CS 346: Cryptography **Fall 2024**

Homework 5: Public-Key Cryptography

Due: December 4, 2024 at 11:59pm (Submit on Gradescope) **Instructor:** David Wu

Instructions. You **must** typeset your solution in LaTeX using the provided template:

https://www.cs.utexas.edu/~dwu4/courses/fa24/static/homework.tex

You must submit your problem set via Gradescope (accessible through Canvas).

Collaboration Policy. You may discuss your general *high-level* strategy with other students, but you may not share any written documents or code. You should not search online for solutions to these problems. If you do consult external sources, you must cite them in your submission. You must include the names of all of your collaborators with your submission. Refer to the official course policies for the full details.

Problem 1: Collision-Resistant Hashing from RSA [14 points]. Let $N = pq$ be an RSA modulus and take $e \in \mathbb{N}$ to be a prime that is also relatively prime to $\varphi(N)$. Let $u \xleftarrow{R} \mathbb{Z}_N^*$, and define the hash function

$$H_{N,e,u} \colon \mathbb{Z}_N^* \times \{0,\ldots,e-1\} \to \mathbb{Z}_N^* \quad \text{where} \quad H_{N,e,u}(x,y) = x^e u^y \in \mathbb{Z}_N^*.$$

In this problem, we will show that under the RSA assumption, $H_{N,e,u}$ defined above is collision-resistant. Namely, suppose there is an efficient adversary $\mathcal{A}$ that takes as input (N,e,u) and outputs $(x_1,y_1) \neq (x_2,y_2)$ such that $H_{N,e,u}(x_1,y_1) = H_{N,e,u}(x_2,y_2)$. We will use $\mathcal{A}$ to construct an efficient adversary $\mathcal{B}$ that takes as input (N,e,u) where $u \xleftarrow{R} \mathbb{Z}_N^*$ and outputs x such that $x^e = u \in \mathbb{Z}_N^*$.

(a) Show that using algorithm $\mathcal{A}$ defined above, algorithm $\mathcal{B}$ can efficiently compute $a \in \mathbb{Z}_N$ and $b \in \mathbb{Z}$ such that $a^e = u^b \pmod N$ and $0 \neq |b| < e$. Remember to argue why any inverses you compute will exist.

(b) Use the above relation to show how $\mathcal{B}$ can *efficiently* compute $x \in \mathbb{Z}_N$ such that $x^e = u$. Note that $\mathcal{B}$ does *not* know the factorization of N, so $\mathcal{B}$ cannot compute $b^{-1} \pmod{\varphi(N)}$. **Hint:** Consider the value of $\gcd(b,e)$ and Bezout's identity.

Problem 2: Baby Bleichenbacher [24 points]. In this problem, we will explore a simplified variant of Bleichenbacher's CCA attack against PKCS#1 encryption. Let (N,e) be the public-key for an RSA-based encryption scheme, where $N = pq$ is a product of two primes and e is invertible modulo $\varphi(N)$ (i.e., there exists d such that $ed = 1 \bmod \varphi(N)$). For $x \in \mathbb{Z}_N$, define the function $f_x \colon \mathbb{Z}_N \to \{0,1\}$ where $f_x(r) = 1$ if the value of $x \cdot r \pmod N$ is greater than $N/2$ (where we view $x \cdot r \pmod N$ as an integer between 0 and $N-1$), and 0 otherwise.

(a) Construct an algorithm that given $O(\log N)$ queries to f_x recovers the value of $x \in \mathbb{Z}_N$. Your algorithm can make *arbitrary* queries to f_x. Prove the correctness of your algorithm.

(b) Suppose an adversary has intercepted an RSA ciphertext $c \in \mathbb{Z}_N$ where $c = x^e \pmod N$ for some $x \in \mathbb{Z}_N$. Moreover, suppose the adversary has access to a "partial" decryption oracle that takes as input an input $z \in \mathbb{Z}_N$ and outputs 1 if $z^d \pmod N$ is greater than $N/2$ (where we view $z^d \pmod N$

as an integer between 0 and $N-1$), and $d = e^{-1} \pmod{\varphi(N)}$ is the RSA decryption constant. Use your result from Part (a) to show how the adversary can decrypt c to obtain the message x by making $O(\log N)$ queries to this partial decryption oracle.

Problem 3: Bleichenbacher Attack on PKCS#1 [30 points]. Recall that a RSA-PKCS#1 encryption of a message $m \in \{0,1\}^n$ under an RSA public key (N, e) is formed by first constructing a padded message $x \leftarrow \mathsf{PKCS}(m) \in \mathbb{Z}_N$ and then computing $c \leftarrow x^e \in \mathbb{Z}_N$. This scheme was used in the SSL 3.0 handshake.

In the SSL 3.0 handshake, the client chooses a random "pre-master secret" k (used to derive the session keys) and encrypts k using RSA-PKCS#1 under the server's public key to obtain a ciphertext c. Upon receiving the encrypted key c, the server attempts to decrypt the message; if the decryption yields a message that is not a well-formed PKCS#1 message, the server sends an abort message to the client. Otherwise, it continues with the handshake. In this problem, we will say that a message has a proper PKCS#1 padding if it starts with the two-byte sequence $00\|04$.[1] Bleichenbacher showed that this single bit of leakage (via a "padding oracle") can be leveraged to mount a full key-recovery attack against SSL 3.0. In this problem, you will implement this attack.

We have provided starter code that contains a basic implementation of RSA-PKCS#1 encryption (see `main.py`) with our modified PKCS#1 header. Your objective is to implement Bleichenbacher's attack (described below). In particular, your algorithm should be able to decrypt an intercepted RSA-PKCS#1 ciphertext given knowledge of only the public key (N, e) and access to the following padding oracle:

> *on input a ciphertext $c \in \mathbb{Z}_N$, the padding oracle outputs* 1 *if $c^d \in \mathbb{Z}_N$ is a valid (modified) PKCS#1 message (starts with the two-byte sequence $00\|04$) and* 0 *otherwise*

We include a description of Bleichenbacher's attack (for our modified PKCS#1 header) below. You may also refer to the original attack description (for standard PKCS#1 headers) in Section 3.1 of this paper. See Section 3.2 for the analysis of the algorithm (this is not required for implementing the algorithm, but can be beneficial for understanding why the algorithm works).

In the following, we say a ciphertext $c \in \mathbb{Z}_N$ is "PKCS-conforming" if it decrypts to a message that has a valid PKCS#1 padding (as defined according to the above oracle). Then, on input a PKCS-conforming ciphertext $c \in \mathbb{Z}_N$, Bleichenbacher's attack proceeds as follows:

1. Let k be the length of N in bytes and let $B = 2^{8(k-2)}$. Initialize $M_0 \leftarrow \{[4B, 5B-1]\}$ and $i \leftarrow 1$.
2. Let $s_0 \geq N/(5B)$ be the smallest integer such that the ciphertext $c(s_0)^e \bmod N$ is PKCS-conforming.
3. Depending on the number of intervals the set M_i contains, proceed as follows:
   - If M_{i-1} contains exactly one interval $[a, b]$, then choose small integer values r_i, s_i such that
   $$r_i \geq 2\frac{bs_{i-1} - 4B}{N}$$
   and
   $$\frac{4B + r_i N}{b} \leq s_i < \frac{5B + r_i N}{a}$$
   until the ciphertext $c(s_i)^e \bmod N$ is PKCS-conforming.

[1] In the actual PKCS#1 padding scheme, the header is the sequence $00\|02$; we use a modified header in this assignment. In practice, PKCS#1 will also check that the header is followed by sufficiently-many non-zero padding bytes, but for simplicity in this problem, we will ignore this detail and only require checking the first two bytes.

- If M_{i-1} contains two or more intervals, then search for the smallest integer $s_i > s_{i-1}$ such that the ciphertext $c(s_i)^e \bmod N$ is PKCS-conforming.

After computing s_i, the set M_i is updated as follows:

$$M_i \leftarrow \bigcup_{(a,b,r)} \left\{ \left\lceil \max\left(a, \left\lceil \frac{4B + rN}{s_i} \right\rceil\right), \min\left(b, \left\lfloor \frac{5B - 1 + rN}{s_i} \right\rfloor\right) \right\rceil \right\}$$

for all $[a,b] \in M_{i-1}$ and $\frac{as_i - 5B + 1}{N} \leq r \leq \frac{bs_i - 4B}{N}$. If M_i contains a single interval of length 1 (i.e., $M_i = \{[a,a]\}$), then output a as the solution. Otherwise, set $i \leftarrow i + 1$ and repeat.

The invariant in the above algorithm is that the message associated with c is always contained in one of the intervals in the set M_i.

In our starter code, we use a 128-bit RSA modulus (which is easily factored), but your implementation should also support a 1024-bit RSA modulus with several minutes of computation (factoring a general 1024-bit modulus is well beyond the reach of current techniques). Your task is to implement the `decrypt` method in `bleichenbacher.py`. You **cannot** change the interface for `__init__` or `decrypt`; otherwise, you are free to implement the algorithm however you prefer (using *standard* Python libraries). Your code will be evaluated only for correctness (so if you prefer a different approach to breaking RSA-PKCS#1, such as factoring the modulus, that is also acceptable[2]). Some helper functions are provided in `util.py`.

Debugging hints:

- You should avoid all floating-point operations. Floating-point operations have limited precision and you are working with very large integers (far beyond what is supported by standard floating-point values). You should use integer operations throughout your implementation. For instance, to compute $\lfloor a/b \rfloor$, you should write `a // b` and for $\lceil a/b \rceil$, you should write `(a + b - 1) // b`.
- The most time-consuming part of the algorithm is usually finding the initial s_1. For a 1024-bit modulus, this can take a few minutes.
- The set M_i usually contains a single interval. The case where M_i has multiple intervals is uncommon.
- In the main loop, the size of the interval typically decreases by a factor of ≈ 2 in each iteration.
- A helpful invariant you can use is that at every step in the protocol, the padded PKCS message must be contained in the set of intervals M. You can check that this is true at the very beginning, and that this remains true after each update step. You can view Bleichenbacher's algorithm as a special case of your algorithm from Problem 2 where the oracle output tells you whether x lies in a certain interval of size B (rather than whether x is greater than or smaller than $N/2$).
- When testing with larger primes p, q, you need to choose them so they are valid RSA moduli (i.e., $\gcd(e, \varphi(N)) = 1$).

Submission instructions: Upload your code (consisting of *only* `bleichenbacher.py`) to Gradescope under Homework 5A. Note that your implementation must work with our provided `main.py` and `util.py`. Your submission will be autograded, and upon submission, your code will be run on a simple test case. There is **no** written component for this question.

[2] Indeed, if you break the 1024-bit scheme via a factoring algorithm, you will automatically receive an "A" in this course (and probably throw in a Ph.D. too).

Problem 4: Time Spent [2 points]. How long did you spend on this problem set? This is for calibration purposes, and the response you provide does not affect your score.

Optional Feedback. Please answer the following *optional* questions to help us design future problem sets. You do not need to answer these questions. However, we do encourage you to provide us feedback on how to improve the course experience.

(a) What was your favorite problem on this problem set? Why?

(b) What was your least favorite problem on this problem set? Why?

(c) Do you have any other feedback for this problem set?

(d) Do you have any other feedback on the course overall?