Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter 9, Cryptographic protocols

§9.1 Secure communications protocols (design principles); §9.2 Network encryption protocols (intro) (book pages 167–180)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
Lecture 15, Wed Oct 21: TLS design and PKI — recommended background

`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.

Chapter 9

Cryptographic protocols

DRAFT

So far this overview has covered many of the basic ingredients of cryptography: ciphers, MACs, public-key encryption schemes, and signatures. Often these ingredients are called *primitives*. To execute real tasks, these primitives are almost always used as the building blocks for a more complex structure, which is known as a cryptographic protocol.

If we go back to the days of classical cryptography, the primitive *was* the protocol. To send a message to another user, one party would simply encipher the message and transmit it to the other user via the mail or messenger. Modern cryptographic protocols are vastly more complex: they often involve multiple rounds of communication, combining different ingredients such as negotiation, key exchange, digital signatures, and symmetric encryption into a single interoperable standard.

Not only are cryptographic protocols more sophisticated, the adversaries they face are more powerful. An active adversary who can tamper with messages in a simple encryption scheme may succeed in changing the data received. An adversary who can do the same within a protocol might succeed in altering the encryption parameters, or even *disabling encryption entirely.* Moreover, these adversaries often have more opportunity to meddle than historical adversaries did: the widespread use of protocols like TCP/IP and 802.11 WiFi, which route sensitive data via untrusted parties, makes data tampering relatively easy. Today's adversary simply needs to impersonate (or compromise) the WiFi access point at your local coffee shop to gain control over every packet sent from or received by your machine.

A few common protocols. There are many standard protocols used to secure communications between devices, and hundreds of proprietary ones. The most widely used of these is the Transport Layer Security (TLS) protocol, which is used to encrypt nearly all website connections, as well as most email and application-related connections. TLS is arguably the most important protocol on the Internet, and a failure in TLS would expose most communications. But TLS is not the only game in Internet encryption: for bulk traffic encryption (such as Virtual Private Networks), encryption protocols such as Wireguard and IPsec operate at lower layers of the network stack. Wireless systems often implement their own

low-level network protocols: Bluetooth, WiFi and cellular networks each implement protocols to encrypt vulnerable data as it flows over the air. Finally, at a higher level, many applications, ranging from encrypted messaging applications to cloud backup systems to blockchain-based payment systems, each implement their own standard (and non-standard) encryption protocols.

TLS

To briefly illustrate the complexity of even a "simple" encryption protocol, consider the TLS protocol. TLS encryption is designed to protect TCP/IP-based network connections made between two devices, typically a client and a server. To the applications that initiate these connections, the encryption protocol is mostly invisible. However, under the hood, TLS involves a complex dance: a typical connection involves a *negotiation* in which the parties determine what encryption algorithms to use, a *key exchange* in which the parties use public-key cryptography to agree on a key, *digital signing* to ensure that the messages from one or both parties are authentic, and finally *record encryption* to symmetrically encrypt each byte flowing between the parties.

The complexity of the TLS handshake provides a huge attack surface for adversaries to exploit, and exploit it they have: at least in research papers! As just one example, in the mid-2010s, researchers discovered that TLS clients could be *downgraded* to use older, insecure "export-grade" encryption schemes that are easily broken [**?**, **?**]. These export systems were an ancient legacy of the 1990s. when US law prevented the export of strong cryptography — and yet they had been forgotten inside of the TLS protocol for decades after the law was changed. While these broken schemes are no longer present in the latest version of TLS, similar or worse flaws have been discovered in many previous versions.[1]

The process of designing and validating protocols like TLS is error-prone, and unfortunately, tends towards a great deal of manual effort. While complete breaks of cryptographic primitives — modern ciphers, MACs and digital signatures — are fortunately quite rare, protocol bugs remain a major threat to Internet security. We discuss many of these in Chapter 9.

Apple AirTags

To give an example of a very different style of protocol, consider the protocol used by Apple's AirTags. AirTags are a simple location tracker device that can be attached to physical objects such as a piece of luggage. Each tag contains a small transmitter that repeatedly sends a signal via Bluetooth, that can be received by its owner's phone — or by *any* compatible phone in the area. If you lose your AirTag in a bar, it can broadcast an identifier that will

[1] The current version of TLS at the time of this writing is TLS 1.3, and it uses a digital signature to authenticate the handshake. This change was designed to address the attacks described above.

DRAFT

be received by a stranger's phone. That "volunteer" phone will then report both the tag identifier and its current location up to a central server run by Apple.

Of course, a problem with the AirTag protocol is that it poses a major privacy risk: if an AirTag consistently broadcasts a fixed, unchanging serial number, then anyone with a Bluetooth receiver can recognize your presence. It does not matter if the receiver can immediately translate that signal into your name, since there are many commercial advertising systems that are designed to narrow down the identify of a customer by using their movements and tying them to this sort of identifier. In addition, AirTags create the possibility that *Apple* will track its users, by monitoring AirTag reports sent to the server.

To address these concerns, Apple designed a sophisticated protocol that periodically changes the identifier emitted by the AirTag. This identifier is determined using a symmetric *pseudorandom function* that takes in a tag-specific secret key and periodically "rotates" the identifier, so a tracker will see what appear to be unrelated identifiers that change every few minutes. To ensure that location data is protected, each AirTag formats its identifier as the public key for a public-key encryption scheme. When a volunteer phone receives the identifier, it can encrypt its own GPS coordinates directly to this public key and upload the data to Apple. If an individual loses their tag, they can keep a copy of the tag's secret key in another device: this allows them to re-calculate each of the tag's identifiers, and query Apple's server to see if it one of the identifiers has been reported by a volunteer. They can then derive the appropriate secret key to decrypt the ciphertext containing the tag's GPS coordinates.

In this way, Apple never sees a fixed identifier or the GPS coordinates of any AirTag.

DRAFT

9.1 Secure Communications Protocols

Secure communication (or *transport*) protocols are designed to establish keys and transmit data between two endpoints. While this represents only a subset of the giant world of protocols, secure communications is such a common use-case that it covers a large number of the protocols we use in practice.

As a general warning: this section is intended to help readers understand how real-world protocols work. It is *not* intended to encourage you to design your own new protocol! If you need to send data securely, you probably want to use TLS. In the event that TLS genuinely doesn't work for you, try to find a different and well-reviewed protocol that suits your use-case. I strongly discourage you from designing your own security protocols, and if you decide to do this anyway, don't tell anyone you read this book!

Threat models. The main concern in secure communications is that communications flow over an insecure channel, where an adversary may attempt to eavesdrop or tamper with messages. In general when we design modern protocols, we tend to assume a powerful MITM attacker that can read, block, replay, modify, or replace any messages at will (see

Figure 9.1.)[2]

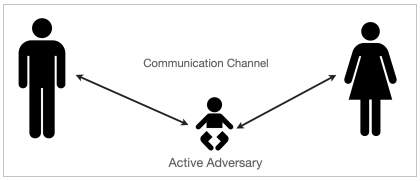

Figure 9.1: Illustration of a communication channel between two honest parties, with an active adversary routing messages between the two.

Some protocols also make even stronger assumptions about the attacker. For example, protocol designers sometimes consider the possibility that an attacker will be able to temporarily compromise one of the participants, with the goal of stealing secret keys and decrypting past or future communications. Many protocols make a strong effort to be resilient in the face of these attacks, although there are natural limits to how far this can go.

State machine and protocol beliefs. The ideal protocol would only involve one party: there, everyone would naturally agree on everything. The whole problem with interactive protocols is that there are too many people. Ensuring that multiple participants agree on the correct state is challenging enough in ideal conditions, and security protocols operate under the worst. Most protocol attacks involve disrupting or altering communications between participants, so that they no longer possess *accurate beliefs about the other parties' state.*

In an abstract sense, one way to think about a protocol is to imagine that each participant is running its own giant state machine, where every incoming message causes the receiver to (potentially) switch to a new state. A protocol with two parties is an even bigger machine, where the state of the larger machine is defined by the joint state of the two parties. A well-functioning protocol will traverse a small and well-defined set of states for that larger machine. The adversary's goal is to visit many exciting new ones.

Real state machines. Most interactive protocols do, in fact, have a real state machine that keeps track of where each party is in the protocol execution. However this captures only a part of the actual state. For example, TLS has at least a dozen options that can be requested by the parties. If each combination of options represented an explicit "state" in a real finite-state machine, the complexity of implementations would explode. So instead, protocol designers resort to careful checks as they move from place to place. For a (confusing) example of what this can look like, take a look at an absurd diagram from the most recent Wi-Fi standard in Figure 9.2!

To move from the abstract to the concrete: consider a protocol in which Alice sends a list of encryption algorithms she supports, and Bob picks one off the menu (or aborts, if

[2]In some academic papers this form of attacker is referred to as a Dolev-Yao adversary [109].

DRAFT

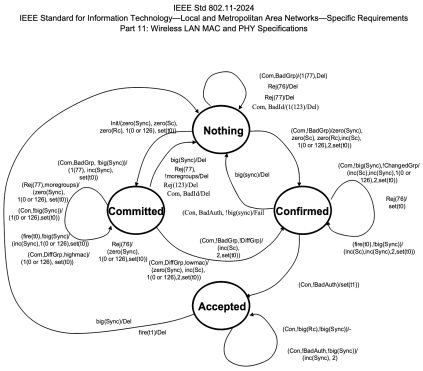

Figure 9.2: A state machine for the SAE protocol, which is part of the 802.11:2024 standard. Source: [167, §12.4.8].

DRAFT

none are acceptable.) One way to picture this system is that each combination of algorithms from Alice moves the Bob into a different state. From that state, there is a correct choice he should issue. A clever attacker might modify Alice's message so that Bob ends up in the wrong state, and thus the choice he makes will be different than it should be. Much of the goal of a well-defined protocol is to ensure that this can never happen: that both Alice and Bob agree on exactly what state the other should be in, and what it *thinks* it is in. If disagreements occur, then the protocol can simply terminate. This reduces the best possible attack to the worst one, *i.e.*, the attacker can simply block the protocol from proceeding at all.[3]

Negotation and cryptographic agility

Secure communications protocols don't usually lead with cryptography. In the real world, the first phases of these protocols frequently perform a non-cryptographic *negotiation* stage that allows the peers to agree on elements such as the protocol version, features, or specific algorithms they wish to use. There are usually two different approaches to negotiation:

1. **Versioning.** No protocol has every been perfect out of the gate, and for various reasons (including the existence of exploitable vulnerabilities) it is usually necessary to deploy new versions. Many protocol designers recognize this, and provide a means for peers to negotiate the *version* of the protocol they will use. For example, TLS

[3] In general, attackers can almost always block the execution of a protocol if they have sufficient control over the network. Protocols that try to fight this normally need to place strong restrictions on the attacker, such as limiting how many parties they can corrupt, or which messages they can tamper with.

peers can in principle negotiate anything between TLS v1.0 and TLS v1.3, depending on what version the peers support. Providing a means to support versioning makes protocol upgrades significantly easier, although it also opens the door for *downgrade attacks* in which an attacker forces peers to negotiate a different version of the protocol than they otherwise would.

2. **Agility.** While versioning provides a coarse-grained mechanism to negotiate protocol features, some protocols also support fine-grained configuration options. These will often include the specific cryptographic parameters or algorithms to use (*e.g.,* ciphersuites), but can also include options such as mutual authentication and features like session resumption.

How much agility is the right amount? There has been a long-running debate in the engineering community about whether simple "protocol versioning" or fine-grained agility is the correct way to build a protocol. IPsec, SSL and TLS traditionally erred towards extreme agility, to the point where one standard actually represents an entire zoo full of protocol variants.[4] The sheer number of TLS options helped to fuel a run of vulnerability discoveries in the 2010s. In response, many cryptographic engineers have argued for less agility within protocols; both TLS v1.3 and Wireguard represent varying moves in that direction.

The other end of the spectrum removes all options, and simply uses versioning to determine which features are supported. A downside of the versioning-only approach, however, is that the addition of any new cryptographic algorithm will require a full protocol version upgrade, with all the overhead that brings. These considerations are particularly relevant today, since government and industry are in the midst of a transition to using post-quantum cryptographic algorithms (see §7.3.5.)

DRAFT

Protecting the authenticity of the negotiation. Regardless of what happens during negotiation, a difficulty is that this phase of the protocol often runs before peers have established shared cryptographic keys to authenticate their messages with. This raises the possibility that a clever MITM attacker might tamper with the negotiation messages, forcing a "downgrade" from what the peers would have selected naturally. Protocol designers have essentially two options to deal with this:

1. **Use of digital signatures (or PSKs).** If both sides possess the other peer's public key (or it can be delivered in a trustworthy way, such as within a certificate), then peers can *digitally sign* their negotiation messages.[5] This prevents most tampering,

[4] Even worse, it was something of a nightmare for implementers, who needed to write complex state machines that re-used code for many different agility features and versions, resulting in the implicit creation of still further chimera protocol variants [61].

[5] If a trustworthy shared secret, called a pre-shared key, is available, then the peers can use this secret to symmetrically authenticate the messages instead.

except for the simple replay of older messages.[6] In general, this approach makes the negotiation much more secure. In TLS v1.3, for example, all handshake messages are signed by (at least) the server to ensure integrity.

This approach does not work in all protocols, however. In many cases some peers do not have long-term public keys: for example, clients in most TLS connections typically do not use certificates to authenticate. A second downside is that this approach requires peers to send relatively bulky signatures, and produces added computation during the handshake.

2. **Post-hoc authentication.** A second approach is for peers to conduct at least part of the negotiation in unauthenticated plaintext. Then, once a protocol handshake has completed and shared key material is available, the peers can *retrospectively* verify that they agree on their respective "views" of the negotiation transcript, for example by exchanging MACs of the negotiation messages they sent and received. This approach is used in most pre-v1.3 TLS connections, in Wireguard, and even in part during TLS v1.3 connections (*e.g.,* in the common case where the client does not have a certified public key.)

   The danger of post-hoc authentication is that the key agreement algorithm itself sits downstream of negotiation. Thus an attacker could potentially tamper with negotiation in ways that result in an insecure key exchange: for example, by downgrading the peers to use a broken cryptographic algorithm. In the worst case, an attacker might be able to break the key exchange, obtain the shared key, and then forge any post-hoc authentication messages to cover its tracks! In a more general sense, this approach only works if *every possible* negotiated key exchange is secure, since the existence of one insecure option will unravel the whole protocol.[7] This is a very strong assumption to make, particularly if your protocol is designed to be extensible.

DRAFT

Out in the real world, protocols use various combinations of the two approaches. For example, TLSv1.3 in its most popular single-sided authentication configuration (where the client is unauthenticated and the server has a certificate) has the server sign its negotiation messages, while the client uses post-hoc authentication on its own view. Wireguard uses only post-hoc authentication, but controls the risk by tightly limiting algorithm choices.

Key exchange

Once protocol options have been negotiated by the peers, the next stage of a typical secure communications protocol requires the parties to agree on a shared symmetric secret. If

[6] Most interactive protocols deal with replay attacks by including *nonces* in the signed portion of the protocol, and/or using the nonces within key derivation.

[7] For example, in the FREAK and Logjam attacks on TLS 1.2 (§9.2.1), attackers were able to downgrade the TLS protocol to use "export-grade" key exchanges and thus break the entire protocol.

the peers already possess long-term shared symmetric keys, then these keys can be used to derive a session key using purely symmetric cryptography.[8] If not, key agreement is typically conducted using a public-key algorithm such as (elliptic curve) Diffie-Hellman.

Key exchange algorithms. Many interactive protocols have settled on elliptic-curve Diffie-Hellman (ECDH) as their preferred key exchange algorithm, both for efficiency reasons and because it can be configured to provide an optimal set of features like peer authentication and forward-secrecy. This algorithm is typically implemented over a curve like Curve25519, Curve448, NIST P-256 or P-384 (see §7.3.8.)

In fact, ECDH works so well that "*just use ECDH to agree on keys*" would be the end of this discussion, except for one massive wrinkle: elliptic-curve and traditional Diffie-Hellman are known to be severely vulnerable to quantum attacks. Although appropriate quantum computers don't exist yet, governments and standards agencies are extremely concerned that they will in the near future. To mitigate these threats, agencies like NIST have begun a rapid transition to post-quantum (PQC) algorithms such as ML-KEM (§7.3.5) [299, 21, 193]. In many cases these algorithms are being deployed in a "hybrid" mode where both ECDH and PQC schemes contribute to the key material. This benefit of hybrid key agreement is that the key exchange will retain security even if one of the schemes is completely broken. See §7.3.7 for much more on this subject.

Preventing MITM attacks. The main threat to secure key exchange algorithms is modification of messages by an MITM attacker. In the classical MITM attack (§7.3.7) the attacker substitutes messages from both peers so that each peer unknowingly ends up sharing a key with the attacker. The attacker can then decrypt or modify messages as it wishes.

MITM attacks are not possible if one or both peers can verify the authenticity of the other peer's key exchange messages. For Diffie-Hellman exchanges this is commonly done by having at least one side *sign* their messages. If the other peer knows the public key of the signer (perhaps by manually exchanging it, or by receiving a certificate containing it), they can verify that these messages were not produced by an attacker. This digital signature can also be dual-purposed to protect negotiation messages. If peers have public keys but do not wish to use signatures, they can alternatively use a static (or static/ephemeral) Diffie-Hellman exchange (see 7.3.7) to ensure that the remote party is authentic: this approach is used in Wireguard and Signal Protocol.

What if we don't have certificates or trusted public keys? A challenge may come up in protocols where the parties do not have long-term keys, or cannot easily verify the authenticity of a peer's public key.

[8] This is exactly the approach used in *e.g.,* WPA2's Personal mode: both sides must possess a shared password/key, and the handshake uses this to derive a session key. The downside is that compromise of the long-term key can be catastrophic for the security of past and future sessions.

DRAFT

One example comes up in the Bluetooth protocol, which uses an ECDH key exchange when pairing two devices. Since these devices do not have certificates or sophisticated user interfaces for entering key material, the Bluetooth designers needed a simpler way to detect MITM. The approach they came up with leverages the fact that during an MITM attack, the peers will necessarily see different ephemeral keys (because the MITM must replace the true keys with its own.) The protocol hashes these messages down to a short "pairing code", and then asks the user to ensure that both devices have received the same result.[9]

Key confirmation. In most key exchange algorithms, once the key exchange has been completed, it is important that both parties *confirm* that the other party has agreed on the same key. This ensures that the key agreement protocol has completed correctly and that both sides have derived the same session key. Without key confirmation, some protocols are vulnerable to replay attacks in which an attacker re-plays old handshake messages to confuse participants. A final confirmation message generally protects against this concern.

Data encryption

The normal goal of a secure transport protocol is to securely exchange *application data* between the two parties. To ensure that transmitted data receives both confidentiality and authenticity protection, the recommended approach is to use a secure authenticated encryption scheme (such as an AEAD construction, see §5.3.2.) These schemes ensure that encrypted data is both confidential and authenticated. While this appears straightfoward, real protocols contain a surprising amount of detail.

DRAFT

Encryption key derivation. Most key exchange protocols will produce a single shared key, which is sometimes called a *master shared secret*. While in principle it is possible for the parties to use this key directly to encrypt data, it is not good practice. One major concern in bi-directional protocols (where data is sent in either direction) is *reflection attacks* where an attacker records packets sent by Alice, then replays them back to Alice as though they come from Bob. A further concern is *nonce reuse*, where the two parties accidentally use the same nonces/IVs to encrypt different messages under the same key, leading to tragic outcomes (see §5.3.6.)

While it is possible to engineer around both concerns, a more robust approach is simply to "pluralize" the session key into multiple separate keys prior to encrypting data. For example, a common pattern in bi-directional protocols is to use a Key Derivation Function (KDF) to produce at least two different encryption keys, one for each direction of communication. This removes the need to worry about reflection attacks and nonce-reuse, since the parties no

[9] In practice, the Diffie-Hellman exchange must be carefully augmented to ensure that an MITM attacker cannot brute-force the final message, and thus find collisions in the short pairing code. This is done by having that party *commit* to its message in the first move.

longer use the same key to encrypt. Protocols may derive additional keys for other purposes, as well. For example TLS v1.3 derives a whole tree of keys (see Figure 9.5).

Delivery assumptions. When transmitting data over reliable protocols such as TCP, protocol designers can generally assume that encrypted bytes will be received in the same order that they are sent, at least during normal operation. (In abnormal conditions where an adversary tampers with messages, the protocol must detect this condition and treat it as a fatal error for the session.) This makes encryption easier, since both sides will have fully synchronized state or else they will be in a detectable failure condition.

In some protocols this assumption is not valid. For example, packet-oriented protocols such as Wireguard, IPsec/ESP and DTLS all encrypt individual IP (or UDP) packets. Due to the way IP networks work, these packets may arrive out of order and some may never arrive at all. Decryptors in these protocols need to process each packet as it arrives, without reference to earlier packets that may not have been received.

Recordizing streams. For protocols where messages are already divided into appropriately-sized chunks (such as packets, frames or "messages" in a text messaging protocol), senders can encrypt each incoming chunk of data as it arrives from the application. For stream-oriented protocols like sockets, the protocol may need to make its own decisions about how to "chunk" the incoming byte-stream for encryption.

Protocols such as TLS and SSH typically chop the incoming data into *records*, each of which contains a length field and is encrypted separately. A record in a stream-oriented protocol may contain as little as one byte of data, such as a keypress in SSH.[10] Alternatively, records can be many kilobytes in length. The appropriate size really depends on the nature of your application and the data that is arriving. There are some concerns to keep in mind, however:

1. **Encryption overhead.** When encrypting a record with an AEAD scheme, the ciphertext will typically be larger than the plaintext, due to the *authentication tag* overhead in these schemes. While this overhead (typically about 16 bytes) is of minimal concern when encrypting large record payloads, it can represent a non-trivial percentage of total bandwidth when records are very small.

2. **Latency.** Encrypted data must never be passed to an application before the record has been fully decrypted and authenticated. For applications like SSH, this means that low-latency events (such as keystrokes) may need to be sent immediately in a separate record, even when it might be more bandwidth-efficient to wait and gather many keystrokes together into a larger record.[11]

[10] Records may even be zero bytes in length: a good example is *keep-alive* packets.

[11] Note that a tempting (but highly unsafe) approach I've seen in broken implementations is to encrypt stream-oriented data using a simple unauthenticated stream cipher, since this will allow receivers to decrypt

DRAFT

Data encryption. Once the protocol has broken the input into records or packets, these structures should be encrypted using a modern AEAD scheme. AEAD schemes take four inputs: a key, a *nonce*, the data to be encrypted, and any additional *associated data* that will not be encrypted, but should be authenticated (for example, any fields in the packet/record that are being deliberately sent in plaintext.)

When sending data over reliable transports such as TCP connections, nonces are relatively easy to manage. Sender and recipient can initialize a simple counter to zero when the key is established, and then increment this counter for each subsequent record to be encrypted (resp. decrypted.) Since the two sides can assume in-order delivery in normal operation, there is no need to explicitly send the nonce/counter over the wire.[12]

For packet-oriented protocols that support out-of-order delivery, the situation is more complicated. The sender may still use a simple counter as the encryption nonce: however the recipient will need to obtain this nonce in order to decrypt. The simplest solution is to write the counter into the packet itself (as Wireguard does, §9.2.2.) However, in protocols where payload space is at a premium, the nonce will sometimes be derived from existing fields in the message, such as the identification (IP ID) field in an IP packet. Implementers should be cautious about this, since many packet fields are small enough that only a portion of the full counter can be stored within them, and thus the protocol will need to maintain some state to recover the full counter and avoid replay of older messages.[13]

Re-keying, resumption and post-compromise security

Once a handshake has completed, most protocols can transmit substantial quantities of data without the need for further interaction. However, there are some many conditions that will result in a new handshake and key exchange.

Key usage limits. Well-designed symmetric encryption schemes can process huge amounts of data using a single key. However, there are some limits, usually due to details in the protocol or an underlying AEAD scheme. In TLS v1.3, for example, record counters are represented as a 64-bit number: to avoid nonce re-use, the TLS session must be terminated or re-keyed (with a new handshake) once 2^{64} records have been sent. If we assume each record contains one byte of plaintext, this would only occur after 16,777,216 TB have been sent, making this an awfully impressive TLS session! However, some encryption schemes

DRAFT

and process each byte as it is received. The danger is that an attacker can easily modify these bytes in flight. Even if your protocol detects this tampering after the fact, the unsafe data may already have been processed by the application.

[12] Note again that in-order delivery is only an assumption during normal operation. If an adversary tampers with the ordering of the records, the protocol should detect this and end the session.

[13] Note that it is critical to full verify the authenticity of data before any plaintext is sent to the application logic! Some legacy implementations I have seen would allow the receiving application to process decrypted data prior to verifying the authenticity of the ciphertext. This could lead to bogus data reaching the application, which should never be allowed to happen.

force tighter limits. For example, the TLS v1.3 specification lowers the number of allowed "full-size" records to approximately 2^{24} (or about 16.7 million) when using AES-GCM, due to some foolishness in the GCM standard.

Post-compromise security. Some protocols deliberately update their keys more frequently, as a means to provide granular forward-security and *post-compromise security.* For example, the Signal protocol §12.1 uses a *double-ratchet* mechanism to ensure that session keys are updated every time a message is sent from one party to the other. The first ratchet uses a KDF to update the state of the current key, while deleting the original key: this provides forward-security by ensuring that past key material cannot be recovered following a compromise. A second ratchet conducts a fresh Diffie-Hellman key exchange with each message, ensuring that even if a device's session keys are extracted (*e.g.,* via a stolen cloud backup image), the actual device's key material will rapidly update and hence any stolen key material will have a short useful lifespan.

Session resumption. It is common for devices to make many short-lived sessions to another peer: for example, web browsers will repeatedly contact the same server to download static files. Conducting a full public-key key agreement protocol for each new session can be time consuming. Many protocols provide a faster alternative handshake to enable *session resumption.* The assumption in this case is that the two peers have stored some shared secret key from their previous session. In session resumption mode, these shared secrets can be used to derive a new session key using purely symmetric cryptography, which significantly speeds up the connection.

DRAFT

9.2 Network encryption protocols

The network stack. A sensible place to start thinking about Internet-based encryption protocols is to look at the network stack. In computer networking courses (and literally nowhere else), this is often presented using the OSI model — which divides it into a series of seven *abstraction layers.* Applications such as web browsers sit at the top, while the physical network hardware lies at the bottom, as in Figure 9.3. As data flows downwards through these layers, it is packetized, processed, and formatted so that it can ultimately be sent over physical network hardware. On the recipient machine, a similar process occurs in reverse.

While OSI is mostly a teaching aid, Internet-connected systems use the TCP/IP stack, which performs these steps in real life. Applications at the top of the stack open "sockets", which can be stream-oriented or packet-oriented connections to a remote computer. Their data is then chopped into pieces by the TCP or UDP protocols, and the resulting packets are labeled and routed by the IP protocol. The network interface layers at the bottom handle delivery of those packets to the network. This design suggests three obvious locations for encryption:

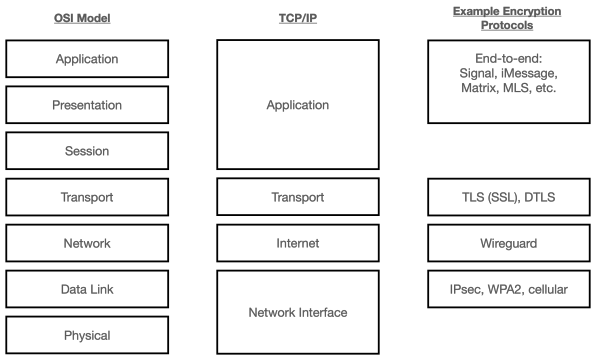

Figure 9.3: The OSI network model and TCP/IP models, divided into layers. On the right, some example network encryption protocols at those layers.

DRAFT

The Application layer. Clearly encryption can be performed at the application layer, meaning it occurs within an application itself. Encryption protocols usually operate at this layer when they work over abstractions that are not common to standard network connections: examples include cloud backup (where files may be encrypted individually for long-term storage) or end-to-end message encryption. The downside of using application-layer encryption is that it involves making custom changes to the application itself, rather than employing the generic and re-usable services that make network stacks useful.

The Transport layer. If the goal is to encrypt standard network connections, a popular location for encryption is at the *transport* layer. This is the location in the network where the network still retains the notion of an individual data session made from one application to another, and the data has not yet been reduced to a storm of data packets. In TCP connections, for example, the data entering this layer is a bi-directional stream of bytes (or "socket") mirroring a single old-fashioned serial connection, and the TCP protocol ensures that this stream is delivered *reliably* even if the network loses some packets or delivers them out of order. Encryption software at this layer can concentrate on separately encrypting each data stream, which is ideal for applications that make connections to different computers (or even different applications on the same remote computer.)

The Data Link and Network layers. A final class of protocols operate at OSI Layer 2 or Layer 3.[14] At these layers, the data from many individual connections has been

[14]In modern systems such as Linux, Layer 3 protocols are aware of concepts such as the *network interfaces* that sit below them. If your system has multiple network interfaces (representing the drivers for WiFi, Ethernet, or VPN interfaces) then those notions will exist at this layer.

chopped apart into packets, which are often mixed together. These layers are ideal for bulk encryption services like Virtual Private Networks (VPNs), where the goal is to encrypt all traffic emanating from a computer, or traveling to a specific set of remote addresses. The data link layer also encompasses dedicated protocols for encrypting radio links, such as WiFi (WPA2) and cellular encryption. Typically the destination here is some remote endpoint (perhaps a firewall device or remote cell tower), which will bulk decrypt the packets and then send them onward to the computers and applications they are destined for.

It is important not to take these abstraction layers too literally! For example, the Transport Layer Security (TLS) protocol is frequently implemented using a software library that resides inside an application's memory space, but it can also be implemented within the OS middleware. Similarly, TCP is normally implemented within the operating system kernel. The concept of network abstraction layers is intended as an aid to teaching and thinking about re-usable services, rather than a literal mapping to the precise layout of a computer's memory or software.

DRAFT

Why does the layer matter? The choice of which layer to encrypt at motivates many decisions that shape an encryption protocol. The first of these is: *who will your protocol establish keys with, and how often?* If you are working at the level of an individual TCP session, then you are negotiating each session individually with one specific endpoint. This destination is chosen by the application, and the connection may never be used again. For example, a web browser may establish a dozen short-lived HTTPS connections to different computers in the course of loading a single website. On the other end of the spectrum, an encrypted WiFi connection may terminate all traffic at one specific access point, and could remain open for long durations of time.

The second decision involves the nature of key establishment and authentication. Does your remote endpoint require a shared symmetric key (sometimes known as a *pre-shared key*, or PSK) in order to establish a remote connection? Or does it use certificates and key exchange to establish a secure connection? For short-lived browser connections, pre-shared keys may be impractical, and hence the second approach will make the most sense.

A further decision involves the nature of the data encryption itself. For reliable stream-oriented connections such as TCP sessions, encryption can be implemented using stateful ciphers such as stream ciphers, which may be configured at the start of the stream and then used continuously until it completes. Packet-oriented protocols, on the other hand, require encryption and decryption be applied individually to each packet – since packets may not arrive in the correct order, or arrive at all. This can require that encryption protocols be aware of low-level details such as packet sequence numbers or fragmentation. Packet-oriented protocols can also struggle with tight space limitations, since elements such as MACs may need to be included in every single packet.