Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter 7, Public-key cryptography

§7.1.3 Digital signatures; RSA signatures and signature padding; §7.3.8 Elliptic curve cryptography and popular curves (book pages 139–140, 145–147, 154–156)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
Lecture 13, Wed Oct 14: digital signatures and elliptic curves

`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.

This definition has some surprising implications: note that it implies that schemes cannot admit any useful form of ciphertext *malleability*: after all, if an attacker could directly maul the ciphertext C^* into the encryption of a related message, they could submit the mauled ciphertext for decryption and potentially learn which message was encrypted inside of C^*. Thus, achieving chosen ciphertext security usually requires that decryptors should be able to check the messages they receive, and reject anything that has been mauled.

7.1.3 Digital signatures

While public-key encryption and key exchange handle the very important task of assuring confidentiality, this is only one of the two main guarantees we require in order to communicate securely. The second is the *authenticity* of communications. In earlier sections we discussed MACs, which allow us to ensure the authenticity of arbitrary messages provided that both sender and recipient share a symmetric key. The equivalent tool in the public-key world is called a *digital signature*.

Like a public-key encryption scheme, a digital signature scheme involves a public verification key vk that can be given out to all parties, as well as a secret key sk that must be held by the signing party itself. A digital signature scheme can be expressed in terms of several algorithms as follows:

1. $\mathsf{Keygen}(1^\lambda) \rightarrow (vk, sk)$. This algorithm is executed by the signing party. It takes in the desired key strength λ and outputs a *keypair* consisting of a public verification key vk and secret key sk.
2. $\mathsf{Sign}(sk, M) \rightarrow \sigma$. To sign a message M, the signer specifies the signing key sk as well as a message. The result is a signature σ.
3. $\mathsf{Verify}(vk, M, \sigma) \rightarrow \{\mathsf{True}, \mathsf{False}\}$. The verifier applies this algorithm on input the verification key vk, the message M and signature σ and obtains True if the signature verifies correctly, or False otherwise.

DRAFT

Security of digital signatures. Before we get any deeper into the construction of digital signatures, it's important to specify exactly what properties we require of a digital signature scheme. For a signature scheme to be secure, it must possess a strong property that we call *strong existential unforgeability under chosen message attack*, or $\mathsf{SUF\text{-}CMA}$. In plain language, unforgeability means that no attacker (who does not possess the signing key) should be able to create a new signature that they have not received from the legitimate signer. The "chosen message attack" part means that the attacker can request as many signatures as they want, on messages of the attacker's choice, and they must still not be able to make a new forgery.[9]

easy to win.

[9] A "forgery" here means one of two things: either they have produced a valid signature on a new message M' that they didn't obtain a signature for, *or* they have made a new and valid *signature* on a message

Digital signatures are strictly more powerful than MACs, since anyone can verify a digital signature without knowing the secret key. This has some deeper benefits: since only one party needs to know the secret signing key, this means that only one party should have been able to create a signature (assuming the signing key wasn't stolen)! This is very different than MACs, where both the sender and receiver can create a valid MAC on any message, meaning that you can't be certain which one of them created the MAC. This added guarantee for signatures is called *non-repudiability* and it's usually considered a good thing, though sometimes it can be undesirable. For example, some messaging schemes leverage the fact that either party in a conversation could have created a MAC to offer a form of "deniability" — in short, the sender can always shrug and claim that a given message was actually authored by the other party in the conversation.

Certificates. One of the most powerful applications of digital signature schemes is that they make it possible for parties to attest to the value of other parties' public keys. This is typically done using *digital certificates.* A certificate is simply a file that contains various pieces of information: typically a party's name (or website address), combined with their public key. The file is then signed by some trusted authority, and anyone who knows the authority's public key can verify the contents of the certificate. An entire *public-key infrastructure* exists for verifying the identity of public keys: trusted parties called Certificate Authorities (CAs) author certificates for individuals, websites or other CAs. Anyone who trusts the Certificate Authority (and has its public key) can then verify the authenticity of a public key received from any party on the Internet.

DRAFT

Concrete signing algorithms. There are many digital signature schemes in use on the Internet. Some common choices include:

1. **RSA signatures.** Although I don't recommend the use of RSA in new systems, for legacy reasons RSA signing is still very common in web public-key infrastructure. The main disadvantage of RSA signatures is that they are relatively large, with signatures and public key sizes ranging from 1,024 bits (bare minimum) to 4,096 bits or longer.

2. **Elliptic Curve Digital Signature Algorithm (ECDSA).** The elliptic-curve Digital Signature Algorithm (ECDSA) was standardized by NIST in 2000 [**?**], and has since been adopted all over the Internet. While the algorithm itself is kind of obnoxious (see further below for a discussion), the public keys and signatures themselves are shorter than RSA signatures. This, combined with NIST certification, has made ECSDA one of the most popular signing algorithms across the Internet, and in cryptocurrency systems.

that they did have a signature for. This means that even "mauling" a valid signature to produce a slightly different (but still valid) signature is considered a break against such a scheme!

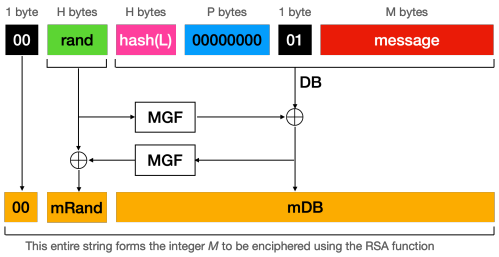

Figure 7.7: The OAEP padding function. The function MGF is just a hash function with an adjustable output length, and is usually built from a standard hash function such as SHA2. The combined length of the top and bottom strings must be equal to the RSA modulus length in bytes. The function hash produces 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

resulting σ forms a signature on the message M. Alice can then verify the signature using Bob's public key (N, e), by checking that:

$$M \stackrel{?}{=} \sigma^e \ mod \ N$$

While this provides a nice intuition, using RSA this way does not actually produce a secure signature scheme! The problem here is the *malleability* of RSA. An attacker who intercepts a message and its corresponding signature (M, σ) can maul both to produce a signature on a related new message. For example, given some integer A, they can compute:

$$M' \leftarrow M \cdot A^e \ mod \ N, \qquad \sigma' \leftarrow \sigma \cdot A \ mod \ N$$

The resulting "signature" σ' will now be a valid signature on M'.[13] A more useful variant of this attack is the observation that one can multiply any two signed messages M_1, M_2 together to form $M' = M_1 \cdot M_2$ and the product of their signatures $\sigma' = \sigma_1 \cdot \sigma_2$ will also be a valid signature on M'![14] These attacks may or may not be useful in real life, but they are more than enough to break the SUF-CMA security of the scheme. In practice, they're probably bad enough to make "textbook" RSA signatures completely unsafe for real-world applications.

DRAFT

Hashing and signature padding

The solution to RSA signatures' malleability problem has two pieces. First, signers must not sign the message M directly: instead they should sign a cryptographic *hash* of the message, $\hat{M} = H(M)$. When using standard cryptographic hash functions, this breaks the simple algebraic relationship that makes malleability attacks possible. Still, hashing by itself is not sufficient. As a second element of protection, the hashed message should be carefully padded to form an integer that is as long as the RSA modulus, and to ensure that the hash is at a known location within the resulting string. Two different padding schemes exist for this purpose, and I will confess that I hate them both.

PKCS #1v1.5 signature padding. The most popular standardized signature padding is a legacy of the 1990s. It involves hashing a message M using some cryptographic hash function, and then padding the result with a certain number of *non-random* fixed bytes. The resulting string is then treated as an integer $\hat{M}$ and sent to the RSA (signing) function to produce a signature: $\sigma \leftarrow \hat{M}^d \ mod \ N$. The scheme is shown in Figure 7.8.

If you've noticed the similarity between this padding scheme and the *insecure* encryption PKCS #1v1.5 padding scheme mentioned in the previous section, you're not imagining anything. The good news is that, unlike its encryption-padding cousin, the signature padding

[13] This works because $\sigma' = M^d \cdot A$ and $\sigma'^e = (M^d \cdot A)^e = M \cdot A^e = M'$.

[14] This holds because if $\sigma_1 \equiv M_1^d \ mod \ N$ and $\sigma_2 \equiv M_2^d \ mod \ N$, then $\sigma_1 \cdot \sigma_2 \equiv (M_1 \cdot M_2)^d \ mod \ N$.



Figure 7.8: The RSA PKCS #1 v1.5 signature padding scheme. The padding string contains as many `FF` bytes as needed to pad the full string to the length of the RSA modulus. Note that the hash digest (right) also contains an identifier field that indicates which hash function is being used; this is not shown in the diagram.

variant is not known to be actively broken.[15] Based on that faint praise, this scheme has more or less become the de-facto standard for RSA signing across most of the Internet, including most X.509 certificates.

While there are no known attacks on properly-implemented PKCS #1v1.5 signature padding, there are some *implementation* footguns. Some verifier implementations will first compute $P \leftarrow \sigma^e \ mod \ N$ and will then attempt to *parse* the padded string P from left to right in order to find the hash digest. The problem here is that these implementations can become tricked into "finding" the hash digest at some earlier location in the string than where it actually belongs. Clever attackers can exploit this in some cases, usually when small RSA exponents are used [128]. To prevent this attack, verifiers should instead construct a candidate string P' that contains both the full expected padding result (including the digest of the purported signed message), and they should then compare the two strings to determine if $P \stackrel{?}{=} P'$.[16] A second concern with non-randomized PKCS #1v1.5 signatures is the possibility of *fault attacks* when something goes wrong during RSA signing, such as an error in the arithmetic or a misplaced bit. See §7.2.2 below for more on this concern.

DRAFT

PSS padding. An alternative padding scheme called Probabilistic Signature Scheme (PSS) [**?**] is generally viewed as better than PKCS #1v1.5 padding, for two reasons: (1) since PSS is *randomized*, it is more resilient to fault attacks, and (2) it has a security proof. Unfortunately, PSS is not as widely deployed as PKCS #1v1.5, largely due to path-dependence and the difficulty of upgrading legacy systems. This is particularly remarkable, given that PSS is thirty years old!

[15] As a note, "not known to be insecure" does not mean great. The question of whether the scheme is truly secure remains open. Some clever papers have gone so far as to try to *prove* that PKCS #1v1.5 signature padding is safe, but were only able to do so for weird variants of RSA that use three prime factors to form the public modulus [177]. The authors conjecture that this result ought to hold in normal two-prime RSA, but we don't know how to evaluate that conjecture. This is uncertainty is why I hate this signature padding; too bad it's a standard.

[16] Although there are no secrets in the strings being compared, be sure to do it correctly! At least one fun implementation vulnerability occurred when Nintendo's Wii devices used C's `strncmp()` function to compare these two strings, instead of using `memcmp()` [91]. Since C string comparison ends when the first `00` byte is encountered, this made it pretty easy to forge signatures.

group messaging systems. The downside of this approach is that a single party now has total control over the key that the group will use, meaning that there is no notion of "contributory behavior" where each party must contribute to the value, nor do group members always receive post-compromise security. This may or may not be a concern. Other systems, such as the Messaging Layer Security (MLS) standard, use more complicated group key agreement protocols like TreeKEM [62]. These protocols provide more assurance, but at a large cost in terms of complexity.

Post-quantum key exchange. The challenge with ML-KEM is that (1) it features much larger public keys than ECDH, and (2) it is a key encapsulation scheme (i.e., a public key encryption scheme that encrypts symmetric keys) rather than being a full forward-secure key exchange algorithm.[20] This makes post-quantum key exchange a bit more complicated than simple ECDH. See §7.3.5 for more details.

MITM attacks

General attacks on public-key encryption

7.3.8 Elliptic Curve Cryptography

Attacks and security considerations.

Popular curves

Many curves are available to implementers, but a few are more widely used. It seems weird to talk about "popular" elliptic curves, but in fact many cryptographic engineers have strong aesthetic attachments to specific curves.

The NIST P-curves. Historically, some of the most common curves in deployed systems are the NIST pseudorandom, or "P-curves", a set of curves defined using the short Weierstrass equation $y^2 = x^3 + ax + b$ over a prime field $\mathbb{F}_p$ of various sizes [88]. These curves were generated in the 1990s by cryptographers from the NSA, and they come in several field sizes, where each indicates the length of p in bits: these include P-256, P-384 and P-521. Using today's best known attacks, each curve is believed to provide (non-quantum) discrete logarithm security equal to approximately half the field size, and so for example P-256 provides approximately 128 bits of security.[21]

[20] It is possible to build a full forward-secure key exchange algorithm from ML-KEM, but it requires that both sides generate and exchange "ephemeral" ML-KEM public keys for each key agreement, then use them one time for encryption. This is much more costly than just using ECDH in terms of bandwidth.

[21] As of this writing, the NSA has deprecated P-256 for use in Suite B cryptography, instead recommending P-384 as the minimal acceptable curve size [141].

DRAFT

The P-curves are popular because they pre-date the other curve options, and because historically they were the only curves approved for use in FIPS-certified products (this changed in 2023 when NIST added new curves to the approved algorithms list.) However, many people criticize the P-curves for a variety of reasons. One reasonable argument is that it is somewhat challenging to write secure code for P-curve operations that is both efficient and hardened against side-channel leakage. OpenSSL, for example, spent a great deal of time optimizing their curve implementations of P-256 and P-384 for these goals [222]. Standards such as Curve25519 use a design that is fundamentally more amenable to fast calculation and side-channel resilience.

Do the NIST P-curves contain an NSA backdoor? The short answer to this question is: probably not. Nonetheless, some serious cryptographers have at least discussed the possibility [54]. The main concern here is that the parameter generation process for the P-curves was conducted internally by NSA employees, which raises the possibility that the agency could have selected the parameters (a, b, p) in ways that enable some clever attack, one that still remains unknown to the public research community.

In principle the generation process for the P-curves is supposed to guard against this possibility. Each curve begins with a prime p and a small value of a (typically $a = -3$) as well as some *seed*, which is hashed through SHA-1 to form the value b. The resulting curve is then evaluated for security against known attacks, and the process is repeated with a new seed if it fails. The use of hashing is supposed to constrain the choice of b, so that outsiders can be certain it was not chosen maliciously. The problem with this approach is that the *seeds* chosen by the NSA are not recognizable values like π or well-known pieces of English text: they are simply opaque random numbers. Thus it is at least *possible* that the NSA could have brute-force tested millions of seeds in order to find curves that are vulnerable to some secret attack.[22]

A strong rebuttal to this hypothesis is that the NSA itself designed these curves to protect U.S. classified material for several decades, and (to employ a vulgar phrase) one does not defecate where one eats. A more compelling scientific argument was made by Neal Koblitz and Alfred Menezes in 2016 [190]. These scientists point out that, given the NSA's computing power in the 1990s, it is highly unlikely that the agency could have tested more than 2^{48} curves. If this search was sufficient to find a curve that is vulnerable to the NSA's hypothetical attack, that would imply that a vast number of possible curves would have to be vulnerable: for P-256, around 2^{209} out of 2^{257}, which vastly dwarfs the fraction of curves affected by any other known attack. The existence of such a broad vulnerability would make the entire field of ECC non-viable. In other words, if one believes that such an attack exists, the solution is not to choose different curves, but to abandon elliptic curve cryptography altogether. For what it's worth, I find this argument to be fairly convincing.

[22] Unverifiable rumor has it that Jerry Solinas of the NSA chose the seeds by hashing specific pieces of English text through SHA1, but the original text has been lost to time. Filippo Valsorda has offered a $12,000 bounty for any volunteer who can find the correct pre-images [315]. As of this writing, nobody has claimed the prize.

DRAFT

Curve25519 and Curve448. Two popular alternatives to the NIST P-curves are Curve25519 and Curve448. Both curves are included in the TLS v1.3 standard [274], and have also been approved by NIST for use with FIPS-certified products since 2023 [88].

Curve25519. Curve25519 is a purpose-built curve designed by Daniel J. Bernstein in the early 2000s [52]. The curve uses the equation $y^2 = x^3 + 486662x^2 + x$ over a prime field $\mathbb{F}_p$, and is optimized for efficiency on modern CPUs. To this end, it makes some opinionated choices, among which are: (1) the prime $p = 2^{255} - 19$ is set extremely close to a power of 2 in order to optimize calculations, and (2) for key exchange (called X25519) its equation and arithmetic use the Montgomery form, which ensures that public keys need only consist of a single 256-bit x coordinate, rather than a pair (x, y) or a different compressed representation.[23] Applications such as Schnorr-style signatures can use the birationally equivalent twisted Edwards 25519 curve, which obtains the same security level and similar performance [55].

Curve448. A limitation of Curve25519 is its group order, which limits discrete logarithm security to approximately 128 bits. To address this, Mike Hamburg designed Curve448 [154]. Curve448 is an untwisted Edwards curve with equation $y^2 + x^2 = 1 - 39081x^2y^2$ over the field $\mathbb{F}_p$ where $p = 2^{448} - 2^{224} - 1$. This field size is sometimes called a "Goldilocks" prime, because it defines a golden ratio, which in turn enables fast Karatsuba multiplication. Curve448 (sometimes Curve448-Goldilocks, or Ed448) provides up to 224 bits of discrete logarithm security.

Decaf and Ristretto. One pitfall of both Curve448 and Curve25519 is that the resulting groups have composite order $c \cdot q$, rather than prime order as in the NIST P-curves. This is due to the presence of a *co-factor* in the group order, of size $c = 4$ for Curve448, and $c = 8$ for Curve25519. This is not a problem for standard EC Diffie-Hellman, but it can throw a monkey wrench into some fancier cryptographic protocols that are designed for prime-order groups. To emulate a prime-order group in this setting, there are techniques such as Decaf [153] for Curve448, and a library called Ristretto255 [155] does the same for Curve25519.

Rigidity. A major argument used to promote both Curve25519 and Curve448 is their use of "rigid" parameter sets. This is mostly a response to the concerns about NIST's P-curves mentioned in the previous section. Both of these curves use parameter sets that are relatively constrained. For example, Curve25519's constant 486662 and Curve448's -39081 are the smallest constants that satisfy all design requirements. It is not clear how well rigidity actually works to constrain designers, since there is still a degree of flexibility in setting the design requirements. Whether you care about this or not is largely a function of how concerned you are about secret attacks known only to curve designers.

[23] This form of the curve is optimized for EC Diffie-Hellman key exchanges (sometimes called X25519), where the only operation needed is scalar point multiplication. For other applications, the Edwards curve is used.

DRAFT