CS549 Course Project

Implementing and Attacking Prefix Preserving Encryption

Summary

Project Tasks:

1) Discussion about the pseudo-randomness of CryptoPAn.

CryptoPAn is a python implementation of the Prefix-Preserving encryption scheme for IP addresses as presented in the paper: *Xu. et al, "Prefix-Preserving IP Address Anonymization: Measurement-based Security Evaluation and a New Cryptography-based Scheme", in Proceedings of the IEEE International Conference on Network Protocols, Paris,2002*.

They have demonstrated in their paper that any prefix-preserving anonymization function takes the following form:

$$\mathbf{F(a = a_1, ..., a_N) = a'_1, ..., a'_N \quad where\ a'_i = a_i\ xor\ f_{i-1}(a_1, ..., a_{i-1})\ for\ i = 1, ..., n\ and\ f_0 = cst}$$

As we can see in this construction two addresses that share a k-bit prefix will have the exact same value for $\mathbf{a'_1, ..., a'_k}$. Moreover this also implies that the k+1 bit will be different, therefore if we know the k+1 for one of the two addresses the other address has the complementary value.

In order to get a secure encryption of the IPv4 space we need **F** to be a pseudorandom function (PRF) (a function from $\{0,1\}^n$ to $\{0,1\}^n$ indistinguishable from a uniform function f in $Func_n$. F is a keyed function but the key is not mentioned in the previous formula. To get a PRF we can use a "Block cipher".

They have proposed the following construction method for function f_i:

- $\mathbf{f_i\ (a_1, ..., a_i)}$ =L(R(P($\mathbf{a_1, ..., a_i}$),K))
- P being a padding function that expands the input of size "i" into a longer string that matches the block size of R.
- R is a **block cipher** with key K, Rijndael cipher was chosen by the authors.
- L is the Least Significant Bit

"Rijndael and AES differ only in the range of supported values for the block length and cipher key length. For Rijndael, the block length and the key length can be independently specified to any multiple of 32 bits, with a minimum of 128 bits, and a maximum of 256 bits. **AES** fixes the **block length** to **128 bits**, and supports **key lengths** of **128**, 192 or 256 **bits** only." *(from "Rijndael: Note on naming, 9/04/2003, section 4)*.

Here is the CryptoPAn implementation:

- Requires a 32 bytes key
  - 16 bytes (128 bits) for AES key
  - 16 bytes for padding
- Block cipher: AES; Encryption mode: Electronic Code Book

```
self._cipher = AES.new(key[:16], AES.MODE_ECB)
```

AES CBC mode is usually not recommended as it leaks information about which plaintext blocks are identical. However, in our case we do want the same blocks to be encrypted in the same way otherwise if we had for example two keys in our IP trace their respective encryption will be different

and the only way to differentiate them would be to decrypt the IP trace and this is not the use case this encryption scheme has been designed for!

Indeed here the goal is to preserve IP prefixes once the IP trace has been encrypted and shared with the public or other entities and neither of them will get the symmetric key to decrypt the IP trace, therefore the encryption must be done with ECB mode.

The padding is done by filling the missing bits with the bits from the padding key.

For f_0 the padding will use all 128 bits of the padding key. For f_1 it will use the bits[1:127], for f_2 it will use bits[2:127] and so on.

This is done through this function:

```
def _gen_masks(self):
    """Generates an array of bit masks to calculate n-bits padding data.
    """
    mask128 = reduce (lambda x, y: (x << 1) | y, [1] * 128)
    self._masks = [0] * 128
    for l in range(128):
        # self._masks[0]   <- 128bits all 1
        # self._masks[127] <- 1
        self._masks[l] = mask128 >> l
```

And this code:

```
for pos in range(pos_max):
    prefix = ext_addr >> (128 - pos) << (128 - pos)
    padded_addr = prefix | (self._padding_int & self._masks[pos])
    if sys.version_info.major == 2:
        # for Python2
        f = self._cipher.encrypt(self._to_array(padded_addr, 16).tostring())
    else:
        # for Python3 (and later?)
        f = self._cipher.encrypt(self._to_array(padded_addr, 16).tobytes())
    flip_array.append(bytearray(f)[0] >> 7)
result = reduce(lambda x, y: (x << 1) | y, flip_array)
```

The for loop is the equivalent of $\mathbf{f_{i-1}\ (a_1, \ldots, a_{i-1})\ for\ i = 1, \ldots, n}$. It goes from 0 to 32 or 128 (excluded) for respectfully IPv4 or IPv6 addresses.

In prefix is stored the k-bit prefix where k=pos.

padded_addr is the padded address, padded with 32 (or 128) – pos bits of the padding key. The padding is simply an OR operation between prefix and a mask. The mask for position 0 is [1]*128 (all bits of the padding key are preserved), for position 1 it is [0] + [1]*127 (the first bit of the key is not preserved) and so on.

2) Extend it to ensure prefix preservation for any length of prefixes

CryptoPAn is actually already preserving prefixes of any length. We can see it from the code that I presented above. The loop function was iterating through every single bit of the ip address and therefore preserving prefixes of any length.

If we want to preserve by block of 8 bits we would need to change the f function so that instead of going through bit by bit it goes byte by byte and the for loop would have to do the same. But this is not asked in this project so I haven't tried it.

We can also see that the preservation is bit by bit thanks to the following example:

key = 'ZmI2ZGMzM2QzYWEyYmZlYzlmYzcyZTlm'

cp.anonymize('192.192.168.10') >>> '60.64.87.250'
cp.anonymize('192.193.168.10') >>> '60.65.151.242'
cp.anonymize('192.194.168.10') >>> '60.66.151.254'
cp.anonymize('192.195.168.10') >>> '60.67.169.254'
cp.anonymize('192.196.168.10') >>> '60.71.150.1'
cp.anonymize('192.197.168.10') >>> '60.70.105.249'
cp.anonymize('192.198.168.10') >>> '60.69.150.14'
cp.anonymize('192.199.168.10') >>> '60.68.105.245'
cp.anonymize('192.200.168.10') >>> '60.72.105.254'
cp.anonymize('192.201.168.10') >>> '60.73.169.241'
cp.anonymize('192.202.168.10') >>> '60.75.87.246'
cp.anonymize('192.203.168.10') >>> '60.74.87.242'
cp.anonymize('192.204.168.10') >>> '60.79.105.245'
cp.anonymize('192.205.168.10') >>> '60.78.86.9'
cp.anonymize('192.206.168.10') >>> '60.77.150.13'
cp.anonymize('192.207.168.10') >>> '60.76.104.13'

In this experimentation I voluntarily chose to have my IP going from 192.192.X.X to 192.207.X.X. Between 192 and 207 there is a distance of 15. Also 192= **1100** 0000 and 207 = **1100** 1111 therefore if the scheme is preserving every bit the corresponding addresses should vary from 0000 to 1111 and have at least a 8+4 bits prefix.

original	bit	bit prefix with 192	encryption		bit prefix with 64
192	1100 0000	8	64	0100 0000	8
193	1100 0001	7	65	0100 0001	7
194	1100 0010	6	66	0100 0010	6
195	1100 0011	6	67	0100 0011	6
196	1100 0100	5	71	0100 0111	5
197	1100 0101	5	70	0100 0110	5
198	1100 0110	5	69	0100 0101	5
199	1100 0111	5	68	0100 0100	5
200	1100 1000	4	72	0100 1000	4
201	1100 1001	4	73	0100 1001	4
202	1100 1010	4	75	0100 1011	4
203	1100 1011	4	74	0100 1010	4
204	1100 1100	4	79	0100 1111	4
205	1100 1101	4	78	0100 1110	4
206	1100 1110	4	77	0100 1101	4
207	1100 1111	4	76	0100 1100	4

In the picture above I have shown the consistency in bit prefix with 192 and its encrypted equivalent 64 (it could be done with another number). We can see that the prefix is consistent on both side of the table. This confirms that the prefix consistency is indeed done bit by bit!

3. Perform two groups of attacks: CPA and CCA, respectively.

For each of them, querying n pair(s) of plaintext/ciphertext, n = 1 to 100, the adversary will reconstruct the most bits out of the 10k IP addresses (including 1 brute-force, if you prefer). Implement such two attacks (illustrate the algorithms for your proposed attacks in the report), as well as report 100 groups of results for each of them and their percents of reconstructed bits (out of all the IP addresses).

a) Framework:

I have chosen to perform all operations related to IP addresses in a binary tree. Independently of the chosen attack the code starts by doing the following:

1. The file containing the IP trace is parsed using a python script ("anonymized.py"). The script goes through the file and save each unique IP in a new file called "unique_real_IP.csv". For each of those the corresponding ciphered IP is saved in a separate file called "anonymized_IP.csv". (A third file with frequencies is also created but not used.)
   In all of these three files the addresses are in the base10 representation: 8.8.8.8 (for ex).

```
// Run python script with your IP_FILE.
parseFile(IP_FILE);
```

2. The real ip addresses are stored in a vector of string and then stored one by one in a binary tree.

```
// Get real IP file
std::vector<std::string> rawIPs = readFile(REAL_IP_FILE);

// Push IPs in a binary tree
unsigned int nodeCount(1);
std::shared_ptr<BinaryTreeNode> binaryTree = createTree(rawIPs, &: nodeCount);
```

3. The ciphered ip addresses are stored in a vector of string and the tree is updated so that the nodes corresponding to the real IP are flipped to represent the ciphered ip.

```
// Get anonymous IPs
std::vector<std::string> anonIPs = readFile(ANON_IP_FILE);

// Anonymize the tree
unsigned int nodesFlipped(0);
anonymizeTree(binaryTree, rawIPs, anonIPs, &: nodesFlipped);
```

This framework should be close to the one used in the original paper.

The binary tree representation is ideal to look for/update/add ip addresses in the data structure. It is fast and memory efficient, indeed looking for an IP or adding an IP is always done in constant time: 32 operations (for each of the 32 bits in ip addresses)!

Nodes are defined like this:

```
class BinaryTreeNode
{
private:
    std::shared_ptr<BinaryTreeNode> left_;
    std::shared_ptr<BinaryTreeNode> right_;
    bool bitValue_, flipped, recovered;
```

Each is located at a certain height in the binary tree, and that height is equivalent to a bit position in an ip address. Height 31 corresponds to bits 31 which is the Most Strong Bit and the value of this bit is stored in “bitValue”.

A node can have two children, a left and right one. By default the left is created with a bitValue of 0 and the right one with a value of 1 and the parent node has “flipped=false”. However, this can be reversed to anonymize addresses (see explanation below).

This is an example taken from the paper that details the process of anonymization for 4-bit addresses.

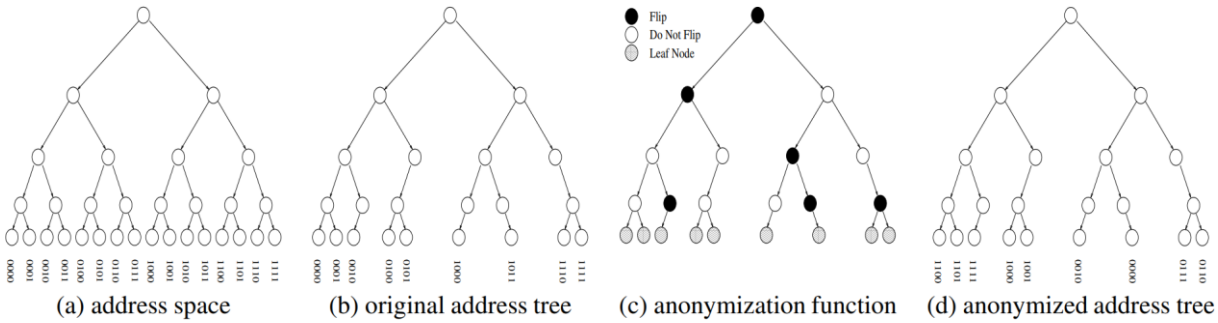


Figure 1. Address Trees and Anonymization Function

In (a) is represented the whole ip space; with 4-bit 16 different IPs from 0000 to 1111 are possible. The **root node** (which does not represent a bit) is at height = 5. The leaves are at height 0.

In (b) is our IP trace; it is a subset of (a).

In (c) some nodes of (b) have been flipped to map the original tree to the anonymous tree. In the case of a length preserving scheme it is indeed possible to go from one mapping to the other by simply flipping node’s state.

Here 0000 is mapped to 1100 which means all real addresses starting by 0 have a ciphered IP starting by 1 and conversely. The only way to have this reflected in the tree is by flipping the root node: all its left children now start with a 1 and its right children start with a 0.

Below is the table with the mapping between (b) and (c).

original Ips	anonymized Ips
0000	1100
0001	1101
0010	1111
0100	1000
0101	1001
1000	0010
1011	0000
1110	0111
1111	0110

In the end given an address and its encrypted equivalent the tree is anonymized like this:

```
void BinaryTreeNode::updateIP(const std::bitset<32>& realIP, const std::bitset<32>& anonymousIP,
                              const std::shared_ptr<BinaryTreeNode>& root, unsigned int& nodesFlipped)
{
    std::shared_ptr<BinaryTreeNode> nodeIndex = root;

    for (unsigned int i = realIP.size() - 1; i < realIP.size(); --i)
    {
        if((realIP[i] != anonymousIP[i]) && !nodeIndex->flipped)
        {
            nodeIndex->flipped = true;
            ++nodesFlipped;
        }
        if(realIP[i] == 0) nodeIndex = nodeIndex->left_;
        else nodeIndex = nodeIndex->right_;
    }
}
```

An example of anonymization is available in Annex 2.

b) Performance metric for CPA and CCA attacks:

To track how well an attack can be I am using two metrics.

The first one is the **number of nodes** that have been **recovered** and the ratio compared to the total number of nodes. The second one is the **number of bits** that are **recovered** (and the ratio with the total number of bits in the address space).

The findings of the attacker outside of the address space are not taken into account, since they do not give any information about this space.

To count the number of nodes recovered by the attacker I use the “recovered” parameter of the nodes. This field is set to “false” when nodes are created and set to “true” when the attacker discover a node’s state either through CPA or CCA.

How many bits can be recovered when the state of a node is revealed to the attacker?

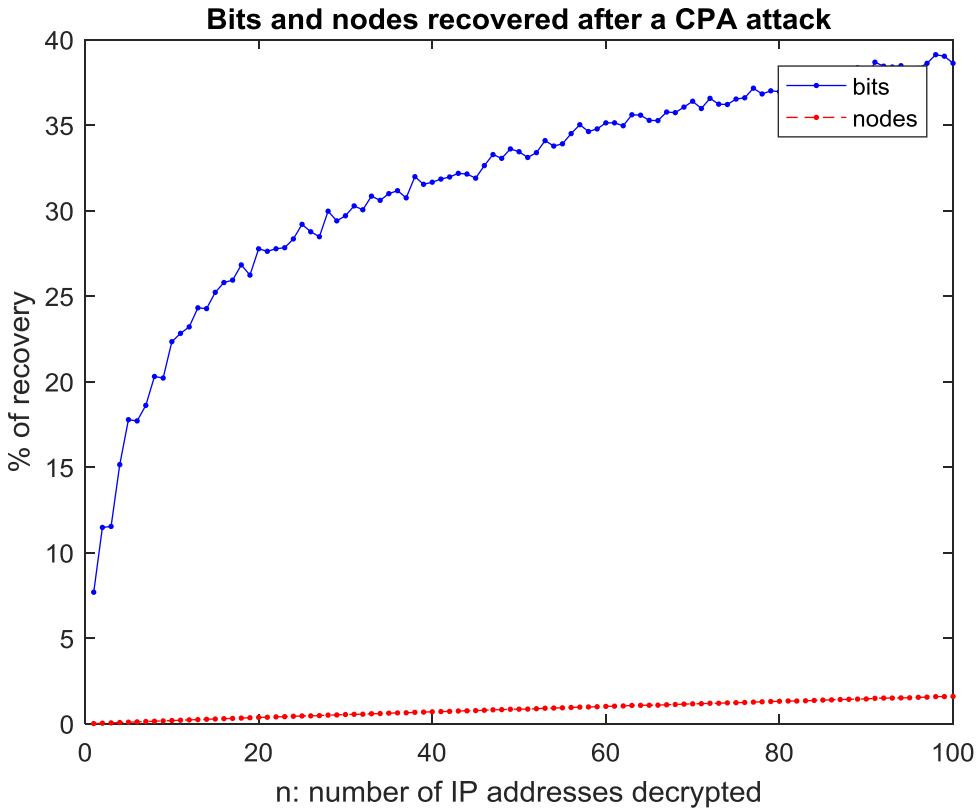
In the example above, finding root node’s state means that the attacker can decrypt the first bit of all IPs. The rule behinds this is that **once a node is revealed** the **number of bits** that are **revealed is** equal to the **number of leaves that have this node as a parent**. Therefore in my code to get the number of bits recovered once a node is revealed I perform a DFS starting from the node and I increase a counter every time a leaf is found.

c) Chosen Plaintext Attack:

For this attack I chose to try by picking "n" addresses at random in the set of given IPs for n: 1 -> 100. Each value of n is tried 10 times and the average result is plotted. To make sure an address is not chosen twice I store in a set the IP indices (indices in the vector of IP) that have already been tried.

Here are some information about the tree space:

- n°nodes = 135,859
- n°bits = 421,824
- unique IPs= 13,182



The results of this experimentation are gathered in two files "all_average_nodes.csv" and "all_average_bits.csv". In those files, line Y is the average value for the recovery ratio of nodes or bits for n = Y. With these values I created the graph below.

The ratio of recovered nodes remains very low as we can see, the red line which represents the ratio of nodes recovered starts at 0.0235 % and ends at **1.6127%**. However this value is not shocking, indeed in the best scenario every time an IP is tried all the nodes are new therefore only 32 new nodes are discovered (31 as we can't count the root twice).

For n=100 that would give a ratio of 31*100 + 1 / total nodes = 3,101/135,859 = **2.2825 %**.Our ratio of 1.6% is actually not very far from the maximum ratio. It is interesting to see that even **with 1.6% of** the total amount of **nodes** we have **recovered** almost **40% of the bits**.

We could make the attack so as the attacker would only decrypt addresses that have a low k-bit prefix. This way more nodes are recovered after each decryption.

To run this attack: /bin/bash runCPA.sh

If you are compiling the code using the bash script mentioned above, the code does not need to be edited. However if you use an IDE you may need to specify the location of the python script as well as the IP.csv file.

d) Chosen CipherText Attack:

1. Attacker get the frequency of each encrypted IP in the IP trace

2. Attacker decrypts the IP from the most frequent to the least

I realized that all addresses in the IP trace that we were given are unique and it is therefore pointless to perform the attack has described above as it will lead to the exact same results as the CPA attack.

From my understanding the only scenario were the CCA attack would yield better results than the CPA attack would be if the attacker in CPA would have no guess on which addresses may be in the tree space and therefore his random choice would be in the IPv4 space and not the tree space. In that scenario the results would most likely be poor as many guesses would correspond to addresses out of the tree space and thus bringing no information to the attacker.

The CCA attacker on the other hand will always have at least the results presented in c) CPA attack. Indeed the attacker only needs to look at the encrypted IP trace and ask the oracle to decrypt any IP of its choice. The CCA attacker is always using addresses that are in the tree space and thus getting the most information that can be possibly recovered.

Appendix 1: python script - anonymization.py:

Command: python3 PATH/TO/IP.csv [print]

If "print" is specified the script prints the key and all unique addresses and their equivalent cipher as in the picture below. The key is always printed.

Three files are created. The first one "*unique_real_IP.csv*" contains all the unique IP addresses and the second one "*anonymized_IP.csv*" contains the equivalent ciphered IPs. Finally the third one contains the frequency of each unique IP.

Note that it is a one to one mapping: an IP at line Y in file 1 has its cipher equivalent at line Y in file2.

A random key is generated at each run using the bash command: "date +%s | sha256sum | base64 | head -c 32 ; echo"

```
gregoire@greg_pad:~/project$ ls
IPtest.csv  __pycache__  anonymize.py  yacryptopan  yacryptopan.py
gregoire@greg_pad:~/project$ python3 anonymize.py IPtest.csv print
key: b'ZWI5NzBiMWE5NzZhYTIyNzJmNzU4ZTk4'

ip: 0.0.0.0
anon_ip: 128.0.15.224

ip: 0.0.0.1
anon_ip: 128.0.15.225

ip: 0.0.0.4
anon_ip: 128.0.15.228

ip: 0.0.0.6
anon_ip: 128.0.15.231

ip: 0.0.0.8
anon_ip: 128.0.15.239

ip: 0.0.0.9
anon_ip: 128.0.15.238
```

The IP test file used for this experimentation is available with the source code.

Appendix 2: anonymization of the tree.

In this experimentation I show that once a tree is created the anonymization function works as expected. In the picture we can see that the algorithm has successfully flipped the good number of nodes to get an anonymized IP from the real IP.

In this example 18 nodes were flipped.

```
IP:0.0.0.0
anon IP: 241.131.248.10
bit32 orgi: 00000000000000000000000000000000 0000
bit32 anon: 11110001100000111111110000000 1010   (13+2 number 1)
n°nodes fliped: 15

IP:0.0.0.1
anon IP: 241.131.248.11
bit32 orgi: 00000000000000000000000000000000 0001
bit32 anon: 11110001100000111111110000000 1011
n°nodes fliped: 0

IP:0.0.0.4
anon IP: 241.131.248.12
bit32 orgi: 00000000000000000000000000000000 0100
bit32 anon: 11110001100000111111110000000 1100
n°nodes fliped: 0

IP:0.0.0.6
anon IP: 241.131.248.14
bit32 orgi: 00000000000000000000000000000000 0110
bit32 anon: 11110001100000111111110000000 1110
n°nodes fliped: 0

IP:0.0.0.8
anon IP: 241.131.248.7
bit32 orgi: 00000000000000000000000000000000 1000
bit32 anon: 11110001100000111111110000000 0111
n°nodes fliped: 3

IP:0.0.0.9
anon IP: 241.131.248.6
bit32 orgi: 00000000000000000000000000000000 1001
bit32 anon: 11110001100000111111110000000 0110
n°nodes fliped: 0
Total nodes flipped: 18
```

This example can be recreated by compiling the code as:

g++ main.cpp binaryTree.cpp main.h -DPRINT_ANON_TREE -o code

With the flag –DDPRINT_ANON_TREE the anonymization function will print out the real IP and its 32 bits decomposition and likewise for the anonymized IP.

The location of the python script and the test may need to be updated.