Zhuowen (Kevin) Zou
CSE 291I: Applied Cryptography

May 11, 2020 21:09
Due: Soon!

Homework 6

0. (a) I used the algorithm described in the paper. In particular, for the 100,000 moduli $\{N_1, ..., N_{100000}\}$, I used the product tree to compute $P = \prod_{i=1}^{100000} N_i$, then used the remainder tree to compute $gcd(P/N_i, N_i)$ for each i. Two of the N_i's turns out to be factored, i.e. $N_i = pq$ for some primes p and q. With that in mind, we now crack the RSA part of the encryption.

   In the standard RSA encryption scheme, we know $ed = 1 \mod (p-1)(q-1)$ for public key e, secret key d, and primes p, q. Thus, given e, p, q, we can use modular inverse and extended CRT* to find d. Then for each N_i and their corresponding d, we try to decrypt the file. A correct decryption will be indicated by the correct padding $0002[random]00[data]$. And we're done.

   The hardest part for me is actually getting Sage right. Recall that multiprecision integers works much faster in Sage mode than in normal python, and due to the lack of documentation for sage 8.0, I was a bit lost in applying the correct division for my remainder tree.

   Implementation detail: recall that the remainder tree computation require modding $\prod_{i=a}^{b} N_i^2$ for some a, b that defines the boundary of dichotomy in the tree. These values are computed in my product tree which uses a DFS algorithm to traverse the tree in a post-order manner, which makes the stack that stores the squares post-order. When using the stack, the remainder tree is traversed using a reversed post-order manner.

   Memory-wise, it is better to use BFS for both trees because the storage of products and remainders can be occupied by only two levels of the tree, roughly halfing the memory requirement for my algorithm. Thank Prof. Heighner for suggestion on Sage usage and algorithm optimization.

   *extended CRT allows non-coprime modular constrain to be considered together.

   (b) 1) Key choice: PBP is using 1024-bit RSA to encrypt 256-bit AES, which makes most part of the padded AES string to be the padding. It is rather inefficient, though it pose no internal security threats. In addition, current recommendation of RSA key length is 2048 instead of 1024. 2) It doesn't provide cipher text integrity. It is possible to change the part of the cipher text that does not concern the aes key.

1. (a) The intuition is that for every query, we can figure out about one bit of information about x.

Suppose we make the following queries:

$$P_x(2^i) = c_i$$

for $i = 0, 1, ..., \lfloor log_2 N \rfloor$ and query replies $c_i \in \{0, 1\}$. For the first query $i = 0$, we know whether $x > N/2$ or $x < N/2$, and thus $x = x_0 = c_0(N/2) + x_1$, where x_0 is defined to be x and x_1 is defined to be $x_0 - c_0(N/2) < N/2$. For the second query, we know whether $2x > N/2$ or $2x < N/2$. Notice that $2x = 2((x - c_0(N/2)) + c_0(N/2)) = 2(x - N/2) = 2(x_1) \mod N$. This means that we now know whether $x_1 > N/4$ ($c_1 = 1$) or $N/2 > x_1 > N/4$ ($c_1 = 0$), and thus $x = x_0 = c_0(N/2) + c_1(N/4) + x_2$ for some integer $x_2 < N/4$.

By induction, given c_k and $x = \sum_{i=0}^{k-1} c_i \frac{N}{2^{(i+1)}} + x_k$ for some $0 <= x_k < \frac{N}{2^k}$, we may deduce from the query k that $x = \sum_{i=0}^{k} c_i \frac{N}{2^{(i+1)}} + x_{k+1}$ for some $0 <= x_{k+1} < \frac{N}{2^{k+1}}$. To show this:

$$c_k = 1 \iff N/2 < 2^k x = 2^k(\sum_{i=0}^{k-1} c_i \frac{N}{2^{(i+1)}} + x_k) = (\sum_{i=0}^{k-1} c_i \frac{N}{2^{(i+1-k)}} + 2^k x_k) = 2^k x_k \mod N$$

$$\iff x_k > \frac{N}{2^{k+1}} \iff x_k = x_{k+1} + \frac{N}{2^{k+1}} \text{ for some } 0 <= x_{k+1} < \frac{N}{2^{k+1}},$$

And for the other case,

$$c_k = 0 \iff N/2 > 2^k x_k \mod N \iff x_{k+1} \triangleq x_k < \frac{N}{2^{k+1}}$$

.

We now conclude that, because x is an integer,

$$x = \sum_{i=0}^{\lfloor log_2 N \rfloor} c_i \frac{N}{2^{(i+1)}}.$$

This is obvious: an $x_{\lfloor log_2 N \rfloor + 1}$ will have to be smaller than 1, which has to be 0 for an integer x. Thus roughly $log_2 N$ points are necessary and sufficient to know x.

(b) The intuition is that we treat the $c^{1/e}$ in the question as x, and do the "attack" described above. It is easy, however, because we are only required to know how to query $2^i x$ for all i. Since

$$2^i x = 2^i c^{1/e} = ((2^e)^i c)^{1/e},$$

we may make the following $\lfloor log_2 N \rfloor + 1$ queries, where $d = 2^e$ and O is the oracle:

$$c_i = O(d^i c)$$

for $i = 0, 1, ..., \lfloor log_2 N \rfloor$. And from the result in (a), we have

$$c^{1/e} = \sum_{i=0}^{\lfloor log_2 N \rfloor} c_i \frac{N}{2^{(i+1)}}$$