, the embedding of other identifiers into IP addresses effectively binds the anonymization functions applied to these identifiers and the IPv6 addresses together. The same applies also in cases where IP addresses are embedded into other identifiers. Note that this is not an inherent weakness of the IP address anonymization scheme presented here but a general problem to consider when one combines several anonymization functions to data structures that are not independent.

V. Implementation

The anonymization function described in the previous section has been implemented as a C library and a sample program that converts IP addresses to anonymized IP addresses. Both, IPv4 and IPv6 addresses are supported. The anonymization part is based on the *Crypto-PAn* implementation, which was rewritten in C and extended with the lexicographical-order-preserving property. Furthermore, the AES implementation from the OpenSSL project (*libcrypto*) is used for the cryptographic functionality.

Internally, the $used_i$ variables are stored in a tree similar to the original address tree from Figure 2(a). This approach has rather small computational complexity as adding new nodes into a binary tree and looking up nodes is very efficient. The disadvantage is the memory consumption — for an n-bit address space we need a tree with $2^{(n+1)} - 1$ nodes.

Because of the way we have designed the anonymization scheme, all IP addresses from the trace must be known prior to starting the anonymization. Two possible ways have been implemented to determine the IP addresses used in the trace. One is to scan the whole trace for addresses and create the tree on the fly. The other one is to let the user define subnets, from which addresses in the trace come. The latter has the advantage of creating consistent anonymization on traces from the same subnet but with slightly different usage of IP addresses. It also allows for parallel execution on parts of the trace on different computers. For densely used parts of the address space, this “approximation” seems to be very efficient.

In order to decrease memory consumption, marking a subnet as completely used removes all but the top node corresponding to that subnet in the address tree. However, the current implementation does not actively check for full subnets in order to prune the tree. The approach in which the address tree is built in memory works for IPv4 addresses (tested with 10^6 randomly generated IP addresses) as well as for IPv6 addresses. However, it does not scale well to IPv6 with respect to memory complexity as IPv6 addresses are longer and hence require a deeper tree with more nodes.

The current implementation is capable of anonymizing IPv4 and IPv6 addresses. In addition, transformations for IEEE 802 MAC addresses and 64-bit signed and unsigned integers have been implemented. These transformations, however, do not preserve the prefixes and use anonymization methods different from the IP address anonymization technique described in this paper. Support for additional data types is planned.

The application programming interface (API) provided by the C library `libanon` for anonymizing IPv4 addresses is shown in Figure 3. To use the library, an IP anonymization object using `anon_ipv4_new()` has to be allocated first and the key set using `anon_ipv4_set_key()`. Before addresses can be anonymized, `anon_ipv4_set_used()` has to be called for each address or subnetwork that is used in the trace file. Afterwards, calls to `anon_ipv4_map_pref_lex()` can be used to obtain anonymized addresses. The anonymization object can be destroyed by calling `anon_ipv4_delete()`. Similar API functions are provided for IPv6 addresses and other data types.

The `libanon` implementation has been tested on several traces with randomly generated IP addresses and verified to preserve lexicographical-ordering. The verification process was to order the non-anonymized trace file by IP addresses and then check if the anonymized trace file is still lexicographically ordered.

Memory consumption and performance have been measured for various numbers of IPv4 and IPv6 addresses. Memory consumption has been measured by stopping the anonymization program after the anonymizations are done but before memory is deallocated. Standard system tools were used to measure how much memory was used by the stopped program. Besides that, a counter has been added to keep track of the number of nodes in the $used_i$ tree. This allows us to calculate how much memory was requested for the tree with `malloc()` since the data structure for a tree node has a size of 16 bytes.

The results are summarized in Tables I and II We clearly see that for anonymizing IPv6 addresses, much more memory is needed than for anonymizing IPv4 addresses. We also can see that after adding the memory footprint of the program on empty input, more memory is allocated than theoretically requested by `malloc()`. This discrepancy is caused by the `malloc()` behavior. Depending on the size of the trace and possible marking of subnets as completely used, the memory consumption might be a problem. In order to decrease the memory footprint, pruning of the tree for completely used subnets might be implemented after addition of new nodes. This could help especially for densely used address spaces. In addition, a special purpose memory allocator will likely reduce the `malloc()` overhead.

TABLE I
MEMORY FOOTPRINT FOR IPV4 ANONYMIZATION.

number of IP addresses	number of nodes	measured memory footprint	theoretical memory requests
0	1	3 212 KB	16 B
1	33	3 220 KB	32 B
10	301	3 220 KB	4 KB
100	2 646	3 220 KB	41 KB
1 000	23 182	3 744 KB	362 KB
10 000	199 080	7 836 KB	3 110 KB
100 000	1 656 713	42 024 KB	25 886 KB

TABLE II
MEMORY FOOTPRINT FOR IPV6 ANONYMIZATION.

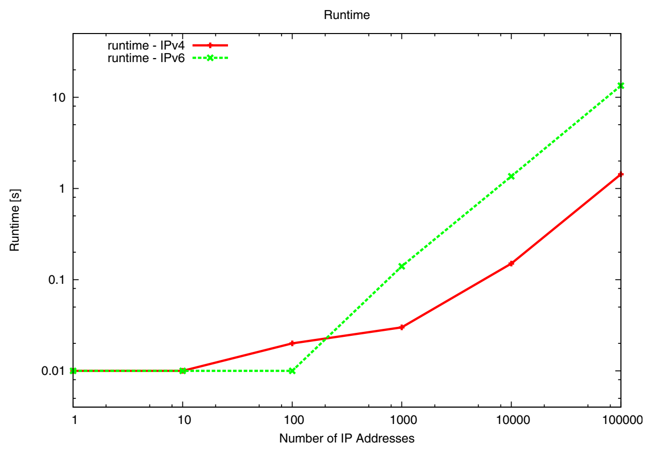
number of IP addresses	number of nodes	measured memory footprint	theoretical memory requests
0	1	3 212 KB	16 B
1	129	3 216 KB	2 KB
10	1 248	3 216 KB	19 KB
100	12 143	3 480 KB	189 KB
1 000	118 189	5 860 KB	1 846 KB
10 000	1 147 052	30 012 KB	17 922 KB
100 000	11 080 902	262 860 KB	173 139 KB

Runtime was measured using bash’s builtin `time` function. The benchmark measurements were done on a Xeon 3GHz machine with 1GB of RAM. The results are summarized in Table III. For the runtime measurements, swap was not used. However, when we tried to anonymize 10^6 IPv6 addresses, swapping was needed and caused the runtime to be incomparably longer. As expected, the runtime for anonymizing IPv6 addresses is longer than anonymizing an equal number of IPv4 addresses.

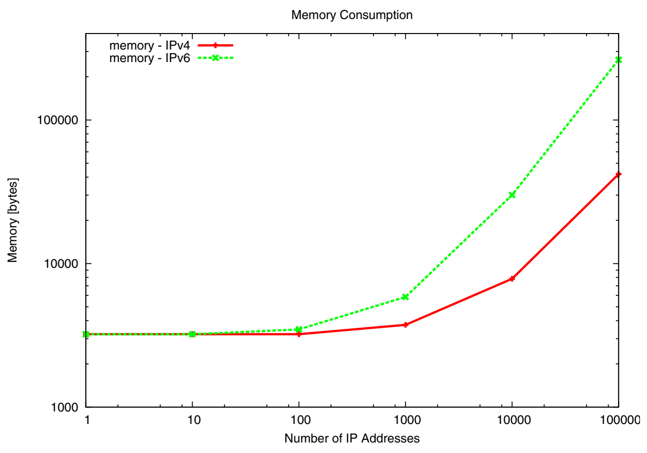
TABLE III
RUNTIME OF IPV4 AND IPV6 ANONYMIZATION.

number of IP addresses	runtime IPv4	runtime IPv6
1	0.01 s	0.01 s
10	0.01 s	0.01 s
100	0.02 s	0.01 s
1 000	0.03 s	0.14 s
10 000	0.15 s	1.36 s
100 000	1.43 s	13.4 s

Figure 4 visualizes our runtime and memory usage measurements. It can be seen that the runtime is correlated to the memory allocation overhead, which dominates the performance of our current implementation.



(a) IP address anonymization runtime



(b) IP address anonymization memory consumption

Fig. 4. Measured runtime and memory footprint for IPv4 and IPv6 address anonymization. The runtime is generally acceptable for offline analysis as long as the binary tree data structure fits into main memory. Memory consumption increases significantly faster for IPv6 addresses.

VI. Application

Two tools have been implemented in order to analyze SNMP traffic traces. The first tool `snmpdump` allows to convert SNMP messages contained in raw `pcap` traces [11] into an XML [12] representation. Fragmented SNMP messages are reassembled using the `libnids` library. The XML representation serves as an intermediate exchange format and as the binding interface between operators who contribute traffic traces and researchers involved in the analysis of these traffic traces. The details of the schema definition can be found in [13].

The second tool `snmpanon` can be used to filter and anonymize traffic traces. The `snmpanon` tool reads SNMP messages from an XML file (or standard input), anonymizes and filters data and finally writes out a new XML document with sensitive information removed or anonymized. Highly sensitive elements (e.g., SNMP community strings) can be deleted or replaced with empty content by passing appropriate filter expressions to `snmpanon`. These filter expressions "filter-out" information before anonymization starts to ensure that filtered information is not implicitly kept in anonymized traces by influencing how anonymization transformations work.

The remaining sensitive elements can be anonymized by applying the `libanon` transformations. A configuration file defines how many different transformation instances are created by `snmpanon` and the set of variables transformed by a given transformation instance. The configuration file uses regular expressions to bind sets of type and object type names to a transformation instance. This approach allows us to deal with object types which use various slightly different SMI types to represent the same information.

IP addresses were originally represented in SNMP MIB modules using the `IpAddress` SMI data type [14]. The `InetAddressType` and `InetAddress` textual conventions [15] were introduced in 2000 to handle multiple network layer addressing formats. The `InetAddress` serves as a discriminated union for different network layer address formats which includes besides IPv4 and IPv6 addresses also domain names. The `InetAddressType` serves as the discriminator of the union.

Consistent handling of IPv4 addresses requires some elaborate code to identify the discriminator object for a given `InetAddress` object and the search may not always succeed. In such cases, our implementation reverts back to heuristics to decide whether a given `InetAddress` value should be anonymized or not. A trace of all heuristic decisions is written to the output so that it is possible to verify whether the heuristics did do the right thing for a given input.

VII. Related Work

Several projects on IP address anonymization and in particular on prefix-preserving IP address anonymization exist. One of the first publicly available tools to do prefix-preserving IP address anonymization was *tcpdpriv* [16] (using the `-A50` option). Unfortunately the anonymization used is susceptible to an attack described in [17]. Of particular interest is a more secure tool *Crypto-PAn* [9], implementing a cryptography-based scheme described in [7]. This anonymization scheme is not suffering from tcpdpriv's weaknesses.

Ruoming Pang and Vern Paxson proposed a high-level programming environment for packet trace anonymization [18]. The basic idea is to use simple transformation scripts which are applied to rewrite portions of a packet trace. The approach has been applied to anonymize FTP traces. The "filter-in" principle is applied to produce only output for data that has been recognized by a transformation script.

David A. Maltz et.al. consider the problem of anonymizing router configuration data [19]. Their approach is to generally hash strings that are not keywords while transforming some important data types (IP addresses, AS numbers, BGP community attributes) specially to preserve their meaning.

Mario Baldi and Fulvio Risso recently proposed an XML-based language called Packet Details Markup Language (PDML) [20] which expresses information related to decoded packets (namely protocol names, field names and their values). The PDML format is relatively close to the XML format used by the tools described in Section VI. The main difference is that PDML aims to be a generic protocol format while our XML format is by design SNMP specific and thus a bit easier to understand and use since element names carry semantics while PDML element names are generic and additional attributes are used to provide the necessary context. Furthermore, PDML includes features for improved rendering of information, which is clearly not needed for our purposes.

VIII. Conclusion

To improve our understanding of the actual usage of network management protocols, it is necessary to obtain management traffic traces from operational environments. Traffic anonymization is one approach to protect sensitive information contained in such traffic traces. The anonymization of IP addresses in particular is challenging since IP address prefixes are relevant for the forwarding semantics. The analysis of SNMP traces in addition requires that IP addresses appearing in variable names are lexicographically ordered.

A prefix-preserving and lexicographical-order-preserving IP address anonymization scheme has been introduced by extending the prefix-preserving cryptography-based scheme from *Crypto-PAn* [7] to preserve lexicographical order. The scheme has been rigorously proven to be correct. Its limits as well as security aspects are discussed.

The algorithm has been implemented in the form of a C library called `libanon` which is part of the `snmpdump` package for analyzing SNMP traffic traces. In its current implementation the library has a rather large memory footprint. As future work one could implement pruning of the internally used tree for completely used subnets or use some more advanced algorithms, like path compression, to decrease the memory requirements.

The tool `snmpanon` uses the `libanon` anonymization library to anonymize SNMP traces. Future work will focus on the collection of SNMP traces from various operational networks and their subsequent analysis using the tools and algorithms described in this paper.

References

[1] J. Case, R. Mundy, D. Partain, and B. Stewart, "Introduction and Applicability Statements for Internet Standard Management Framework," SNMP Research, Network Associates Laboratories, Ericsson, RFC 3410, Dec. 2002.

[2] J. Schönwälder, A. Pras, and J. P. Martin-Flatin, "On the Future of Internet Management Technologies," *IEEE Communications Magazine*, vol. 41, no. 10, pp. 90–97, Oct. 2003.

[3] A. Pras, T. Drevers, R. van de Meent, and D. Quartel, "Comparing the Performance of SNMP and Web Services based Management," *IEEE electronic Transactions on Network and Service Management*, vol. 1, no. 2, Nov. 2004.

[4] F. Baker, "IP Forwarding Table MIB," Cisco Systems, RFC 2096, Jan. 1997.

[5] R. Raghunarayan, "Management Information Base for the Transmission Control Protocol (TCP)," Cisco Systems, RFC 4022, Mar. 2005.

[6] K. McCloghrie, "SNMPv2 Management Information Base for the Internet Protocol using SMIv2," Cisco Systems, RFC 2011, Nov. 1996.

[7] J. Xu, J. Fan, and M. H. Ammar, "Prefix-Preserving IP Address Anonymization: Measurement-based Security Evaluation and a New Cryptography-based Scheme," in *Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP'02)*, Nov. 2002.

[8] J. Fan, J. Xu, M. Ammar, and S. Moon, "Prefix-Preserving IP Address Anonymization," *Computer Networks*, vol. 46, no. 2, pp. 253–272, Oct. 2004.

[9] J. Xu, J. Fan, M. H. Ammar, and S. Moon, "Crypto-PAn," 2003, http://www.cc.gatech.edu/computing/Telecomm/cryptopan/.

[10] R. Hinden and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture," Nokia, Cisco Systems, RFC 3513, Apr. 2003.

[11] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proc. Usenix Winter Conference*, Jan. 1993.

[12] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0," Textuality and Netscape, Microsoft, University of Illinois," W3C Recommendation, Feb. 1998.

[13] J. Schönwälder, "SNMP Traffic Measurements," International University Bremen, Internet Draft <draft-schoenw-nrmg-snmp-measure-00.txt>, Dec. 2005.

[14] K. McCloghrie, D. Perkins, and J. Schönwälder, "Structure of Management Information Version 2 (SMIv2)," Cisco Systems, SNMPinfo, TU Braunschweig, RFC 2578, Apr. 1999.

[15] M. Daniele, B. Haberman, S. Routhier, and J. Schönwälder, "Textual Conventions for Internet Network Addresses," SyAM Software, Johns Hopkins University, Wind River Systems, International University Bremen, RFC 4001, Feb. 2005.

[16] G. Minshall, "tcpdpriv," 1996, http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html.

[17] T. Ylonen, "Thoughts on how to mount an attack on tcpdpriv's "-A50" option..." http://ita.ee.lbl.gov/html/contrib/attack50/attack50.html.

[18] R. Pang and V. Paxson, "A High-level Programming Environment for Packet Trace Anonymization and Transformation," in *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2003.

[19] D. A. Maltz, J. Zhan, G. Xie, and H. Zhang, "Structure Preserving Anonymization of Router Configuration Data," in *Proceedings of the 4th Internet Measurement Conference*, Oct. 2004.

[20] M. Baldi and F. Risso, "Using XML for Efficient and Modular Packet Processing," in *Proceedings of IEEE Globecom 2005*, Dec. 2005.