Guided Research Final Report
Prefix- and Lexicographical-order-preserving IP Address Anonymization

Matúš Harvan
m.harvan@iu-bremen.de

Spring Semester 2005

Supervisor: Jürgen Schönwälder

Contents

1 Executive summary 2

2 Summary 3

3 Introduction 4

4 Prefix-preserving IP address anonymization 5

5 Prefix- and lexicographical-order-preserving IP address anonymization 8

6 Security Section 12

7 Implementation 13

8 Conclusion 15

1 Executive summary

The aim of this project is to investigate on the existence and feasibility of a prefix-preserving and lexicographical-order-preserving IP address anonymization. Such an anonymization has been found and proven in a rigorous way to work correctly. Furthermore, the anonymization scheme has been implemented in the form of a C library and a sample tool.

The found IP address anonymization scheme can be used for various purposes. However, the main reason for this project is the need to anonymize SNMP traffic traces. Simple Network Management Protocol (SNMP) is a protocol to access management and control information of network devices. It is very lightweight and capable of easily monitoring thousands of devices simultaneously. Therefore, it is used extensively in enterprise networks and by ISPs. However, it is not understood how exactly SNMP performs in practice, what are the various interactions, where exactly the real problems are and optimizations are done for assumed bottlenecks as there is no data available to the research community from operators of large networks. The operators are concerned about the privacy of their networks' users and afraid of providing potential attackers with sensitive information about their network allowing for easier break-ins. The found anonymization scheme is the first step to a complete anonymization of SNMP traffic traces and would in the end allow access to traces of real-world traffic to the research community.

2 Summary

The aim of this project is to investigate on the existence and feasibility of a prefix-preserving and lexicographical-order-preserving IP address anonymization. Such an anonymization has been found and proven in a rigorous way to work correctly. Limits on its usage as well as security aspects are discussed. Furthermore, a C library implementing the found anonymization scheme has been developed.

The main reason for investigating on the prefix- lexicographical-order-preserving anonymization scheme is the need to anonymize SNMP traffic traces. Simple Network Management Protocol (SNMP) is a protocol to access management and control information of network devices. It is a lightweight, stateless protocol used extensively in enterprise networks and by ISPs. Large amounts of new MIB modules are being produced (on customer requests) by companies as Enterasys, Juniper or Cisco[5], showing the popularity and wide deployment of SNMP. However, due to the lack of available traces from operational Internet networks it is not known how exactly SNMP is used in practice, which particular management applications are preferred and how efficiently they make use of available protocol options. The lack of traces is caused by network operators' concerns for the privacy and security of their networks. The found anonymization scheme is the first step to a complete anonymization of SNMP traffic traces and would in the end allow access to traces of real-world traffic to the research community. This would help clarify how exactly SNMP is used, allow to study interaction patterns of different SNMP implementations, compare performance and evaluate different SNMP approaches.

3 Introduction

The aim of this project is to investigate on the existence and feasibility of a prefix-preserving and lexicographical-order-preserving IP address anonymization. A suitable anonymization scheme has been found, rigorously proven to be correct and implemented in the form of a C library. Limits of the proposed anonymization and security aspects are discussed as well.

The main reason to investigate on such an anonymization scheme is the need to anonymize SNMP traces. This is of particular importance as no traces of SNMP traffic from real-world networks are available, rendering analysis of SNMP in real networks almost impossible. Therefore, it is not understood how exactly SNMP is used in practice and how protocol options are being used by management applications. This leads to optimizations done for assumed interactions while not being sure if and where exactly optimizations would be needed. The main reason for the absence of SNMP traces is the concern of network operators about disclosing sensitive information by providing the traces for networking research. However, having a suitable anonymization scheme would be likely to relieve the current reluctance to provide an insight into their networks and give researchers access to anonymized versions of the traces. The main challenge involved in anonymizing SNMP traces is to find a lexicographic-order- and prefix-preserving IP address transformation. It has to be a prefix-preserving transformation so that the anonymized trace would still be usable if prefix relationships were important. The lexicographic-order-preserving requirement comes from the way how SNMP works. Larger objects like tables (potentially indexed by IP addresses) are stored in lexicographic order and are sequentially retrieved via several smaller queries. In order for these SNMP interactions to be recognizable in anonymized traces, the IP anonymization scheme has the additional requirement of being lexicographic-order-preserving.

4 Prefix-preserving IP address anonymization

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* [3] (using the `-A50` option). Unfortunately the anonymization used is susceptible to an attack described in [8]. Of particular interest is a more secure tool *Crypto-PAn* [7], implementing a cryptography-based scheme described in [6]. This anonymization scheme is not suffering from tcpdpriv's weaknesses. The cryptography-based anonymization paper [6] formally characterizes prefix-preserving functions and shows that all such functions follow a canonical form. Although none of the mentioned projects treats preserving of lexicographical ordering, the latter project provides important results used later in our work and therefore will be explained in more detail. The rest of this section is adapted from [6].

First we have to formally define what we mean by prefix-preserving anonymization.

Definition 1 (Prefix-preserving Anonymization). *(adapted from [6]) Two IP addresses $a = a_1a_2 \dots a_n$ and $b = b_1b_2 \dots b_n$ share a k-bit prefix ($0 \leq k \leq n$) if $a_1a_2 \dots a_k = b_1b_2 \dots b_k$ 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 a complete binary tree while Figure 1(b) 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(c) and an anonymized address tree is shown in 1(d). It should be clear that the described anonymization function is prefix-preserving.

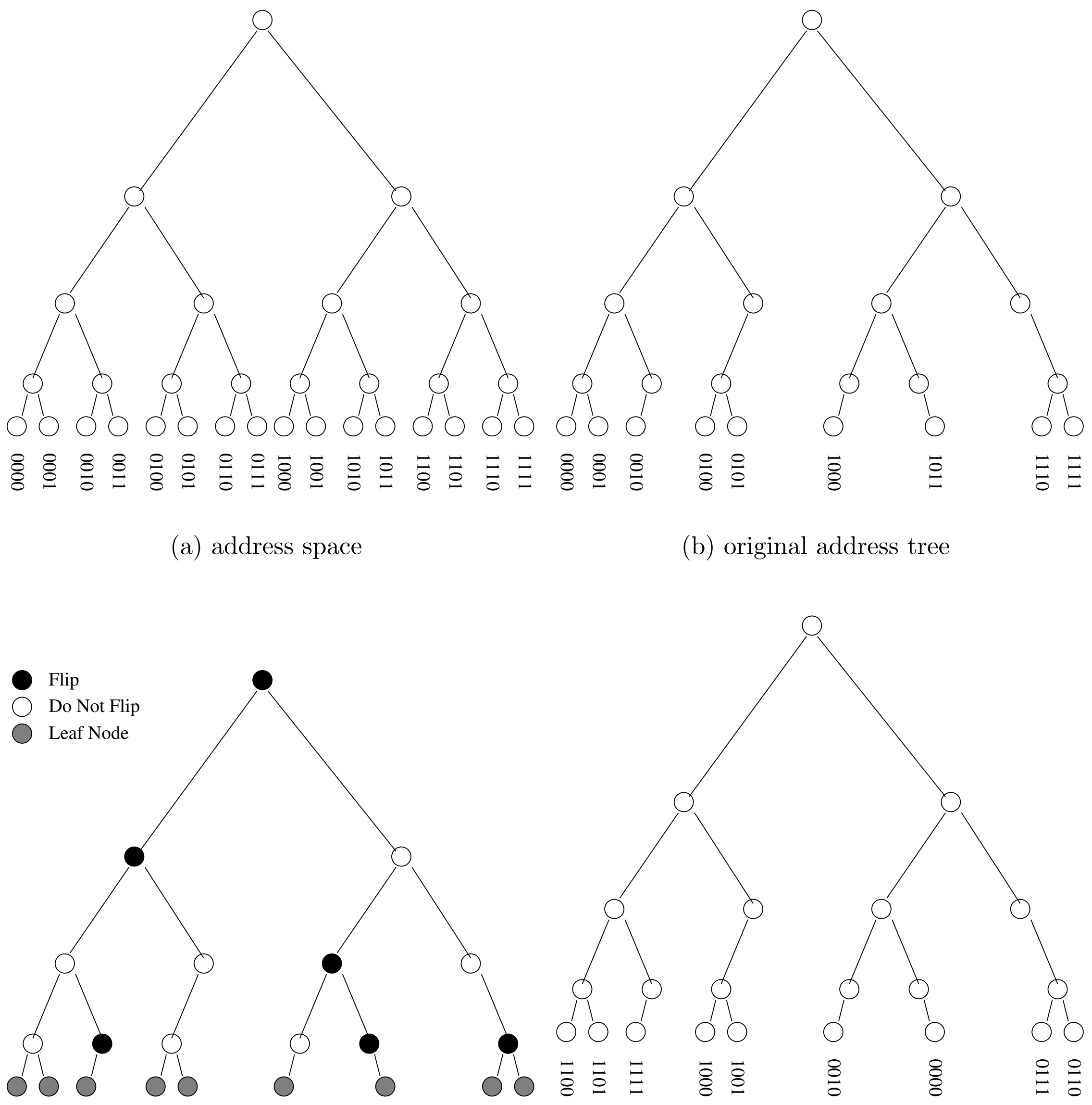
Theorem 1 (Canonical Form Theorem[6]). *(adapted from [6]) Let f_i be a function from $\{0,1\}^i$ to $\{0,1\}$ for $i = 1, 2, \dots, n-1$ and f_0 be a constant function. Let F be a function from $\{0,1\}^n$ to $\{0,1\}^n$ defined as follows. Given $a = a_1a_2 \dots a_n$, let*

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

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 $f(a_1a_2 \ldots a_k) = 1$ or 0, respectively.

[6] proves theorem 1 and finds a suitable anonymization function by using the Rijndael cipher for functions f_i. This approach is implemented in a tool called Crypto-PAn[7]. “Note that there is a natural one-to-one mapping between the canonical form of a prefix-preserving anonymization function and its graphical representation. Each node in an anonymization tree (see Figure 1), as represented by its prefix $a_1, a_2 \cdots a_k$, will be labeled “flip” or “no flip”, when $f(a_1a_2 \ldots a_k) = 1$ or 0, respectively.”[6]



(a) address space

(b) original address tree

(c) anonymization function

(d) anonymized address tree

Figure 1: Address tree and prefix-preserving anonymization function [6]

5 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 lexicographic ordering $<^l$ is defined by*

$$a <^l b \Leftrightarrow a_1a_2\ldots a_n <^l b_1b_2\ldots b_n \Leftrightarrow (\exists m > 0)(\forall i < m)(a_i = b_i) \wedge (a_m < b_m) \qquad (2)$$

[1]

So for example we have the following order on these two IPv4 addresses: $1.2.3.4 <^l 1.12.3.4$.

Please note that this definition treats only the case where both IP addresses are of the same length (eg. 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*

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

In order to preserve the lexicographical ordering 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 $i = 1, 2, \ldots, n$. $used_i$ is defined recursively as*

$$used_i(a_1a_2\ldots a_i) = used_{i+1}(a_1a_2\ldots a_i0) \vee used_{i+1}(a_1a_2\ldots a_i1) \qquad (3)$$

$used_n(a_1a_2\ldots a_n)$ is `true` *if the IP address $a_1a_2\ldots a_i$ is in the traffic trace and* `false` *otherwise.*

This function 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. Figure 2(a) shows the same anonymization function as used in the previous example. Clearly, all nodes in the original address tree have $used_i() = 1$. Observe that 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 ordering. However, if for one of the child nodes we have $used_i = 0$ (there is no IP address from that particular subtree present in the trace), flipping the corresponding bit does not break lexicographical ordering. Figure 2(b) 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(c) and the anonymized address tree (using the restricted anonymization function) is shown in Figure 2(d).

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

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

where

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

$$\begin{aligned} f'_i(a_1, a_2, \ldots, a_i) = f_i(a_1, a_2, \ldots, a_i) \\ \wedge \neg \left(used_{i+1}(a_1, a_2, \ldots, a_i, 0) \wedge used_{i+1}(a_1, a_2, \ldots, a_i, 1)\right) \end{aligned} \tag{6}$$

for $i = 1, 2, \ldots, n$. 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 necessary 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 ordering.

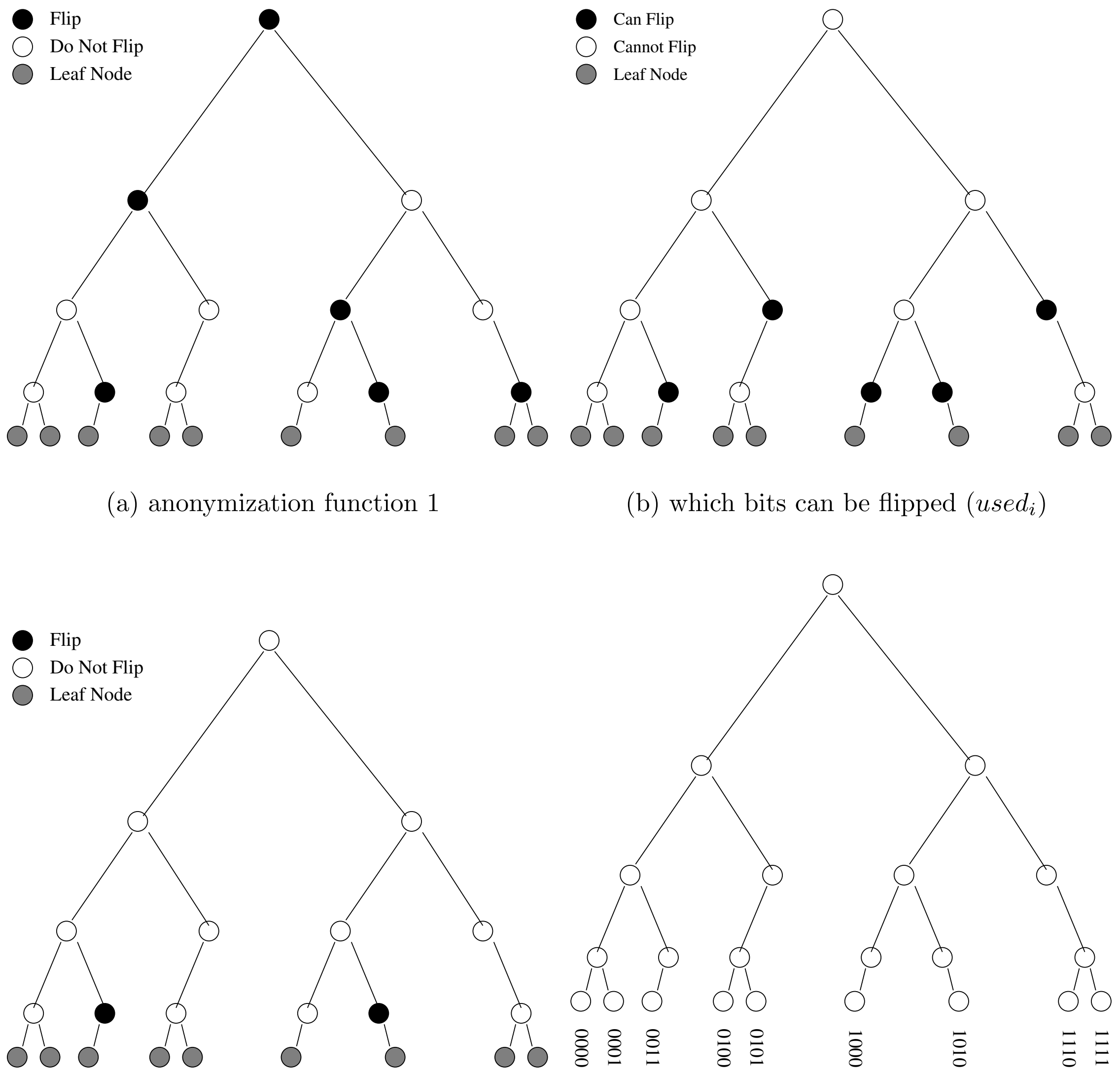
Proof. To show that F is prefix-preserving, it is sufficient to observe that $a'_i = a_i \oplus f'_{i-1}(a_1, a_2, \ldots, a_{i-1})$ is of the form as required by theorem 1 and hence F is prefix-preserving. For proof of theorem 1 please see [6].

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

$$\begin{aligned} a'_i &= a_i \oplus (f_{i-1}(a_1, a_2, \ldots, a_{i-1}) \wedge \neg \left(used_i(a_1, a_2, \ldots, a_{i-1}, 0) \wedge used_i(a_1, a_2, \ldots, a_{i-1}, 1)\right)) \\ &= b_i \oplus (f_{i-1}(b_1, b_2, \ldots, b_{i-1}) \wedge \neg \left(used_i(b_1, b_2, \ldots, b_{i-1}, 0) \wedge used_i(b_1, b_2, \ldots, b_{i-1}, 1)\right)) \\ &= b'_i \end{aligned}$$

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 $a \ne b$.



(a) anonymization function 1 (b) which bits can be flipped ($used_i$)

(c) anonymization function 2 (d) anonymized address tree

Figure 2: Address tree and prefix-preserving lexicographical-order-preserving anonymization function

$$a'_{k+1} = a_{k+1} \oplus f_i(a_1, a_2, \ldots, a_k) \wedge \\ \neg \left(used_{k+1}(a_1, a_2, \ldots, a_k, 0) \wedge used_{k+1}(a_1, a_2, \ldots, a_k, 1)\right) \quad (7)$$

If $f_k(a_1, a_2, \ldots, a_k) = 0$, then from equation 7 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 $f_k(a_1, a_2, \ldots, a_k) = 1$, then we have to consider four possible cases for the values of $used_{k+1}$ in equation 7. Please note that as $a_1 a_2 \ldots a_k = b_1 b_2 \ldots b_k$ we have that

$$used_{k+1}(a_1, a_2, \ldots, a_k, 0) = used_{k+1}(b_1, b_2, \ldots, b_k, 0)$$

and

$$used_{k+1}(a_1, a_2, \ldots, a_k, 1) = used_{k+1}(b_1, b_2, \ldots, b_k, 1)$$

.

1. $used_{k+1}(a_1, a_2, \ldots, a_k, 0) = 0$ and $used_{k+1}(a_1, a_2, \ldots, a_k, 1) = 0$
   The values of $used_i$ imply that no IP address from the subtree below $a_1 a_2 \ldots a_k$ 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.

2. $used_{k+1}(a_1, a_2, \ldots, a_k, 0) = 0$ and $used_{k+1}(a_1, a_2, \ldots, a_k, 1) = 1$
   This implies that one of the IP addresses may 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 the a_{k+1}, b_{k+1} bits may be flipped.

3. $used_{k+1}(a_1, a_2, \ldots, a_k, 0) = 1$ and $used_{k+1}(a_1, a_2, \ldots, a_k, 1) = 0$
   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 the a_{k+1}, b_{k+1} bits may be flipped.

4. $used_{k+1}(a_1, a_2, \ldots, a_k, 0) = 1$ and $used_{k+1}(a_1, a_2, \ldots, a_k, 1) = 1$
   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 $a'_{k+1} < b'_{k+1} \Rightarrow F(a) <^l F(b)$.

6 Security Section

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 all 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 lexicographical order requirement 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, 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 also has to be noted that the address space of IPv6 compared to the one of IPv4 is significantly larger. This would allow for a more secure anonymization in case IPv6 addresses were used in the traces instead of IPv4 addresses. 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 traces anonymization, we have to bear in mind that *Crypto-PAn* was evaluated with respect to traces of real traffic containing IP coming 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 would likely be all from a small set of subnets. Furthermore, attack techniques based on well-known frequently accessed servers like DNS root servers or frequently visited web servers would not be very efficient as their addresses would most probably not figure in the SNMP traces, definitely not with a high frequency.

The number of bits flipped also depends on the choice of key for anonymization. However, choosing the key in such a way that more bits get flipped would not make the anonymization more secure. It would only result 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 might, however, use this idea to define a metric based on how many bits can be flipped:

$$\begin{aligned} q &= \frac{\text{number of times when a bit can be flipped}}{\text{size of address space}} \\ &= \frac{\text{number of times}\neg(used_i(...0) \wedge used_i(...1))}{\text{size of address space}} \end{aligned}$$

7 Implementation

The anonymization function has been implemented as a C library and a sample program to demonstrate its usage. The prefix-preserving part was taken from *Crypto-PAn*, rewritten in C and extended with the lexicographical-order preserving property. Furthermore, AES implementation from the OpenSSL project (`libcrypto` library) 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(b). This approach has rather small computational complexity as adding new nodes into a binary tree and looking up nodes in it is very efficient ($O(\ln \#\text{nodes}) \approx O(\#\text{IP addresses})$). The disadvantage is the memory consumption - for a full address space we would need a tree with $O(\#\text{IP addresses}^2)$ 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. 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 implementation does not actively check for full subnets in order to prune the tree. The approach with building the address tree in memory works for IPv4 addresses (tested with 10^6 randomly generated I addresses), but clearly does not scale well to IPv6 with respect to memory complexity. The current implementation works for IPv4 only.

The library has the following API:

```
typedef uint32_t ipv4_t;

int initialize();
void set_key(const uint8_t * key);
void set_used(ipv4_t ip);
void set_used_prefix(ipv4_t ip,int prefixlen);
ipv4_t anonymize_pref_lex(const ipv4_t orig_addr);
void clean_up();
```

The library first has to be initialized with `initialize()` and the encryption key set with `set_key()`. Then we have to pass the trace file with IP addresses twice. During the first run, we call `set_used()` for each address. Alternatively, we could define complete address prefixes (subnets) to be used with `set_used_prefix()`. During the second run we anonymize the addresses with `anonymize_pref_lex()`. When all the anonymization is finished, memory has to be deallocated with `clean_up()`.

The 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. Additionally, memory consumption has been measured by inserting a `scanf` into the anonymization program after the anonymizations are done but before the memory is deallocated. Then using `pmap ‘pgrep sample-lex‘ | grep total` was used to measure how much memory was actually used by the program. Besides that, a counter has been added to keep track of the number of nodes in the $used_i$ tree. The data type for a tree node takes 16 bytes. This allows us to calculate how much memory was requested for the tree with `malloc`. The rest is constant and can be found by running the program on an empty input file. The results are summarized in Table 1. We can clearly see that after adding the memory footprint of the program on empty input, much more memory is consumed than theoretically requested by `malloc`. Depending on the size of the trace and possible marking of subnets as completely used, the memory consumption might be a problem.

number of IP addresses	number of nodes	measured memory footprint	theoretical `malloc` requests
0	1	2 532K	16
100	2 645	2 536K	42K
1 000	23 172	3 064K	362K
10 000	198 223	7 156K	3 098K
999 999	1 288 7401	304 552K	201 363K

Table 1: Memory footprint (units are Bytes unless stated otherwise)

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. Alternatively, the pointer to parent node could be removed from the node data type. Another possibility would be to to use a custom-made `malloc` adjusted for the tree node data type.

8 Conclusion

A prefix-preserving and lexicographical-order-preserving IP address anonymization scheme has been found by extending the prefix-preserving cryptography-based scheme from *Crypto-PAn* [6] to preserve lexicographical order. The scheme has been rigorously proven to be correct. Its limits as well as security aspects are discussed. Furthermore, it has been implemented in the form of a C library.

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.

As the ultimate goal for future work I would see anonymization of complete SNMP traffic traces. So far, the main problem with this has been the lack of a prefix- and lexicographical-order-preserving IP address anonymization scheme. The found scheme could be integrated with tools like libsnmp and snmpdump, possibly leading to a tool capable of SNMP traces anonymization. Having such a tool would likely relieve the current reluctance of network operators to allow researchers access to SNMP traffic from operation networks. Obtained traces could then be used to analyze the usage, behavior and interaction patterns of SNMP in real-world networks as has been the case with similar projects, successfully dealing with anonymization of FTP traces [4] or router configuration files [2].

References

[1] http://encyclopedia.laborlawtalk.com/Lexicographic_order.

[2] David A. Maltz, Jibin Zhan, Geoffrey Xie, and Hui Zhang. Structure preserving anonymization of router configuration data. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[3] Greg Minshall. tcpdpriv, 1996. http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html.

[4] Rouming Pang and Vern Paxson. A high-level programming environment for packet trace anonymization and transformation. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003.

[5] Jürgen Schönwälder. Characterization of SNMP MIB Modules. In *9th IFIP/IEEE International Symposium on Integrated Network Management*, Nice, May 2005.

[6] Jun Xu, Jinliang Fan, and Mostafa H. Ammar. Prefix-preserving ip address anonymization: measurement-based security evaluation and a new cryptography-based scheme. In *Proceedings of the 10 th IEEE International Conference on Network Protocols (ICNP'02)*, 2002.

[7] Jun Xu, Jinliang Fan, Mostafa H. Ammar, and Sue Moon. Crypto-pan, 2003. http://www.cc.gatech.edu/computing/Telecomm/cryptopan/.

[8] Tatu Ylonen. Thoughts on how to mount an attack on tcpdpriv's "-a50" option... http://ita.ee.lbl.gov/html/contrib/attack50/attack50.html.