Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter 7, Public-key cryptography

§7.1.2 Public-key encryption and its security; §7.2.1 RSA and RSA encryption padding; §7.3.1 Hybrid encryption and KEMs; PKE from (EC)DH. Note: this draft excerpt contains several unfinished section headings with no text under them yet (book pages 133–138, 141–145, 148–151)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
Catch-up reading for the Oct 12 lecture (RSA and public-key encryption), posted Oct 19

`https://practicalcrypto.org`

This is an excerpt from an unfinished book draft: expect gaps, rough edges, and errors, and note that section numbers and page numbers may change in later drafts. Cross-references to other chapters point to material that is not included here. Copyright © 2026 Matthew D. Green. All rights reserved. Posted for the students of the course above; **please do not redistribute** — link to the course site instead. Built 2026-10-19.

MITM attack, short for Meddler-in-the-Middle (or Man-in-the-Middle), and it's illustrated in Figure 7.2.

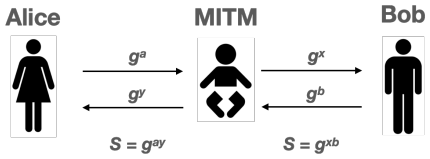

Figure 7.2: MITM attack against Diffie-Hellman. Here the (baby) attacker substitutes its own values g^x and g^y, causing Alice and Bob to each derive a different shared secret with the attacker. If Alice and Bob use this secret for encryption, the attacker can decrypt and re-encrypt each message as it passes through the channel.

This attack stems from the fact that Diffie-Hellman messages are easy to forge: in the naïve protocol there is no way for Alice to know if a message was sent by the "real" Bob, or someone pretending to be Bob. Making Diffie-Hellman secure against active attackers requires that Alice and Bob must have some way to authenticate that messages they receive really come from the right party. The most common way to do this is for *e.g.,* Bob to digitally sign his messages, using a key that Alice can be certain belongs to him. This is accomplished in TLS using signatures and digital certificates, as we discuss in §9.2.1.

DRAFT

7.1.2 Public Key Encryption

A limitation of the traditional Diffie-Hellman key exchange protocol is that the two parties must be *online* when establishing a key. This is because the original protocol is fundamentally interactive: Alice must send a message, then Bob must send back his own. Only then can the pair securely communicate using their shared secret key. This can work well when both parties are connected to the Internet, but it may be a problem if, for example, Bob is asleep and his phone is not connected to the Internet; in this case Alice will have to wait for him to come back online before she can send her message. This is a real problem that comes up in badly-designed encrypted messaging apps (Figure 7.3).

To address this situation, Ron Rivest, Adi Shamir and Leonard Adleman proposed a slightly different primitive: *public-key encryption.* As with key exchange, public-key encryption allows Alice and Bob to communicate securely without having pre-exchanged secret keys. However, in a public-key encryption scheme, there is less need for interaction. To receive messages from anyone, Bob generates a *pair* of keys: a public key pk, and a secret key sk. Bob can post his public key anywhere — on a website, on a specialized "key server", or even printed on a T-shirt. Then Alice (or any other party) can encrypt a message to Bob using only his public key. Only Bob can decrypt the message using his secret key.

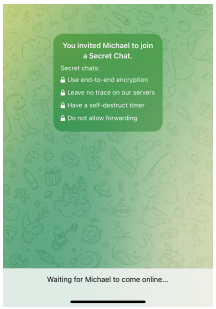

Figure 7.3: Starting an encrypted chat with an offline party in Telegram v5.0.

DRAFT

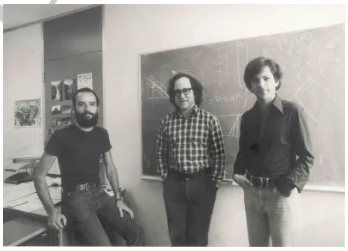

Figure 7.4: Adi Shamir, Ron Rivest and Leonard Adleman.

In the textbook description, a public-key encryption scheme consists of three different algorithms.

1. $\mathsf{Keygen}(1^\lambda) \rightarrow (pk, sk)$. This algorithm is executed by the recipient. It takes in some optional information, such as the desired key strength, which we denote by λ,[4] and outputs a *keypair* consisting of a public key pk and secret key sk.
2. $\mathsf{Encrypt}(pk, M) \rightarrow C$. To encrypt a message M, any sender can provide a recipient's public key pk. The encryption algorithm outputs a ciphertext C that can only be decrypted by the recipient.
3. $\mathsf{Decrypt}(sk, C) \rightarrow M$. The recipient applies this algorithm on input her secret key sk and a ciphertext C to obtain a decryption. This algorithm outputs the decrypted plaintext M, or error if decryption fails.

Key encapsulation: public-key encryption from Diffie-Hellman. Although Rivest, Shamir and Adleman proposed their own public-key encryption scheme, called the RSA cryptosystem, this scheme is not necessarily the one we recommend people use today. Instead, modern public-key encryption is frequently built using *hybrid encryption*, often on the back of the elliptic curve variant of Diffie-Hellman. The basic intuition here is almost identical to the normal DH key exchange, except that Alice's "key share" is constructed only one time and published as her public key $pk_A = g^a$, with the secret exponent a forming her secret key sk.[5]

With Alice's key pk_A in hand, Bob (or anyone else) can generate his own *ephemeral* public key $B = g^b$, and then derive a shared secret value $S = g^{ab} = pk_A^b$. This is hashed, typically along with the public key and B using a specialized hash function called a *key derivation function* to obtain a symmetric encryption key:[6]

$$k \leftarrow \mathsf{KDF}(pk_A \| B \| pk_A^b)$$

You'll notice that this is not quite a full public-key encryption scheme yet: instead, we've built something called a *key encapsulation scheme*. To complete the picture and encrypt a meaningful message to Alice, we also require a symmetric authenticated encryption (AEAD) scheme to encrypt the actual message M. With this piece in place, a full "hybrid" ciphertext might look like this:

$$C = (B, \mathsf{AEAD.Encrypt}(k, M))$$

[4] In the textbook description, we write this input as 1^λ, which indicates a string of `1` bits of length λ. This bit of notational chicanery is useful, since we can now describe the running time of the algorithm as a function of its input length, AKA λ.

[5] Note that for this intuition we are using standard finite-field Diffie-Hellman instead of its elliptic curve variant.

[6] Although Diffie and Hellman made this observation, the researcher Taher Elgamal proposed a more complete version of the cryptosystem, which now bears his name.

DRAFT

A concrete recommendation: HPKE. Of course, the above description is just an intuition. If you're looking for an overall recommendation on which public-key encryption scheme to use today, I strongly recommend the HPKE standard [**?**]. HPKE, short for *hybrid public-key encryption*, is a flexible and extensible standard that incorporates the elliptic-curve Diffie-Hellman scheme, together with specific authenticated encryption (AEAD) schemes, in a manner that's guaranteed to provide security. HPKE is almost identical to the intuition given above, but provides nearly all of the remaining details and recommends specific AEAD schemes and Diffie-Hellman parameters.

Security of public-key encryption

At one level, the "security" of a public-key encryption scheme seems like it should be easy to define. After all, what we want is simple: *nobody should be able to decrypt our messages!* Over time, however, cryptographers came to realize that nothing is so easy. In part this is because the mathematical definition of "decrypt" is subtle, and kept running up against real-world situations. For example, what if my scheme resists full decryption, but you can learn a few bits of the message (or about the message) without fully decrypting it? Similarly, what if the attacker has access to a "decryption oracle" that allows them to decrypt some messages, but not others?

Ciphertext indistinguishability ("semantic security"). In the mid-1980s, cryptographers Shafi Goldwasser and Silvio Micali considered a very simple question [138]: *what happens if we restrict ourselves to encrypting messages in a small set?* For example, imagine a situation where the actual message sent by an army commander is one of two possible values, such as `ATTACK` or `DEFEND`. If an attacker sees the ciphertext, can they somehow learn something about which message was being encrypted, perhaps by encrypting both messages on their own using the public key?

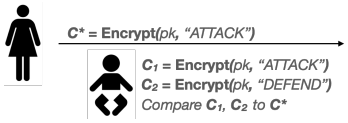

Figure 7.5: An attack on deterministic encryption. The attacker compares an intercepted ciphertext C^* to the encryption of some candidate plaintexts.

The two scientists quickly realized that *deterministic* encryption schemes — schemes in which a given plaintext always encrypts to the same ciphertext, under a given key — would be a disaster in this setting: after all, if an attacker has the scheme's public key and suspects that the message might be `ATTACK`, they can simply encrypt that message to learn what the resulting ciphertext looks like; if any sender is foolish enough to send the same value,

DRAFT

the attacker will instantly know what they are saying! To address this problem, Goldwasser and Micali suggested the use of *probabilistic*, or randomized encryption. This ensures that encryptions of a given message will almost never produce the same ciphertext. To formalize this, they also proposed a stronger notion of encryption security known as (ciphertext) *indistinguishability under chosen-plaintext attack* (IND-CPA), or "semantic security" to judge encryption schemes.

The idea of the semantic security definition is as follows: an attacker who sees the encryption of a message should learn effectively *nothing* about the message itself even after seeing the ciphertext, even if they possess the ability to encrypt messages by themselves (because they have the public key.) While the original Goldwasser/Micali definition was a bit complicated, there is a much simpler "game"-based notion that can express the same concept. It involves a game played between an honest "challenger" and some attacker, as follows:

1. The challenger generates $(pk, sk) \leftarrow \mathsf{Keygen}(1^\lambda)$ and gives pk to the attacker.
2. The attacker can then choose any two *allowed* plaintexts:[7] we'll call these M_0, M_1.
3. The challenger selects a b at random, and gives the attacker a challenge ciphertext of the form:
$$C^* \leftarrow \mathsf{Encrypt}(pk, M_b)$$
4. The attacker makes a guess b' for which of the two messages was encrypted. It wins if $b' = b$.

DRAFT

Note that if even if the attacker never sees the ciphertext C^*, it will still be able to win this game with probability 1/2 simply by guessing! So in practice, we are really interested in how much better (or worse!) the attacker does than this simple guessing strategy: the difference the attacker's success probability and 1/2 is called its *advantage*. The semantic security notion (IND-CPA) holds for a scheme if, for *every possible (polynomial-time) attacker* the advantage of the game is at most negligible in the scheme's security parameter.

Chosen ciphertext security. While semantic security (IND-CPA) is a good start, it is not really sufficient to secure a public-key encryption scheme. Over the years, cryptographers kept thinking of new ways to break encryption schemes. One powerful attack they noticed comes about when the attacker has access to a *decryption oracle*. this refers to any system that contains the secret key and will decrypt ciphertexts of the attacker's choosing.

[7] Allowed plaintexts means that these are plaintexts that the encryption scheme can encrypt. If the encryption scheme has a fixed set of messages, such as RSA, then this can be any pair of those messages. If it is a scheme that takes in a variable-length message, then we will typically restrict the messages so that they are both equally long: this prevents a simple attack where the size of the ciphertext gives away which message was encrypted.

At first blush, decryption oracles seem like a fairly silly thing to worry about. To channel my students: *if the attacker has a decryption oracle, isn't the game already over?* However, it turns out that decryption oracles are surprisingly common in the real world, often in ways that don't trivially break the system. The most common examples include the situation where the attacker's access to the decryption oracle is temporary, or where the decryption oracle only gives emphpartial information about the decryption of a ciphertext.

Temporary oracle access ("non-adaptive" chosen ciphertext attacks). Once upon a time, decryption was performed on machines with actual operators, who might get hungry and leave the machine unattended. The aptly-named "lunchtime attack" imagines that during this time window, an attacker might sneak in for a while (Mission-Impossible-style) and use the machine to decrypt some ciphertexts of their choosing.

The wrinkle in this scenario is that the attacker's access is only temporary. For example, after the attacker leaves the room, they might intercept a brand-new ciphertext. By this time, the operator has returned from lunch and the attacker no longer has access to the decryption oracle. The question raised in this scenario is whether the attacker's earlier decryption queries might somehow give it a more durable advantage in breaking the scheme. In some sense, non-adaptive chosen ciphertext attacks aren't really attacks on specific ciphertexts at all: they are attacks that try to learn something about the scheme's *secret key.* Since in the real world it is quite common for attackers to learn the decryption of ciphertexts, your scheme had better not collapse after they've seen a few decryptions.

Partial oracle access (adaptive chosen ciphertext attacks.) Sometimes the attacker has persistent access to a decryption oracle of sorts, but it isn't a very useful one. For example, a very common type of attack (as we'll see below) is the *format oracle attack.* Here an attacker is allowed to submit ciphertexts to some system, and in response they learn only one piece of information: was the decrypted ciphertext correctly formatted or not?

Concrete examples of format oracle attacks include *padding oracle attacks* where a system decrypts a ciphertext and examines any padding used in the message, sometimes producing a visible error when the padding is invalid. But it can include other, deeper and more application-specific checks as well. For example, a system that accepts encrypted software updates might fail when the software update doesn't install correctly. While the oracle in these situations might only give a small amount of information (usually just a single bit), in some schemes this is enough to gradually learn useful information about a given ciphertext, if we are able to tamper with (or "maul") the ciphertext sufficiently.

Definitions. To capture all of these possible attacks, modern schemes use an extremely strong definition of security, which is known as indistinguishability under adaptive chosen ciphertext attack (IND-CCA.) This game is simply an extension of the IND-CPA game given further above. The main addition here is that the attacker is given access to a decryption oracle: at any point in the game, it can submit any ciphertext it wishes, with the sole exception that it may not ask for the decryption of the challenge ciphertext C^*.[8]

[8] We make this one exception because allowing the attacker to decrypt C^* would make the game trivially

DRAFT

3. **Schnorr signatures (*e.g.,* EdDSA, Ed448).** Some modern systems have begun to implement elliptic-curve variants of the Schnorr signature scheme. Like ECDSA, the resulting signatures are shorter than RSA signatures. However, when used with specific elliptic curves, they can also be quite a bit faster and somewhat more secure than ECDSA. Common implementations include EdDSA and Ed448 [182].

7.2 Schemes in detail

7.2.1 The RSA cryptosystem

Although it's possible to construct public key encryption using the ingredients of Diffie-Hellman as discussed above, the first cryptographers to publicly describe public key encryption came up with a different scheme called RSA. Like Diffie-Hellman, the RSA cryptosystem is inspired by a hard problem from the field of number theory: this time, the problem of *factoring* large integers. This factoring problem is exactly the same one we solve in grade school by hand, when decomposing composite numbers into their prime factors. The initial observation made by Rivest, Shamir and Adleman was that this problem was a good candidate for public-key cryptography, since it appears to be extremely time consuming (*i.e.,* there is no fast classical algorithm for it) when the numbers involved are sufficiently large.

For example, consider the problem of multiplying two long numbers p and q to obtain a product $N = p \cdot q$. Even if these numbers are large, say, 1,000 decimal digits each, this multiplication is easily solved by a modern computer.[10] However, *reversing* the process is not easy. That is, given only N, it's certainly possible to find p and q mathematically: however the best known classical algorithms for doing so might require millions of years on our fastest classical computers.

DRAFT

The "textbook" RSA function. This observation gives the seeds for a cryptosystem, but does not immediately explain how to encipher a message. Rivest, Shamir and Adleman's observation was that they could use a related problem to encipher a message using N as a public key, such that the knowledge of N's factorization (p, q) would enable fast decipherment.

Key generation. To generate an RSA keypair, Bob first selects a pair of large random prime numbers p, q and then multiplies them to form his *public modulus* N. Using his knowledge of p, q, he then computes the Euler totient function $\phi(N) = (p-1)(q-1)$. Finally, he selects a *public* exponent e, which is some positive integer that satisfies the relation $gcd(e, \phi(N)) = 1$. This pair (N, e) becomes Bob's public key, which Bob can give to the whole world. Critically, Bob can use his knowledge of the factorization p, q to select a "secret exponent" d that

[10] Using grade-school multiplication, this will require approximately n^2 simple operations, where n is the number of digits. More sophisticated multiplication algorithms improve on this performance.

satisfies the following relation:

$$e \cdot d \equiv 1 \ mod \ \phi(N)$$

The important thing about this calculation is that Bob can easily find a value d as long as he has knowledge of the totient $\phi(N)$, by using the extended Euclidian algorithm given in §A.1.1. However, finding a value d is difficult if one has only the public key; in fact, finding d is provably as hard as factoring N! Bob can therefore store d as his secret key, which he does not share with anyone.

Encipherment and decipherment. Imagine that Alice wishes to encipher a message to Bob, using his public key (N, e). To do this, she first encodes her message as an integer $M \in \{0, 1, \ldots, N-1\}$. Next, she computes the following modular exponentiation:

$$C \leftarrow M^e \ mod \ N$$

Provided that M and e are large enough (and this is critical, as we discuss below), it is believed that this RSA function is hard to *invert*: that is, given C and Bob's public key, it should be hard for an attacker to find M. Crucially, due to the mathematics of RSA (which we will explain in detail in §7.3.4), the public and secret exponents have a useful property: namely, for any integer M, the value $(M^e)^d \equiv M \ mod \ N$. This means that Bob can easily decipher Alice's ciphertext C by computing:

$$M \leftarrow C^d \ mod \ N$$

DRAFT

RSA encryption in the real world. What we describe above is usually called "textbook RSA", and it should never be used to actually encrypt anything you care about. The reason is that the function above is not a complete encryption scheme. There are several reasons for this:

1. In some cases, the message M can be too small. For example, if both M and e are small enough, then the value M^e (computed without modular arithmetic) might be less than N, which means that no modular reduction is applied. Notice that when this happens, the problem of inverting M^e is simply that of computing an eth root over the integers, which is not hard.

2. Since there is no randomness in the "textbook" RSA function above, a given message M will always encipher to exactly the same ciphertext under Bob's public key. This predictability can create real problems, since an attacker can run a *dictionary attack* on a ciphertext by encrypting many candidate messages $M_1, \ldots, M_N$ and seeing if one of them encrypts to a matching ciphertext.

3. Finally, RSA ciphertexts are highly *malleable*, meaning that an attacker can alter a ciphertext so that the decrypted plaintext changes in meaningful ways. For example,

if $C = M^e \ mod \ N$ is the encryption of a message M, then an attacker can modify the ciphertext by selecting any value A and computing:

$$C' \leftarrow C \cdot A^e \ mod \ N$$

When a receiver decrypts the "mauled" ciphertext C', it will obtain the value $M' \equiv M \cdot A \ mod \ N$. This happens because the following relationship holds:

$$C'^d \equiv (M^e \cdot A^e)^d \equiv M \cdot A \ mod \ N$$

This malleability allows attackers to change the value of encrypted messages in various potentially useful ways. It can also enable sophisticated chosen-ciphertext attacks, as we will discuss in later sections.

To use RSA in the real world, it is necessary to first process your message using a specialized *padding scheme*. These schemes pre-process the message before it is enciphered using the RSA function, and add both randomness and redundant elements that can be verified upon decryption. Once a message is decrypted, the padding scheme can be *inverted* to reveal the original message. We discuss these schemes further below, in §7.2.1.

DRAFT

RSA Encryption padding

To safely encrypt RSA messages, it's necessary to add proper encryption padding to a message before it is enciphered using the RSA function described above. Historically, padding schemes have been a fruitful area for crypto attacks, because until the 1990s we did not know how to build them well. I think it's worth briefly discussing the padding schemes you should *not* use.

Bad padding: PKCS #1v1.5. An obsolete version of RSA padding is supported in TLS 1.2 and earlier versions [232, §7.2]. This padding scheme, called PKCS #1v1.5, must never be used in any new implementations for the reasons we'll discuss further below. The scheme operates by pre-pending a plaintext with a string of random bytes, as well as a small number of delimiter bytes, and then converting the resulting "padded" message into an integer M to be enciphered using the RSA function.

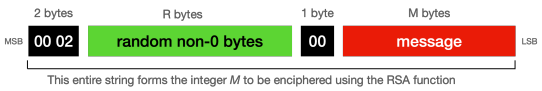

Figure 7.6: The obsolete and broken PKCS #1 v1.5 encryption padding [232, §7.2]. The overall length of the padded string is equal to the length of the RSA modulus in bytes, and the random string must be at least 8 bytes in length.

Upon decryption, receivers are expected to check the structure of the resulting message, and then parse the message portion back out. Unfortunately, this process opens the door to a nasty padding-oracle attack discovered by Daniel Bleichenbacher [71]. If an attacker is merely able to *learn* that the padding checks fail for a given ciphertext (for example, by sending it to be decrypted and receiving a padding error), then it can use this information to slowly decrypt a target ciphertext C. This is done by repeatedly "mauling" C — multiplying it by chosen values — and sending the resulting tens of thousands of mauled ciphertexts in to be checked, until the attacker gradually zeroes in on the original plaintext. This attack proved to be particularly problematic for SSL/TLS implementations, which fix it in a number of kludgy ways. The entire RSA encryption method (and hence PKCS #1v1.5) was entirely removed in TLS 1.3. See §9.2.1 for more on this subject.

OAEP padding. The recommended padding scheme for RSA is called Optimal Asymmetric Encryption Padding (OAEP) [232, §7.1], and it was developed and proven secure[11] by academic cryptographers Mihir Bellare and Phillip Rogaway [46]. Like the previous scheme, OAEP takes in a message that is shorter than the RSA modulus, and concatenates it with a string of random bytes and several structured fields, producing a string that is as long as the RSA modulus. However it now does something very important: OAEP runs the padded string through a custom *Feistel network* that is built using a variable-output-length hash function called a Mask Generation Function (MGF).[12] This permutation "mixes" the padded values together, in a manner that is reversible on decryption. The mixed string can then be interpreted as an integer, and enciphered using the RSA function. When the ciphertext is later decrypted, the Feistel network is easily inverted, and the redundant parts of the padding must be checked for correct structure. The full construction is shown in Figure 7.7.

A neat feature of the OAEP function is that it is extremely sensitive to small modifications in the OAEP-encoded message. If an attacker attempts to maul a ciphertext (perhaps by multiplying it with some chosen value), passing the result backwards through OAEP's hash-based network will almost certainly destroy any useful algebraic relationship that an attacker could exploit. This means that the redundant fields (*i.e.,* the hash field inside of OAEP) will almost certainly fail decryption checks for a mauled ciphertext, and hence the decryptor can safely reject the ciphertext with an error.

Implementation warnings. One small caveat with even OAEP is that it must be implemented carefully. For example, James Manger pointed out that some implementations will check for the presence of the expected 00 byte at the left side of the padding, and will return a distinguishable error when it is not present [211]. In some cases, this information can leak

[11] It should be noted that OAEP is not a total success story. The scheme was an academic result [46] and a notable breakthrough in practice-oriented provable security. However after it was published, Victor Shoup identified flaws in the security proof [294] that allow for specific types of useful malleability, which fortunately are not exploitable when OAEP is used with RSA encryption [133]. Shoup also proposed an improved variant called OAEP+ to fix these problems, but of course nobody uses it.

[12] See Appendix A.3.1.

DRAFT

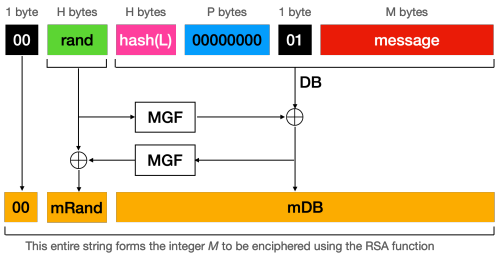

Figure 7.7: The OAEP padding function. The function MGF is just a hash function with an adjustable output length, and is usually built from a standard hash function such as SHA2. The combined length of the top and bottom strings must be equal to the RSA modulus length in bytes. The function hash produces an H-byte output, and L is a "label" that is typically just an empty string. The P-byte *padding string* consists of as many zero bytes as are needed to ensure that the string is long enough (with $P = 0$ being an acceptable length, if no padding is needed.) The hash function is typically implemented using SHA256 or something similar, and the structure of the hash field must be verified upon decryption.

DRAFT

out due to timing channels: some implementations return early when this byte doesn't check out. Even this tiny bit of information enables an attack that is similar to Bleichenbacher's attack, and that can ultimately decrypt OAEP-encoded ciphertexts!

Relationship to factoring. The problem of finding a private exponent d given the public values (N, e) can be shown to be equivalent to factoring: this is because, given (N, e, d) it is possible to recover $\phi(N)$ and hence the factors p, q. The security of the encipherment operation itself is a bit more difficult to connect to factoring. Obviously the security of encipherment *depends* on the hardness of factoring — if an attacker can factor Bob's modulus N, then she can easily find Bob's secret exponent d and decipher messages encrypted to him. However, the problem of recovering M from an RSA ciphertext of the form $C = M^e \ mod \ N$ is a separate task called the *RSA problem*. While theoretically it might be possible to solve the RSA problem without factoring N, nobody has found a faster approach.

RSA Signatures

So far we've discussed only RSA encryption, but of course RSA is often used for authentication as well. This was first noted by Rivest, Shamir and Adelman, who pointed out that the textbook RSA function can be used to make a very weak signature scheme. Here keypairs are generated as in the normal encryption scheme. However, instead of using Bob's secret key to decrypt messages sent by other parties, Bob uses his secret key to *sign* messages that others can verify.

In the simplest (and completely insecure) version of this idea, when Bob wishes to "sign" a message M, he can do so by using his private key d to compute $\sigma \leftarrow M^d \ mod \ N$. The

If you noticed that PSS looks a lot like OAEP, this also isn't a coincidence: both schemes have the same inventors [47]. The basic idea in PSS is to hash the message, and concatenate a random field called the *salt*. This result is then hashed and included in the padded message. To form the remainder of the padded message, the salt (and some fixed padding bytes) is "mixed" with the message hash using an invertible process based on our old friend MGF: this ensures that the resulting message (treated as an integer) is relatively uniform. During signature verification, the mixing process can be reversed to obtain the salt, and then the hashed message can be re-computed from the salt and purported message.

As noted above, a major advantage of PSS is that it is randomized: this makes it more resilient to fault attacks where the signature is computed incorrectly. Since these attacks typically require an attacker to obtain both a valid and faulty signature based on the same (padded) message, the random salt used in PSS tends to produce different padded messages, and hence makes exploiting these attacks challenging. Although this is an advantage, RSA-PSS can also be used in a de-randomized mode where the salt is set to a fixed value (typically all 0 bits): here the scheme is still secure, just with less resilience against fault attacks.

7.2.2 Fault attacks on RSA signing.

7.3 Algorithms in detail

RSA-CRT

7.3.1 Hybrid encryption and KEMs

A problem with many public-key encryption (PKE) schemes is that they tend to be relatively slow when compared to symmetric encryption. This means that we generally don't want to use public-key encryption schemes to directly encrypt any substantial amount of data. Fortunately we don't have to choose! A standard approach to this problem is called *hybrid encryption*.[17] This uses symmetric encryption to encrypt a data payload under some random session key, and then encrypts the session key using public-key encryption. In a bit more detail, a simple hybrid encryption approach works like this:

1. First, generate a random key k for some symmetric encryption scheme.
2. Encrypt k using a PKE scheme, to form a public-key ciphertext C_1.
3. Now encrypt your message payload using symmetric encryption under k to form a second portion of the ciphertext, C_2.
4. The combined "hybrid" ciphertext is $C = (C_1, C_2)$.

This exact origins of this approach are so old that today we call this "folklore", meaning that nobody knows who thought of it first. You can examples of hybrid encryption in

[17] This should not be confused with hybrid PQC schemes, where a classically-secure and post-quantum secure scheme are combined. We'll talk more about those in §7.3.5.

DRAFT

early versions of the Pretty Good Privacy (PGP) software, which literally required hybrid encryption in order to make schemes fast enough to use on 1980s-era personal computers. In practice, hybrid encryption is so common that we don't really think of it as anything special these days: many advertised public-key encryption schemes actually uses hybrid encryption internally.

Key encapsulation mechanisms (KEMs). One observation is that, if you're only using public-key encryption to encrypt random session keys, then you don't need all the power of full public-key encryption anymore. This means that your PKE scheme does not need to be able to encrypt arbitrary messages: it simply needs to be able to generate and encrypt random keys. This can actually simplify the design of a public-key scheme by "building in" the key generation into the encryption algorithm. Schemes with this new algorithm are called *key encapsulation mechanisms* or KEMs.

From a high-level perspective, KEMs possess a Keygen algorithm that works identically to normal public-key encryption. However, in place of the $\mathsf{Encrypt}$ and $\mathsf{Decrypt}$ algorithms, they have key encapsulation and decapsulation algorithms that look like this:

$\mathsf{Encapsulate}(pk) \rightarrow (C, k)$. To encapsulate a session key, any sender can provide a recipient's public key pk. The encapsulation algorithm randomly generates the key k, and outputs it along with a ciphertext C.

$\mathsf{Decapsulate}(sk, C) \rightarrow k$. On input a secret key and encapsulation ciphertext, the decapsulation algorithm outputs the key k (or an error.)

In order to be retain strong security against chosen ciphertext attacks, KEMs must be carefully combined (or *composed*) with a strong authenticated symmetric encryption (AEAD) scheme that uses the resulting key to encrypt the actual payload.[18] This combination can be subtle, in that it requires strong security of both schemes in order for the whole system to remain secure. This is my way of telling you that you should not attempt to plug different schemes together like lego bricks. Instead, I recommend you use a well-studied standard that tells you precisely which schemes to use.

Why are KEMs easier to build? KEMs are a bit easier to build for two reasons. For one thing, they do not require strong *indistinguishability*-based security like a classical PKE scheme: this is because keys are random, and the attacker only "wins" if they can recover the entire session key. For another, it is often much easier to build a scheme that encrypts a high-entropy random value, rather than a chosen message. This is particularly the case in the world of post-quantum encryption.

To give an illustration of how KEMs can be easier to build, consider a scheme based off of Diffie-Hellman. Let g be a generator of a cyclic group $\mathbb{G}$, and let the recipient's public key be $pk_A = g^a$. A simple KEM encapsulation algorithm can be constructed by selecting a random b, computing $C = g^b$, and calculating the session key as $k \leftarrow \mathsf{KDF}(pk_A \| g^b \| pk_A^b)$.

[18] Once upon a time, we called the symmetric schemes Data Encapsulation Mechanisms (or DEMs) [293], but this terminology has mostly been lost to time.

DRAFT

7.3.2 Post-quantum public-key encryption

7.3.3 Diffie-Hellman

PKE from (EC)DH

The problem of building public-key encryption from Diffie-Hellman was first solved by Taher Elgamal [122]. Elgamal's encryption scheme adapted finite-field Diffie-Hellman into a non-interactive encryption scheme in a manner similar to the description in §7.1.2, although the classical Elgamal scheme has some simplifications: for one thing, the Diffie-Hellman shared secret is not fed into a KDF. It is instead multiplied (modulo p) by a message M, which has to be encoded as a group element. The resulting scheme has some subtle flaws, the most significant of which is that it is vulnerable to adaptive chosen-ciphertext attacks (§7.3.7). While Elgamal is sometimes used as a broad term for Diffie-Hellman cryptosystems, "classical" Elgamal should never be used in production systems.

ECIES. In the early 2000s, a number of cryptographers began standardizing public-key encryption schemes based on (EC)DH [5, 40, 293]. The resulting schemes use hybrid encryption: that is, they divide the encryption scheme into two parts. A public-key *key encapsulation mechanism* (KEM) encrypts a secret session key to a recipient's public key. The second portion, or *data encapsulation mechanism* (DEM) uses the secret key to encrypt a message. In practice a DEM is simply a different name for a modern authenticated encryption scheme, and so that terminology has fallen out of favor. The resulting ISO standard produced two complete schemes, the Diffie-Hellman Integrated Encryption Scheme (DHIES) and the Elliptic Curve Integrated Encryption Standard (ECIES) [172]. Here we will focus on ECIES, since it is more efficient.

The ECIES standard relies on an elliptic curve subgroup $\mathbb{G}$ with a generator P, an authenticated encryption scheme $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$, and a function encode that encodes EC points to strings. Public keys are formed by selecting a scalar x in the range $[0, \mathsf{order}(\mathbb{G}))$ and setting $pk = xP$, $sk = x$. To encrypt a variable-length message $M \in \{0,1\}^*$ under pk, the encryption algorithm performs the following steps:

1. Verify that $pk \in \mathbb{G}$ and that it is not the identity element.
2. Sample a random scalar $r \in [0, \mathsf{order}(\mathbb{G}))$.
3. Use scalar multiplication to compute $c_1 \leftarrow rP$ and $S \leftarrow r \cdot pk$.
4. Set $k \leftarrow \mathsf{KDF}(\mathsf{encode}(c_1) \| \mathsf{encode}(S))$
5. Encrypt the message M as $c_2 \leftarrow \mathsf{AE.Enc}_k(M)$.
6. Output the ciphertext $C = c_1, c_2$.

Decryption reverses the process, first parsing C back into c_1, c_2 and then calculating $S \leftarrow sk \cdot c_1$, ad re-computing $k \leftarrow \mathsf{KDF}(c_1 \| S)$. The message (or an error) is obtained by computing $M \leftarrow \mathsf{AE.Dec}_k(c_2)$.

DRAFT

Implementations of ECIES. Although relatively few systems explicitly cite that they are using ECIES, variations of this algorithm pop up in many places. For example, the `crypto_box` algorithm of the NaCl and `libsodium` libraries uses a version of ECIES with Curve25519, ChaCha for the KDF, and ChaCha20-Poly1305 as the symmetric encryption algorithm. NIST also standardizes the public-key portion of this scheme under the creative name "(Cofactor) One-Pass Diffie-Hellman" [32, §6.2.2.2], which means it can be FIPS-certified if it's used with NIST-approved algorithms.

Security of ECIES. A useful feature of ECIES is that, implemented with appropriate components, it is secure against adaptive chosen-ciphertext attacks. This definition is explained in §7.1.2, but briefly: it ensures among other things that the cipher cannot be *malleated* (changed) while still allowing the modified ciphertext to decrypt correctly. The scheme features a security proof in the random oracle model, under an assumption called Gap CDH (§4.4.3).

7.3.4 RSA

Attacks on RSA

Bleichenbacher attacks on RSA padding

7.3.5 Post-quantum algorithms: ML-KEM and HQC

In the 2010s, a coalition of industry, scientists and government standards agencies began to consider new algorithms that could resist future quantum computers. These algorithms, broadly referred to as *post-quantum cryptography* (PQC), had already been a subject of research for several years. However, in 2016, NIST accelerated the process by announcing a competition to select a series of new PQC government standards. In 2024 and 2025, the first five standards were announced: two for public-key encryption (ML-KEM and HQC), and three for signatures (Dilithium, SPHINCS+ and Falcon.)

ML-KEM

ML-KEM stands for *Module Lattice Key Encapsulation Mechanism.* Originally given the name CRYSTALS-Kyber [77] by its designers, ML-KEM is based on a hard problem known as Module Learning with Errors (MLWE) in the set of problems related to lattice-based cryptography. A key feature of these problems is that there exist no known quantum algorithms that can solve them. This means that, while ML-KEM can be executed on a classical computer, even quantum computers will (hopefully) not be able to break it.

As the name implies, ML-KEM is a key encapsulation mechanism, which means that it is designed to securely encrypt (session) keys to a recipient. These session keys can be used to encrypt a message payload using symmetric encryption, which makes the algorithm optimal for hybrid encryption.

DRAFT