University of Illinois, Urbana Champaign
CS/ECE 498AC3/4 Applied Cryptography

Instructor: Dakshita Khurana
Scribe: Luke Cessna
Date: September 29, 2020

LECTURE
11

CCA Secure Encryption

The focus of this lecture is CCA (chosen ciphertext attack) secure encryption. If the reader recalls the previous two lectures, we were introduced to the idea of RSA encryption. What prompted this exploration of RSA encryption was the desire to create public-key encryption. Up until the lectures 9, and 10 we have been assuming two parties, Alice and Bob, were able to achieve semantic security in the face of an eavesdropper assuming they had somehow both met and agreed upon a secret key k in advance. However, we know that such a system is entirely impractical, and now armed with our knowledge of hard problems aquired from the previous two lectures, we can begin our discussion of real-world public-key encryptions in earnest. To motivate this discussion, the lecture examines CCA secure encryption by examining implementations of RSA (Rivest-Shamir-Adleman) encryption. Namely, we examine Bleichenbacher's CCA attack on some of the first versions of RSA encryption. Additionally, we broach the topics of some vital mechanics in exploring such attacks, such as what a random oracle is. Finally, we briefly begin the discussion of trusted third parties to aid against man-in-the-middle attacks.

11.1 A Bit of Backround Knowledge

First, I would like to remind the reader of the textbook RSA system. This is important to keep in mind throughout this reading as all of the systems we are discussing here are based on the assumption of the hardness of factoring primes and the RSA system is a formal defininition of *how*, at least for right now, we put this idea into pratice to encrypt.

Definition 11.1. Textbook RSA system:

- Choose random prime numbers p, q with sizes Approx. 1024 bits and set N = p*q.
- Choose integers e, d s.t. e*d = 1 mod(phi(N))
- output pk = (N, e), sk = (N, d) where pk is the public encyption key and sk is the secret key
- With this, we define RSA encryption as RSA-Enc(pk, x) = x^e (in Z_N)

- and RSA decryption as RSA-Dec(pk, y) = y^d (in Z_N)

We can easily verify that this satisfies correctness $y^d = x^e * d = x^{kphi(N)+1} = (x^{phi(N)})^k * x$ = $(1)^k * x$ = x. Remember that this RSA system is called a trapdoor permutation, which is defined as follows:

Definition 11.2. Trapdoor permutation:

- Three algorithms: $(G, F, F^{(-1)})$
- G: outputs pk, sk. pk defines a function F(pk,*): X→X
- F(pk,x): evalutates the function at x
- F^{-1}(sk, y): inverts the function at y using sk, gives x s.t. F(pk, x) = y

Addintionally, a secure trapdoor permutation is defined to have one more requirement:

Definition 11.3. Secure trapdoor permutation, same requirements as above plus:

- The function F(pk,*) is one-way without the trapdoor sk

This last requirement is an important one, recall that we rely on the hardness of problems for provable security, and if the function F(pk,*) is a one-way function without the secret key k, then we know that this trapdoor permutation is secure. Additionally, before I move on to the rest of the lecture, I would like to point out an important point that is reiterated repeatedly throughout the course. I stated above that we know the trapdoor permutation is provablely secure if the function F(pk,*) is a one-way function without the secret key k. However, as we have seen in other lectures, security systems we have thought secure in the past proved to be broken rather easily. When we talk about these formal definintions, we must remember that it comes down to the implementation of the encryption we are dealing with. As such, this means that the best way to prove the security of something is through repeated tests, and building encryptions on systems that have, so far, withstood the test of time and remain secure. Finally, I state here the RSA assumption. This, as implied by the name, is an assumption that we take without a formal proof other than our knowledge of hard problems. This assumption is an important one because we are basing the security of the hardness problem encryptions we are looking at on it.

Definition 11.4. RSA assumption:

- RSA is a one-way permutation, and for all efficient algorithms A:
- $Pr[A(N, e, y) = y^{(1/e)}] < neglible$
- where p,q ←R n-bit primes, N←p*q, y←R Z_N^*

To state this in plain Engilsh, because we are assuming the hardness of factoring primes, and these encryptions are built on these primes, we can conclude that an adversary A's advantage is related to their ablity to factor primes quickly, which is negligble.

11.2 RSA, Bleichenbacher, and CCA

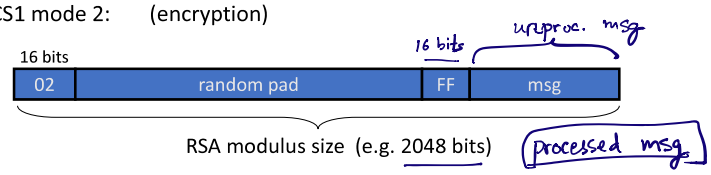
Now we move on to the bulk of lecture 11. Early RSA encryptions and how they are broken by a Bleichencher attack, which is a CCA. An imporant realization about the textbook RSA system is that it is not semantically secure because it is deterministic. Consider a semantic security attack game. Then, an adversary A that sent m_0, m_1 to the challenger and got back c_b could simply send $m_0 = m_1$ to the challenger, where m_b here is equal to the previous m_0. If the c they get back is the same as c_b then they output 0 else output 1. So, we must somehow make our RSA system semantically secure. To do this we must make RSA-Enc probabalistic. this is just like any other probablistic function we have looked at so far, we must add a parameter r that is some form of randomness sampled uniformly. For RSA this looks like the following, RSA-Enc(pk,x;r) = (r^e, H(r)XORm). Two things to Note here. The hash function relies on the fact that r is unpredictable and using solely r in place of H(r) would be bad. The first note is clear, if r were easy to predict, then H(*), which is public, would not be any help in encrypting our message. The second note is very important to rememeber if you were to build your own encryption. r's unpredictablity does not make it a good key to encrypt m with. Take for example the case in which r is 100-bits long (the length for this example doesn't matter), when r is sampled it could be the case in which r's first 30-bits are zero followed by a uniform distribution. This clearly won't work for our purposes of encrypting as we require the entire key to be uniformly distributed. This is why we use H(r) which we assume to be indistinguishable from a uniform distribution.

Let us take a moment and briefly talk about speeding up RSA. Because we assume we are running on computers that are polynomial-bounded, we are often interested in finding ways to improve runtime. To that end we veiw one way of speeding up RSA-Enc. Take some c = m^emod(N), the smaller e is the quicker we will encrpyt. However, the smaller e will cause d to be larger, which would make Decryption slower, as e*d = 1 mod(phi(N)) and RSA-Dec(pk, y) = y^d (in Z_N). The point of this little thought experiment is not to make you always have speed in mind when thinking about encryptions, but rather, when you do begin to think about speedups, you have to think about the potential costs, which could even be security. e must still be a value greater than or equal to 2^{16}+1 to ensure security.

Back to RSA, we have begun our discussion of RSA in practice with PKCS1 v1.5. What is important to take away from figure **??** is the padding at the front of the processed message. I will explain why this is important after this next figure **??** Now, keep in mind both of these figures as I explain this attack. The main idea behind this attack is that the server reveals information about the message. Note that this only matters if the attacker has access to some ciphertext c that is not their own. Now, as in figure 11.2 understand that we are choosing some r, ignore what that is for now, rasied to the power of e which we procced to multiply with the cipherext c. This gives us $(r * (processmessage))^e$ as we know $a^b * c^b$ is just $(a * c)^b$. now that we have this new ciphertext c', we send it to the server. The server responds either yes, the message is valid, or no the message is somehow not valid. At this point, we have to consider our choice of r. If we chose r such that r^e*c is just a bit shifting of the underlying message m, when the server responds yes or no, we know exactly what that bit we are checking is, a 0 or a 1, provided we choose r correctly. This means that with repeated queries involving bit shiftings of c that are sent to the server we can decrypt the entirety of the underlying message m. This is the Bleichenbacher attack, and an excellent example of what a CCA entails. The lecture contains a simplfied version of this attack in which we know that the first bit is always 1 instead of a padding of 02 I include the slide **??** containing this example for the reader if they have difficultly understanding how

RSA in practice: PKCS1 v1.5

PKCS1 mode 2: (encryption)



- Resulting value is RSA encrypted (processed msg)^e.
- Widely deployed, e.g. in HTTPS
- Suffered from a CCA attack

FIGURE 11.1: PKCS1 v1.5 example.

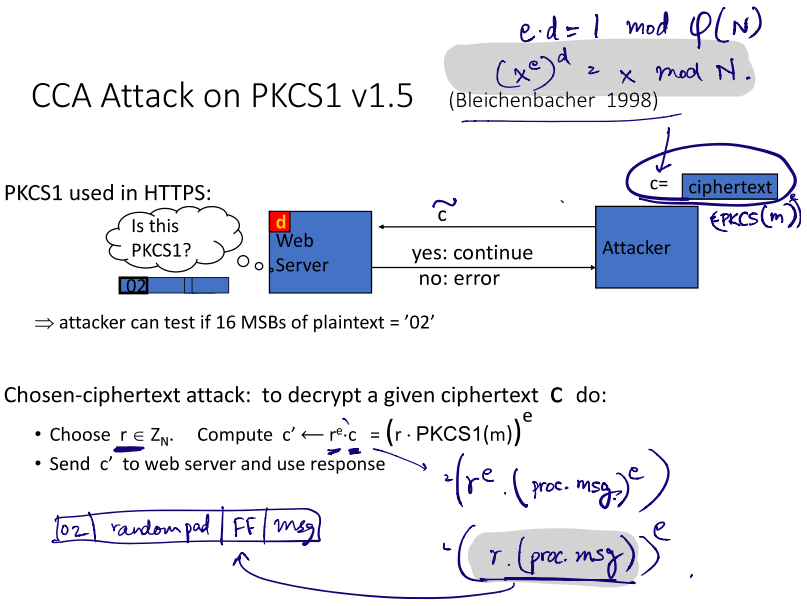


FIGURE 11.2: CCA Attack on PKCS1 v1.5 example.

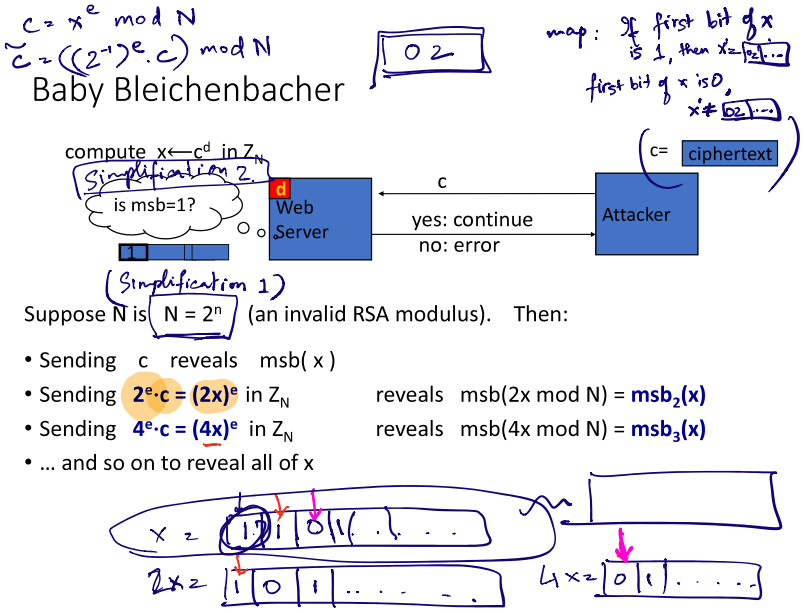


FIGURE 11.3: CCA Attack on PKCS1 v1.5 simplified example.

such an attack applies in the precceding discussion in which the message is more advanced. The precceding explaination of the attack applies in exactly the same way, just with a few simplifications so that we don't have to handwave how r might be chosen.

Here I am going to define the chosen ciphertext security for public key encryption, which is rather simple. Consider some challenger C and Adversary A. C sends pk, a public key to A. Now, A sends any number of ciphertexts c that with one stipulation, for any c_a, c_b in the set of ciphertexts c_n A sends to C, c_a cannot equal c_b. Ther reason for this should seem clear as any encryption that is deterministic can be broken in such an attack game as A can easily tell which message it was just given by checking the ciphertexts it already sent again. Now, C sends back to A $Dec_sk(c_n)$. Eventually, A decides to send m_0, m_1 to C and recieves c = $Enc_pk(m_b)$. A can then choose to continue sending ciphertexts and receiving decryptions of them until it outputs b' = 0 or 1. It should appear obvious where we are going with this. We deine CCS for public key encryption as —Pr[b' = 1— b = 0] - Pr[b' = 1—b=1]— = negligible where b is the b in m_b. Put more simply, CSS for public key encryption is the probablity that an Adversary A outputs b' = 1 in game 0 minus the probablity A outputs b' = 1 in game 1, which must be negligible to pass. I include the figure **??** to give a pictorial representation of this process. Next I briefly touch on PKCS1 v2.0 OAEP. PKCS1 v2.0 OAEP is an attempt to fix the issues with previous versions. Here

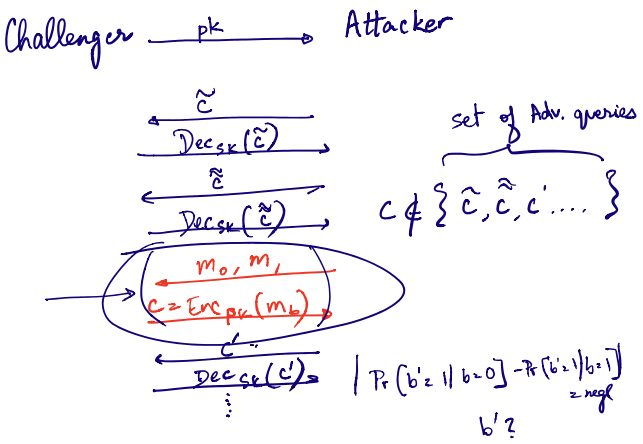


FIGURE 11.4: chosen ciphertext security for public key encryption attack game.

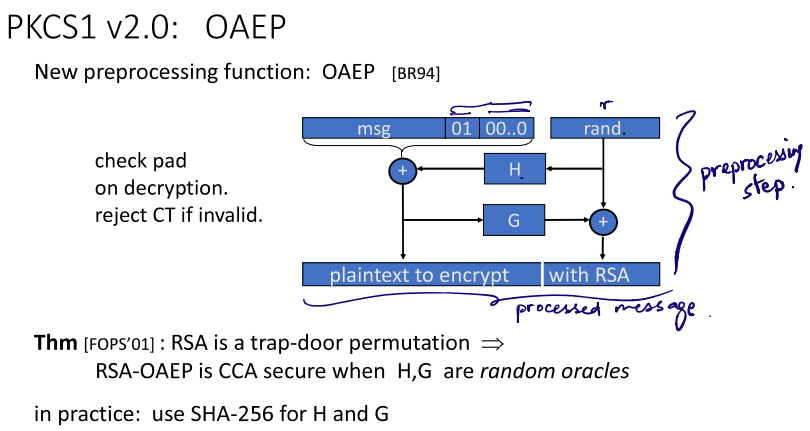


FIGURE 11.5: PKCS1 v2.0 OAEP example

I include the figure **??** to demonstrate how this works. The important point in this figure **??** is that if RSA is a trap-door permuation then RSA-OAEP is CCA secure when H, G are random oracles. This leads us to one of our most important topics in this lecture. Here I define random oracles.

DEFINITION 11.5. Random Oracles:

- A random oracle requires three things
- that is a function H that is a truly random function, with memory.
- it is "observerable"
- it is "programmable"

First, a random oracle is considered to be a function such that on any given input, it outputs a uniform random string, except, if that input has already been seen before, output the same uniform random string as before. This I hope is self explanatory. Second, observerable means that for an attacker to obtain z = H(y), it must querry the oracle on y. This means that it is impossible for an attacker to obatin one of the uniform random

strings of H(*) without using H(*) to get that uniform random string. In fact, we even assume something stronger. That is, given an attack game in which an attacker receives H(*) in one game and H(y) in another, it cannot tell the difference between the two games. Third, programmable is used only in the context of a challenger wrapped in an oracle. Programmable means that the challenger decides what the oracle outputs, while observerable means that the challenger gets to see what the oracle sees. This might seem confusing at first, but again, think of an oracle as a wrapper around a challenger. This wrapper gives the challenger control over what it outputs. These ideas of observerable and programmable simply allow a challenger to modify a function that ouputs uniform randomness. We shall see how this is useful when we analyze OAEP.

Going back to the OAEP and assuming that H and G are random oracles, we can construct the idea of plaintext awareness. First, the reason we go though the trouble of creating a random oracle is so we can say that if an attacker querries and obtains z then it must have already known y and it gains no new information. If we look at figure **??** we see this is what the decryption function does for OAEP. It takes two outputs from the encryption function pt_1, pt_2 and does two checks generating r_1 and m_1. Because pt_1 is generated by XORing the output of a random oracle with m_1, and we assume that the properties of the random oracles hold, then this is indeed true. For an attacker to generate pt_1, then first they must know r_1 because of the properties of random oracles they cannot guess the output of H(r_1) to be XORed with m_1, then, the only way for an attacker to generate pt_1 is to XOR m_1 with the output of H(r_1), meaning the attacker must know m_1 if they know both r_1 and pt_1.

11.3 An Introduction to Authenticated Key Exchange

Finally, we are quickly introduced to authenticated key exchange. We now know how to send secret keys in the presence of an eavesdropper; however, we do not know how to deal with an eavesdropper that controls the network. This is known as a man-in-the-middle-attack. To give a better picture of how such an attack works, imagine Alice tries to send Bob an encrypted secret key so that they can set up their shared secret key. The man in the middle can intercept that message and establish a secret key with Alice himself and then procced to pretend to be Alice and setup a secret key with Bob. This is obviously a problem that our current systems cannot handle. This is where trusted third parties come in. In a general sense , a trusted third party is someone that everyone who wishes to certify who they are talking to goes to. This person then provides some kind of proof to the third party and they in return get a certificate to give to others that proves they are who they say they are. This certificate is check against a public verification key for that person, accepting if they are who they say they are. To do this we need to understand signature schemes. This scheme is define in the following way, Alice posts vk, verification key, and keeps a signk, signature key. Then, Bob sends a message m to Alice and in return receives sigma such that it is a function signature(signk, m) that is "signed" using her signk and can be verified using vk. We can use this in the place of the trusted third party discussed above. Now, we want to define an idea of security for this system s.t. an attacker cannot claim to be another person and recieve the verification. Let us define an attack game with challenger C and attacker A. C starts by sending a vk to A and now A can begin making querries of m_b and receives in return sigma = sign(signk, m). The attacker wins if they find some (m', sigma') s.t. verify(m',sigma',vk) = 1. I include the figure 11.7 to help visualize this process.

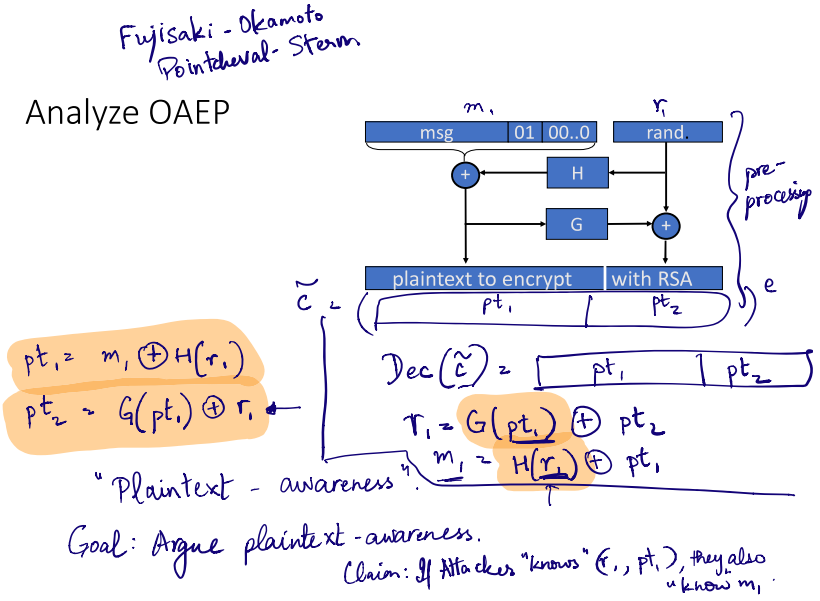


FIGURE 11.6: PKCS1 v2.0 OAEP revisted

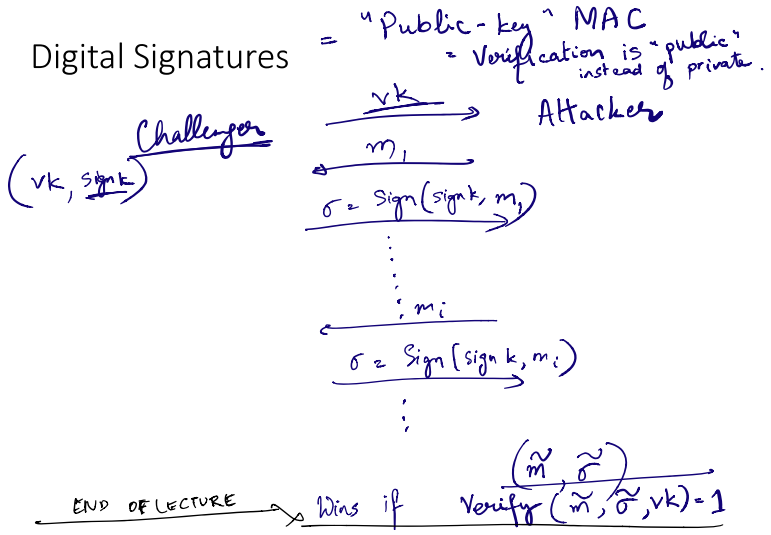


FIGURE 11.7: signature scheme attack game

11.4 Summary

In this lecture we learn how RSA is implemented in real life, what CCAs are and how they are carried out, the extremely important idea of random oracles and how they can be used to prove security, and finally we begin the topic of authenticated key exchange.

Acknowledgement

These scribe notes were prepared by editing a light modification of the template designed by Alexander Sherstov.

References

[1] D. Khurana. CCA Sercure Encryption. AC3/AC4 Lecture slides, 2020.