CS 6222: Introduction to Cryptography **Spring 2020**

Homework 4: Public-Key Cryptography

Due: April 15, 2020 at 5pm (Submit on Gradescope) **Instructor:** David Wu

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

https://www.cs.virginia.edu/dwu4/courses/sp20/static/homework.tex

You must submit your problem set via Gradescope. Please use course code **MB84NW** to sign up.

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 computing IDs of all of your collaborators with your submission. Refer to the official course policies for the full details.

Problem 1. Baby Bleichenbacher [25 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.

In the next problem, we will explore the full Bleichenbacher attack on PKCS#1 encryption. It turns out that the handshake in SSL 3.0 implements a variant of the partial decryption oracle described above, which enables an active adversary to decrypt any ciphertext of its choosing (in the case of SSL 3.0, this was the "pre-master secret" used to derive the keys for the record layer)!

Problem 2. Bleichenbacher Attack on PKCS#1 [35 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 the padded message $x \leftarrow \texttt{00}\|\texttt{02}\|r\|\texttt{00}\|m \in \mathbb{Z}_N$ where r consists of a sequence of random non-zero bytes, and then computing $c \leftarrow x^e \in \mathbb{Z}_N$. This scheme was used in the SSL 3.0 handshake.

Specifically, 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, and otherwise, it continues with the handshake. In this problem, we will say that a valid PKCS#1 ciphertext is one that starts with the two-byte sequence $00\|02$.[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.

(a) Bleichenbacher's attack is described in Section 3.1 of this paper. We have provided starter code that contains a basic implementation of RSA-PKCS#1 encryption (see `main.py`). Your objective is to implement Bleichenbacher's attack described in Section 3.1 of the paper. 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 PKCS#1 message (starts with the two-byte sequence $00\|02$) and* 0 *otherwise*

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 should not 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`. For the submission, please upload your code (consisting of *only* `bleichenbacher.py` to Gradescope under Homework 4A). Note that your implementation must work with our provided `main.py` and `util.py`.

(b) Bleichenbacher's attack still applies if the server performs additional validation on the PKCS#1 message. For instance, Bleichenbacher considers a valid PKCS#1 message to be one that starts with $00\|02$ and is followed by 8 non-zero padding bytes. Suppose the padding oracle outputs 1 only if the ciphertext decrypts to a message that satisfies this requirement. How would this affect the concrete running time of the algorithm you implemented above? Explain briefly. Note that you do not need to perform any concrete calculations here (a short 2-3 sentence explanation suffices). **Hint:** Please feel free to refer to Section 3.2 of Bleichenbacher's paper.

Problem 3. DDH in Composite-Order Groups [10 points]. Let $\mathbb{G}$ be a cyclic group of order $2q$ where q is odd. Let g be a generator of $\mathbb{G}$. Show that the DDH assumption does not hold in $\mathbb{G}$. *Remark:* This shows that the DDH assumption does not hold over $\mathbb{Z}_p^*$ whenever $p = 2q+1$ for some odd q. In fact, the DDH assumption does not hold in $\mathbb{Z}_p^*$ for any prime p (there is an efficient distinguisher based on Legendre symbols). However, assumptions such as CDH or discrete log still plausibly hold over $\mathbb{Z}_p^*$.

Problem 4. Time-Lock Puzzles [20 points]. A time-lock puzzle is a cryptographic mechanism that enables someone to send a message to the future (e.g., the earliest the message can be decrypted is a week from today or a month from today).

[1] 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.

[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).

(a) Consider the following approach based on any symmetric encryption scheme. Sample a random key $k \xleftarrow{R} \{0,1\}^\lambda$ and output the ciphertext $\mathsf{ct} \leftarrow \mathsf{Encrypt}(k, m)$ together with the first $\lambda - t$ bits of the secret key k. To decrypt ct, the adversary would have to brute force the last t bits of the key, which would take time 2^t. By choosing t accordingly, the encrypter can control the minimal amount of time needed before the secret can be revealed. Briefly explain why this scheme is not secure: namely, describe how a *parallel* adversary with ℓ processors can recover m in time $2^t/\ell$. *Note:* This is intended to be a warm-up and a one sentence response suffices.

(b) To construct a time-lock puzzle, it is essential to base it on a *sequential* computation (that is not amenable to parallelism). One such candidate is modular exponentiation in an RSA group. Let $N = pq$ be an RSA modulus and sample a random $g \xleftarrow{R} \mathbb{Z}_N^*$. It is believed that computing the function $f_{N,g}(x) := g^x \in \mathbb{Z}_N$ requires $\Omega(\log x)$ sequential multiplications. This yields the following approach for constructing a time-lock puzzle. The puzzle with time parameter t is the triple (N, g, t) and the solution is the value $z = g^{2^{(2^t)}} \pmod N$. We can use this to encrypt a message by hashing z to obtain a key k and using k to encrypt the message. Show that there is an algorithm that recovers z from (N, g, t) using $O(2^t)$ modular multiplications (over $\mathbb{Z}_N$). Once again, the encrypter can choose the value of t based on the duration under which the secret should remain hidden.

(c) In the above construction, the encrypter needs to be able to efficiently compute the solution z in order to encrypt their message. Show that the encrypter (who chooses N and $g \xleftarrow{R} \mathbb{Z}_N^*$) is able to compute z with only $O(t)$ modular multiplications. Thus, this gives a time-lock puzzle where the decrypter needs to run in time 2^t while the encrypter runs in time t for a tunable parameter t.

(d) In 1-2 sentences, explain why the puzzle solver cannot use the algorithm in Part (c) to solve the puzzle with $O(t)$ multiplications.

(e) Suppose we implement the above construction over $\mathbb{Z}_p$ for prime p. Show that this construction is insecure (i.e., the decrypter can now recover the secret in time much smaller than $O(2^t)$).

Remark: In 1999, Ron Rivest (one of the inventors of RSA) prepared a time-lock puzzle as part of the opening of MIT's Laboratory for Computer Science, to be opened on the 35th anniversary of the lab's opening. As it turned out, two independent groups successfully solved the time-lock puzzle earlier last year (15 years earlier than anticipated)! As it turns out, estimating the speed of computers 35 years into the future is no easy task.

Problem 5: Time Spent [3 extra credit 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. To receive the extra credit for this problem, you must submit your homework to Gradescope (with the provided template) and properly assign *all* problems to their respective pages.

Optional Feedback [0 points]. 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 so far?