CSCI 3330 Assignment 7

Deadline: November 18, 2025, 10:00pm

1 Background

In this assignment, you will be developing and executing two different attacks on textbook RSA. Specifically, by completing this assignment, you will:

- Design and execute an oracle-driven cryptanalytic attack to recover and verify a hidden message.
- Leverage RSA's multiplicative structure to craft ciphertext manipulations and interpret server signals to extract bits of a key.
- Analyze how an attempt to "fix" one side channel introduces a new side channel and exploit it to reconstruct an encrypted symmetric key.

2 AI policy

You may use AI to learn background concepts. You may not use AI to generate, verify, or iterate on solutions for this assignment.

Allowed:

- Concept definitions and explanations: "What is RSA?", "How does RSA + AES work?", "What is prime factorization?"
- Generic examples not derived from this assignment: "Illustrate an attack on textbook RSA," "Walk through an example of how to discover a side channel oracle."
- Non-generative tooling: man pages, official documentation, calculators, programming languages, and trying something for yourself.

Handle with care:

- Tools which improve grammar or clarity.
- IDE autocompletion or documentation lookup that does not inject solution content.

Not allowed:

- Any hint, outline, or solution for these specific questions (including rephrased or "similar" versions), e.g., "Help me with the attack in Q2," "Write an RSA side-channel attack," or "Check my answers."
- Pasting assignment text or numbers into AI to tailor examples.
- Using AI to generate any text, code, calculations, or any other artifacts that you submit.

Ask me questions instead! When in doubt, don't use AI, because **you won't have access to it on tests**, for which these assignments are your preparation.

3 Threat model

In this assignment, like the last, we are once again trying to decrypt encrypted coordinates sent to a server. Unlike the previous assignment, the CBC padding oracle has been eliminated, but OAEP padding has been removed from the RSA implementation. Therefore, the RSA used is what is called "textbook RSA". In the following two sections, you will implement two attacks, one on a simple server that performs few checks and then another on a "fixed" server that attempts to close the side channel exploited in the first section by performing an additional check.

4 Attack against textbook RSA

In this section, you will implement `crack.py`, an attack against RSA that assumes an oracle that (1) informs you of whether your decrypted payload is well-formed and (2) ignores bits that are "too high" (i.e., bits beyond the lowest 128 bits) in the decrypted key. You will be working with the files in the tarball here to attack this encrypted request.

At a high level, in this request, `c` corresponds to the encrypted payload, `iv` corresponds to the IV with which it was encrypted, and `k` corresponds to the RSA-encrypted key with which the payload was AES-encrypted in CTR mode (all of these values are hex-encoded). To understand the protocol at a deeper level, consult `client.py`, a sample implementation which encrypts and transmits a sample message. The linked tarball also includes `crack-template.py`, which you may use as a template for implementing `crack.py`.

1. Implement the `oracle()` function. This function takes an AES-encrypted payload, an IV, and an RSA-encrypted key, and, by analyzing the server's response, returns `True` if the server, after RSA-decrypting the passed encrypted key and using its lowest 128 bits to AES-decrypt the passed encrypted payload, has decrypted a well-formed coordinate message, else `False`. **Hint:** this oracle implementation should be different than the one used in the previous assignment.

2. Implement the `decrypt()` function. This function takes an AES-encrypted payload, an IV, and an RSA-encrypted key, and decrypts and returns the payload that was encrypted with that AES key.

   In the body of this function you should implement a side channel attack which learns bits of the encrypted AES key one bit at a time. Your code should recover the 128-bit AES key via oracle queries (128 total) and then use it to decrypt the encrypted payload and then return the payload that you decrypted. You may find the following python programming hints helpful in implementing this function:

   - To convert any byte sequence b into an integer, call the python code `int.from_bytes(b, 'big')`.
   - To convert a 128-bit AES key integer i into a byte sequence, call `i.to_bytes(128 // 8, 'big')`.
   - To convert a 2048-bit encrypted key integer i into a byte sequence, call `i.to_bytes(2048 // 8, 'big')`. (Since the RSA public key used in this exercise is 2048 bits, all integers encrypted by it will be 2048 bits regardless of their original size.)
   - To raise 2 to the power x modulo n, call `pow(2, x, n)`.

Now execute your attack in `main()`. What is the decrypted coordinate message?

5 Attack against "fixed" RSA

In this section, you will implement `crack2.py`, an attack against RSA that assumes a different oracle than in the previous section. The oracle you face in this section has been "fixed" so that decryption of the 128-bit AES key will fail if bits that are "too high" are set (i.e., when the decrypted key is $\geq 2^{128}$). Thus, the side channel that you exploited in the previous section will no longer work, but this "fix" incidentally introduces a new one for you to exploit. Specifically, you will exploit an oracle that signals when bits are "too high" in the decrypted key. You will be working with the files in the tarball here to attack this encrypted request.

Note that in the `crack2.py` template file, functions may take arguments that you do not need to use. It is an exercise to determine which arguments are actually relevant to your attack.

1. Implement the `oracle()` function. This function takes an AES-encrypted payload, an IV, and an RSA-encrypted key, and, by analyzing the server's response, returns `True` if the server decrypted a key with no bits that are "too high" set (i.e., a key $< 2^{128}$), else `False`. **Hint:** to determine what response signals keys that are too large, consider modifying the key that you encrypt in `client2.py` to be too large.

2. Set `TEXT_PAYLOAD` to the decrypted plaintext you found in the previous section. It must be of type bytes, so if you found `'0.0, 0.0'` then set it to `b'0.0, 0.0'` (note the `b` prefix). In this section we will assume for simplicity that the encrypted payload decrypts to the same plaintext as in the previous section. You will use this value later when brute forcing the decrypted AES key to determine whether each key correctly decrypted the payload.

3. Implement the `factor()` function. This function takes an AES-encrypted payload, an IV, and an RSA-encrypted key and returns all prime factors of the decrypted AES key less than or equal to `LIMIT`. Optionally, a fourth argument can be passed which overrides the default limit of `LIMIT`. You should implement this function by making carefully crafted queries to `oracle()`. Using this function with the default limit you should be able to identify 15 prime factors of the decrypted key featured in this exercise. As a test vector that does not take as long to execute, `factor(c, iv, k, 100)` should return `[2, 3, 3, 5, 7, 11, 19, 79]`. **Hint:** the decrypted key may be divisible by a prime factor more than once, i.e., a prime factor may have multiplicity greater than one. **Hint:** this step will take the longest to execute, so you may want to copy and paste its return value into `main()` to save time on future runs.

4. Implement the `bound()` function. This function takes an AES-encrypted payload, an IV, an RSA-encrypted key, and the product F of all primes identified by `factor()` and returns a value x such that $xR < 2^{128}$ and $(x+1)R \geq 2^{128}$, where the decrypted AES key $k = F \cdot R$. The implementation of this function should use binary search to find x searching the space between 0 and 2^{128}. Therefore, this function should not use more than $\log_2 2^{128} = 128$ calls to `oracle()`. In this exercise, the return value of this function should be a 32-digit decimal number beginning in "202" and ending in "696".

5. Implement the `crack_key()` function. This function takes an AES-encrypted payload, an IV, an RSA-en crypted key, all factors identified by `factor()`, a lower bound such that $R \geq \left\lceil \frac{2^{128}}{x+1} \right\rceil$, and an upper bound such that $R \leq \left\lfloor \frac{2^{128}-1}{x} \right\rfloor$ and returns R, where x and R are defined as in the previous step. This function should find R by brute forcing all R in between the lower bound and upper bound, **inclusive**. When you test the correct R, the corresponding AES key $k = F \cdot R$ should decrypt the encrypted payload yielding the value `TEXT_PAYLOAD`, the same decrypted payload as found in the previous section. Note: in a realistic attack scenario, you may not already know the decrypted payload, so you would need to do something more complicated to determine whether you have correctly decrypted the payload such as determining whether the plaintext has decrypted to valid ASCII bytes or valid coordinates.

Executing `main()`, what is the AES key that decrypts the encrypted coordinate payload?

6 Submission

Email your code and the decrypted payload from Section 4 and the derived AES key from Section 5 in a single `*.txt` or `*.pdf` file attachment to my email address in the Syllabus.