This chapter is broken up into the following topical sections:

- Cryptography and its role in securing the IoT
- Types and uses of the cryptographic primitives in the IoT
- Cryptographic module principles
- Cryptographic key management fundamentals
- Future-proofing your organization's rollout of cryptography

Cryptography and its role in securing the IoT

Our world is witnessing unprecedented growth in machine connectivity over the Internet and private networks. Unfortunately, on any given day, the benefits of that connectivity are soured by yet more news reports of personal, government, and corporate cybersecurity breaches. Hacktivists, nation-states, and organized crime syndicates play a never-ending game of cat and mouse with the security industry. We are all victims, either as a direct result of a cyber breach or through the costs we incur to improve security technology services, insurance, and other risk mitigations. The demand for more security and privacy is being recognized in corporate boardrooms and high-level government circles alike. A significant part of that demand is for wider adoption of cryptography to protect user and machine data. Cryptography will play an ever growing role in securing the IoT. It is and will continue to be used for encrypting wireless edge networks (network and point-to-point), gateway traffic, backend cloud databases, software/firmware images, and many other uses.

Cryptography provides an indispensable tool set for securing data, transactions, and personal privacy in our so-called information age. Fundamentally, when properly implemented, cryptography can provide the following security features to any data whether in transit or at rest:

Security feature	**Cryptographic service(s)**
Confidentiality	Encryption
Authentication	Digital signature or **Message authentication code** (**MAC**)
Integrity	Digital signature or MAC
Non-repudiation	Digital signature

Revisiting definitions from *Chapter 1*, *A Brave New World*, the previously mentioned controls represent four out of five pillars of **information assurance** (**IA**). While the remaining one, availability, is not provided by cryptography, poorly implemented cryptographic instances can certainly deny availability (for example, communication stacks with crypto-synchronization problems).

The security benefits provided by cryptography—confidentiality, authentication, integrity, and non-repudiation—provide direct, one-to-one mitigations against many host, data, and communications security risks. In the not-too-distant past, the author (Van Duren) spent considerable time supporting the FAA in addressing the security needed in pilot-to-drone communications (a prerequisite to safe and secure integration of unmanned aircraft into the national airspace system). Before we could recommend the controls needed, we first needed to understand the different communication risks that could impact unmanned aircraft.

The point is, it is vital to understand the tenets of applied cryptography because many security practitioners—while they may not end up designing protocol level controls—will at least end up making high-level cryptographic selections in the development of security embedded devices and system level security architectures. These selections should always be based on risks.

Types and uses of cryptographic primitives in the IoT

When most people think about cryptography, it is encryption that most comes to mind. They understand that data is "scrambled", so to speak, so that unauthorized parties cannot decrypt and interpret it. Real-world cryptography is comprised of a number of other primitives, however, each partially or fully satisfying one of the previous IA objectives. Securely implementing and combining cryptographic primitives together to achieve a larger, more complex security objective should only be performed or overseen by security professionals well versed in applied cryptography and protocol design. Even the most minor error can prevent the security objective(s) from being fulfilled and result in costly vulnerabilities. There are far more ways to mess up a cryptographic implementation than to get it right.

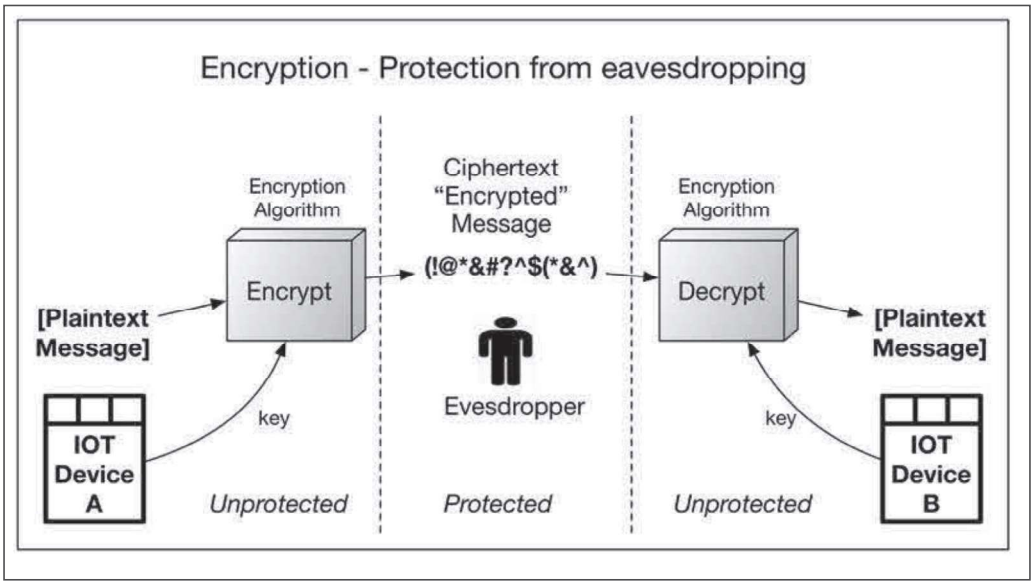
Cryptographic primitive types fall into the following categories:

- Encryption (and decryption):
    - Symmetric
    - Asymmetric
- Hashing
- Digital signatures
    - **Symmetric**: MAC used for integrity and data-origin authentication
    - **Asymmetric**: **Elliptic curve** (**EC**) and **integer factorization cryptography** (**IFC**). These provide integrity, identity, and data-origin authentication as well as non-repudiation
- **Random number generation**: The basis of most cryptography requires very large numbers originating from high entropy sources

Cryptography is seldom used in isolation, however. Instead, it provides the underlying security functions used in upper layer communication and other protocols. For example, Bluetooth, ZigBee, SSL/TLS, and a variety of other protocols specify their own underlying cryptographic primitives and methods of integrating them into messages, message encodings, and protocol behavior (for example, how to handle a failed message integrity check).

Encryption and decryption

Encryption is the cryptographic service most people are familiar with as it is used to so-called scramble or mask information so that unintended parties cannot read or interpret it. In other words, it is used to protect the confidentiality of the information from eavesdroppers and only allow it to be deciphered by intended parties. Encryption algorithms can be symmetric or asymmetric (explained shortly). In both cases, a cryptographic key and the unprotected data are given to the encryption algorithm, which ciphers—encrypts—it. Once in this state, it is protected from eavesdroppers. The receiving party uses a key to decrypt the data when it is needed. The unprotected data is called plaintext and the protected data is called **ciphertext**. The basic encryption process is depicted in the following diagram:



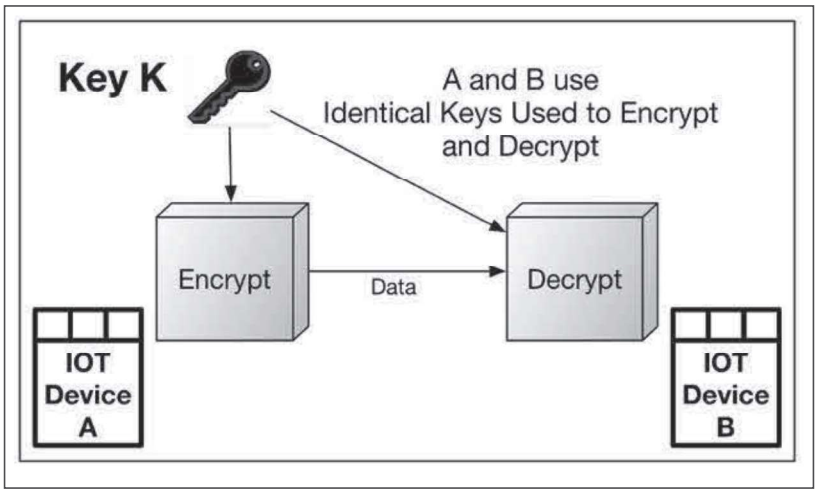
Encrypt-decrypt.graffle

It should be clear from the preceding diagram that, if the data is ever decrypted prior to reaching IOT Device B, it is vulnerable to the Eavesdropper. This brings into question where in a communication stack and in what protocol the encryption is performed, that is, what the capabilities of the endpoints are. When encrypting for communication purposes, system security engineers need to decide between point-to-point encryption and end-to-end encryption as evidenced in their threat modeling. This is an area ripe for error, as many encrypted protocols operate only on a point-to-point basis and must traverse a variety of gateways and other intermediate devices, the paths to which may be highly insecure.

In today's Internet threat environment, end-to-end encryption at the session and application layers is most prominent due to severe data losses that can occur when decrypting within an intermediary. The electrical industry and the insecure SCADA protocols commonly employed in it provide a case in point. The security fixes often include building secure communication gateways (where newly added encryption is performed). In others, it is to tunnel the insecure protocols through end-to-end protected ones. System security architectures should clearly account for every encryption security protocol in use and highlight where plaintext data is located (in storage or transit) and where it needs to be converted (encrypted) into ciphertext. In general, whenever possible, end-to-end data encryption should be promoted. In other words, a secure-by-default posture should always be promoted.

Symmetric encryption

Symmetric encryption simply means the sender (encryptor) and the receiver (decryptor) use an identical cryptographic key. The algorithm, which is able to both encrypt and decrypt—depending on the mode—is a reversible operation, as shown in the following diagram:



Symmetric-encryption.graffle

In many protocols, a different symmetric key is used for each direction of travel. So, for example, Device A may encrypt to Device B using key X. Both parties have key X. The opposite direction (B to A) may use key Y which is also in the possession of both parties.

Symmetric algorithms consist of a ciphering operation using the plaintext or ciphertext input, combined with the *shared* cryptographic key. Common ciphers include the following:

- **AES—advanced encryption standard** (based on Rijndael and specified in FIPS PUB 197)
- Blowfish
- DES and triple-DES
- Twofish
- CAST-128
- Camellia
- IDEA

The source of the cryptographic keys is a subject that spans applied cryptography as well as the topic of cryptographic key management, addressed later in this chapter.

In addition to the cryptographic key and data that is fed to the cipher, an **initialization vector** (**IV**) is frequently needed to support certain cipher modes (explained in a moment). Cipher modes beyond the basic cipher are simply different methods of bootstrapping the cipher to operate on successive chunks (blocks) of plaintext and ciphertext data. **Electronic code book** (**ECB**) is the basic cipher and operates on one block of plaintext or ciphertext at a time. The ECB mode cipher by itself is very rarely used because repeated blocks of identical plaintext will have an identical ciphertext form, thus rendering encrypted data vulnerable to catastrophic traffic analysis. No IV is necessary in ECB mode, just the symmetric key and data on which to operate. Beyond ECB, block ciphers may operate in block chaining modes and stream/counter modes, discussed next.

Block chaining modes

In **cipher block chaining** (**CBC**) mode, the encryption is bootstrapped by inputting an IV that is XOR'd with the first block of plaintext. The result of the XOR operation goes through the cipher to produce the first block of encrypted ciphertext. This block of ciphertext is then XOR'd with the next block of plaintext, the result of which goes through the cipher again. The process continues until all of the blocks of plaintext have been processed. Because of the XOR operation between iterating blocks of plaintext and ciphertext, two identical blocks of plaintext will not have the same ciphertext representation. Thus, traffic analysis (the ability to discern what the plaintext was from its ciphertext) is far more difficult.

Other block chaining modes include **cipher-feedback chaining** (**CFB**) and output feedback modes (OFB), each a variation on where the IV is initially used, what plaintext and ciphertext blocks are XOR'd, and so on.

Advantages of block chaining modes include the fact, stated previously, that repeated blocks of identical plaintext do not have an identical ciphertext form. This prevents the simplest traffic analysis methods such as using dictionary word frequency to interpret encrypted data. Disadvantages of block chaining techniques include the fact that any data errors such as bit flipping in RF communications propagate downstream. For example, if the first block of a large message M encrypted by AES in CBC mode were corrupted, all subsequent blocks of M would be corrupted as well. Stream ciphers, discussed next, do not have this problem.

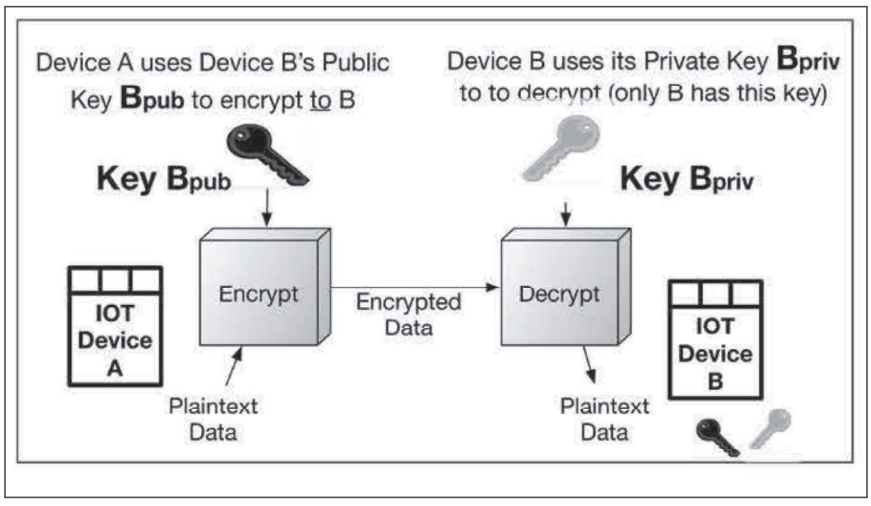
CBC is a common mode and is currently available as an option (among others), for example, in the ZigBee protocol (based on IEEE 802.15.4).

Counter modes

Encryption does not have to be performed on complete blocks, however; some modes make use of a counter such as **counter mode** (**CTR**) and **Galois counter mode** (**GCM**). In these, the plaintext data is not actually encrypted with the cipher and key, not directly anyway. Rather, each bit of plaintext is XOR'd with a stream of continuously produced ciphertext comprising encrypted counter values that continuously increment. In this mode, the initial counter value is the IV. It is encrypted by the cipher (using a key), providing a block of ciphertext. This block of ciphertext is XOR'd with the block (or partial block) of plaintext requiring the protection. CTR mode is frequently used in wireless communications because bit errors that happen during transmission do not propagate beyond a single bit (versus block chaining modes). It is also available within IEEE 802.15.4, which supports a number of IoT protocols.

Asymmetric encryption

Asymmetric encryption simply means there are two different, pairwise keys, one public and the other private, used to encrypt and decrypt, respectively. In the following diagram, IoT device A uses IoT device B's public key to encrypt to device B. Conversely, device B uses device A's public key to encrypt information to device A. Each device's private keys are kept secret, otherwise anyone or anything possessing them will be able to decrypt and view the information.



Asymmetric-Encryption.graffle

The only asymmetric encryption algorithm in use today is that of **RSA** (**Rivest, Shamir, Adelman**), an **integer factorization cryptography** (**IFC**) algorithm that is practical for encrypting and decrypting small amounts of data (up to the modulus size in use).

The advantage of this encryption technique is that only one party possessing the pairwise RSA private key can decrypt the traffic. Typically, private key material is not shared with more than one entity.

The disadvantage of asymmetric encryption (RSA), as stated earlier, is the fact that it is limited to encrypting up to the modulus size in question (1024 bits, 2048 bits, and so on). Given this disadvantage, the most common use of RSA public key encryption is to encrypt and transport other small keys—frequently symmetric—or random values used as precursors to cryptographic keys. For example, in the TLS client-server protocol, RSA is leveraged by a client to encrypt a **pre-master secret** (**PMS**) with the server's public RSA key. After sending the encrypted PMS to the server, each side has an exact copy from which to derive the session's symmetric key material (needed for session encryption and so on).

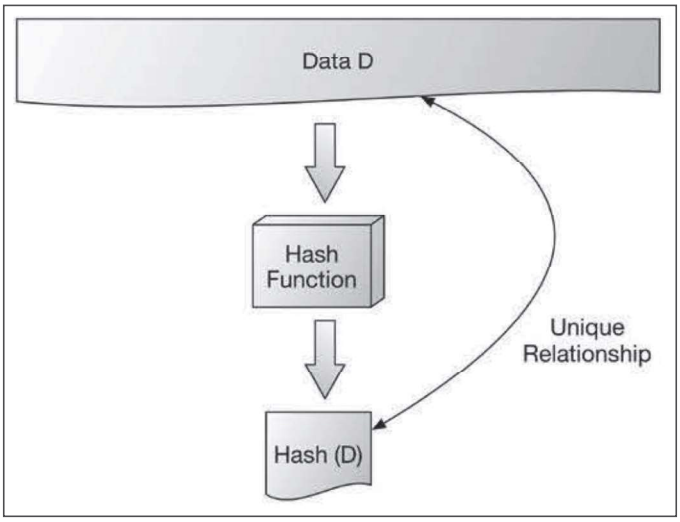
Integer factorization cryptography using RSA, however, is becoming less popular due to advances in large number factorization techniques and computing power. Larger RSA modulus sizes (for improved computational resistance to attack) are now recommended by NIST.

Hashes

Cryptographic hashes are used in a variety of security functions for their ability to represent an arbitrarily large message with a small sized, unique thumbprint (the hash). They have the following properties:

- They are designed not to disclose any information about the original data that was hashed (this is called resistance to first pre-image attacks)
- They are designed to not allow two different messages to have the same hash (this is called resistance to second pre-image attacks and collisions)
- They produce a very random-looking value (the hash)

The following image denotes an arbitrary chunk of data D being hashed into H(D). H(D) is a small, fixed size (depending on the algorithm in use); from it, one can not (or should not be able to) discern what the original data D was.



Hash-functions.graffle

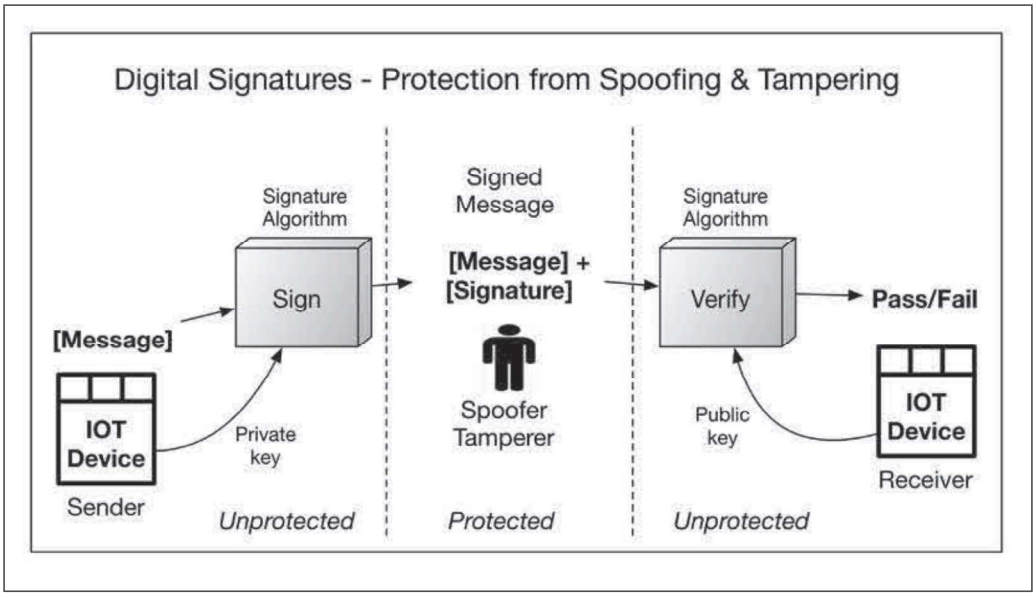
Given these properties, hash functions are frequently used for the following purposes:

- Protecting passwords and other authenticators by hashing them (the original password is then not revealed unless by a *dictionary attack*) into a random looking digest
- Checking the integrity of a large data set or file by storing the proper hash of the data and re-computing that hash at a later time (often by another party). Any modification of the data or its hash is detectable.
- Performing asymmetric digital signatures
- Providing the foundation for certain message authentication codes
- Performing key derivation
- Generating pseudo-random numbers

Digital signatures

A **digital signature** is a cryptographic function that provides integrity, authentication, data origin, and in some cases, non-repudiation protections. Just like a hand-written signature, they are designed to be unique to the *signer*, the individual or device responsible for signing the message and who possesses the signing key. Digital signatures come in two flavors, representing the type of cryptography in use: symmetric (secret, shared key) or asymmetric (private key is unshared).

The originator in the following diagram takes his message and signs it to produce the signature. The signature can now accompany the message (now called the signed message) so that anyone with the appropriate key can perform the inverse of signature operation, called **signature verification**.



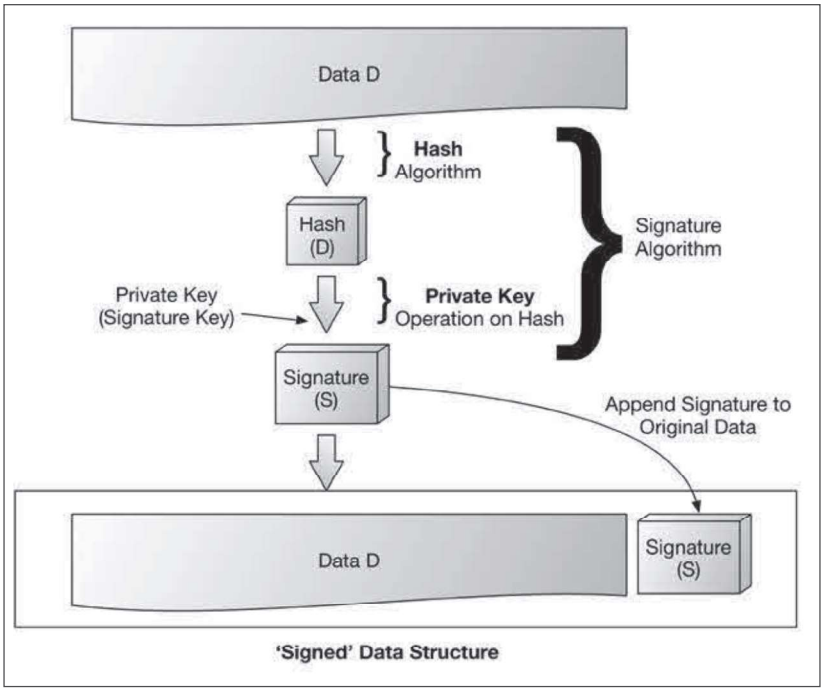
sign-verify.graffle

If the signature verification is successful, the following can be claimed:

- The data was, indeed, signed by a known or declared key
- The data has not been corrupted or tampered with

If the signature verification process fails, then the verifier should not trust the data's integrity or whether it has originated from the right source. This is true of both asymmetric and symmetric signatures, but each has unique properties, described next.

Asymmetric signature algorithms generate signatures (that is, sign) using a private key associated with a shared public key. Being asymmetric and the fact that private keys are generally not (nor should they typically ever be) shared, asymmetric signatures provide a valuable means of performing both entity and data authentication as well as protecting the integrity of the data and providing non-repudiation capabilities.



Asymmetric-signature.graffle

Common asymmetric digital signature algorithms include the following:

- RSA (with PKCS1 or PSS padding schemes)
- **DSA** (**digital signature algorithm**) (FIPS 180-4)
- **Elliptic curve DSA** (**ECDSA**) (FIPS 180-4)

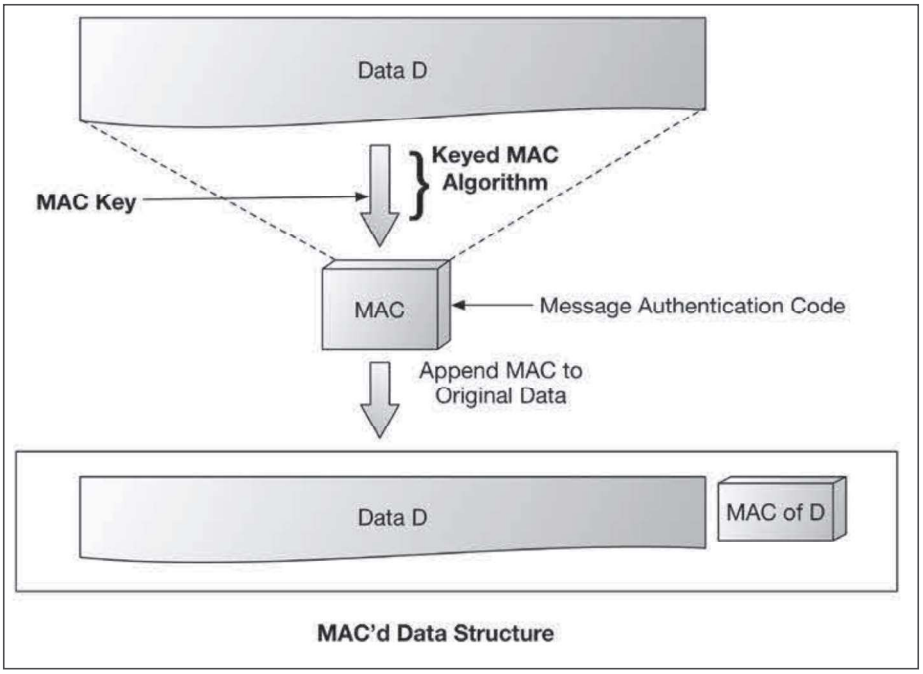
Asymmetric signatures are used to authenticate from one machine to another, sign software/firmware (hence, verify source and integrity), sign arbitrary protocol messages, sign PKI public key certificates (discussed in *Chapter 6*, *Identity and Access Management Solutions for the IoT*) and verify each of the preceding ones. Given that digital signatures are generated using a single private (unshared) key, no entity can claim that it did not sign a message. The signature can only have originated from that entity's private key, hence the property of non-repudiation.

Asymmetric digital signatures are used in a variety of cryptographic-enabled protocols such as SSL, TLS, IPSec, S/MIME, ZigBee networks, Connected Vehicle Systems (IEEE 1609.2), and many others.

Symmetric (MACs)

Signatures can also be generated using symmetric cryptography. Symmetric signatures are also called MAC and, like asymmetric digital signatures, produce a MAC of a known piece of data, D. The principal difference is that MACs (signatures) are generated using a symmetric algorithm, hence the same key used to generate the MAC is also used to verify it. Keep in mind that the term MAC is frequently used to refer to the algorithm as well as the signature that it generates.

Symmetric MAC algorithms frequently rely on a hash function or symmetric cipher to generate the message authentication code. In both cases (as shown in the following diagram), a MAC key is used as the shared secret for both the sender (signer) and receiver (verifier).



Symmetric-signature.graffle

Given that MAC-generating symmetric keys may be shared, MACs generally do not claim to provide identity-based entity authentication (therefore, non-repudiation cannot be claimed), but do provide sufficient verification of origin (especially in short term transactions) that they are said to provide data origin authentication.

MACs are used in a variety of protocols, such as SSL, TLS, IPSec, and many others. Examples of MACs include the following:

- HMAC-SHA1
- HMAC-SHA256
- CMAC (using a block cipher like AES)
- GMAC (**Galois message authentication code** is the message authentication element of the GCM mode)

MAC algorithms are frequently integrated with encryption ciphers to perform what is known as authenticated encryption (providing both confidentiality as well as authentication in one fell swoop). Examples of authenticated encryption are as follows:

- **Galois counter mode** (**GCM**): This mode combines AES-CTR counter mode with a GMAC to produce ciphertext and a message authentication code.
- **Counter mode with CBC-MAC** (**CCM**): This mode combines a 128-bit block cipher such as AES in CTR mode with the MAC algorithm CBC-MAC. The CBC-MAC value is included with the associated CTR-encrypted data.

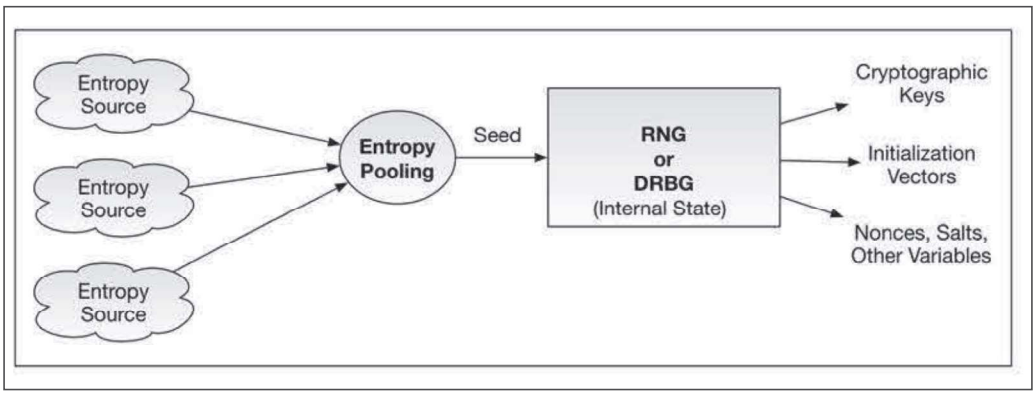
Authenticated encryption is available in a variety of protocols such as TLS.

Random number generation

Randomness of numbers is a keystone of cryptography given their use in generating a number of different cryptographic variables such as keys. Large, random numbers are difficult to guess or iterate through (brute force), whereas highly deterministic numbers are not. Random number generators—RNGs—come in two basic flavors, deterministic and nondeterministic. Deterministic simply means they are algorithm-based and for a single set of inputs they will always produce the same output. Non-deterministic means the RNG is generating random data in some other fashion, typically from very random physical events such as circuit noise and other low bias sources (even semi-random interrupts occurring in operating systems). RNGs are frequently among the most sensitive components of a cryptographic device given the enormous impact they have on the security and source of cryptographic keys.

Any method of undermining a device's RNG and discerning the cryptographic keys it generated renders the protections of that cryptographic device completely useless.

RNGs (the newer generation are called **deterministic random bit generators**, or **DRBGs**) are designed to produce random data for use as cryptographic keys, initialization vectors, nonces, padding, and other purposes. RNGs require inputs called **seeds** that must also be highly random, emanating from high entropy sources. A compromise of seed or its entropy source—through poor design, bias, or malfunction—will lead to a compromise of the RNG outputs and therefore a compromise of the cryptographic implementation. The result: someone decrypts your data, spoofs your messages, or worse. A generalized depiction of the RNG entropy seeding process is shown in the following diagram:



RandomNumberGeneration.graffle

In this depiction, several arbitrary entropy sources are pooled together and, when needed, the RNG extracts a seed value from this pool. Collectively, the entropy sources and entropy pooling processes to the left of the RNG are often called a **non-deterministic random number generator** (**NDRNG**). NDRNG's almost always accompany RNGs as the seeding source.

Pertinent to the IoT, it is absolutely critical for those IoT devices generating cryptographic material that IoT RNGs be seeded with high entropy sources and that the entropy sources are well protected from disclosure, tampering, or any other type of manipulation. For example, it is well known that random noise characteristics of electrical circuits change with temperature; therefore, it is prudent in some cases to establish temperature thresholds and logically stop entropy gathering functions that depend on circuit noise when temperature thresholds are exceeded. This is a well-known feature used in smart cards (for example, chip cards for credit/debit transactions, and so on) to mitigate attacks on RNG input bias by changing the temperature of the chip.

Entropy quality should be checked during device design. Specifically, the min-entropy characteristics should be evaluated and the IoT design should be resilient to the NDRNG becoming 'stuck' and always feeding the same inputs to the RNG. While less a deployment consideration, IoT device vendors should take extraordinary care to incorporate high quality random number generation capabilities during the design of a device's cryptographic architecture. This includes production of high quality entropy, protection of the entropy state, detection of stuck RNGs, minimization of RNG input bias, entropy pooling logic, RNG state, RNG inputs, and RNG outputs. Note that if entropy sources are poor, engineering tradeoffs can be made to simply collect (pool) more of the entropy within the device to feed the RNG.

NIST Special Publication 800-90B (`http://csrc.nist.gov/publications/drafts/800-90/sp800-90b_second_draft.pdf`) provides an excellent resource for understanding entropy, entropy sources, and entropy testing. Vendors can have RNG/DRBG conformance and entropy quality tested by independent cryptographic test laboratories or by following guidance in SP800-90B (`http://csrc.nist.gov/publications/drafts/800-90/draft-sp800-90b.pdf`).

Ciphersuites

The fun part of applied cryptography is combining one or more of the above algorithm types to achieve specifically desired security properties. In many communication protocols, these algorithm groupings are often called **ciphersuites**. Depending on the protocol at hand, a cipher-suite specifies the particular set of algorithms, possible key lengths, and uses of each.

Ciphersuites can be specified and enumerated in different ways. For example, **transport layer security** (**TLS**) offers a wide array of ciphersuites to protect network sessions for web services, general HTTP traffic, **real-time protocols** (**RTP**), and many others. An example TLS cipher-suite enumeration and their interpretation is as follows:

`TLS_RSA_WITH_AES_128_GCM_SHA256`, which interprets to using:

- RSA algorithm for the server's public key certificate authentication (digital signature). RSA is also the public key-based key transport (for passing the client-generated pre-master secret to the server).
- AES algorithm (using 128-bit length keys) for encrypting all data through the TLS tunnel.
- AES encryption is to be performed using the **Galois counter mode** (**GCM**); this provides the tunnel's ciphertext as well as the MACs for each TLS datagram.
- SHA256 to be used as the hashing algorithm.

Using each of the cryptographic algorithms indicated in the cipher-suite, the specific security properties needed of the TLS connection and its setup are realized:

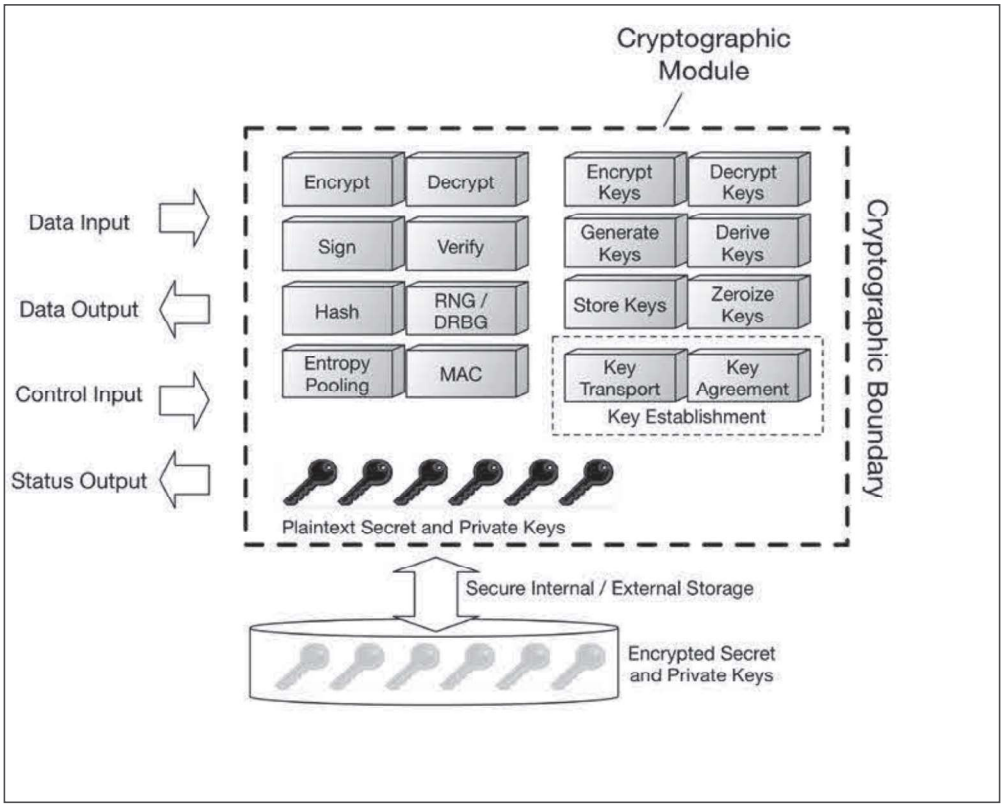
1. The client authenticates the server by validating an RSA-based signature on its public key certificate (the RSA signature was performed over a SHA256 hash of the public key certificate, actually).
2. Now a session key is needed for tunnel encryption. The client encrypts its large, randomly generated number (called **pre-master secret**) using the server's public RSA key and sends it to the server (that is, only the server, and no man-in-the-middle, can decrypt it).
3. Both the client and server use the pre-master secret to compute a master secret. Key derivation is performed for both parties to generate an identical key blob containing the AES key that will encrypt the traffic.
4. The AES-GCM algorithm is used for AES encryption/decryption—this particular mode of AES also computes the MAC appended to teach TLS datagram (note that some TLS ciphersuites use the HMAC algorithm for this).

Other cryptographic protocols employ similar types of ciphersuites (for example, IPSec), but the point is that no matter the protocol—IoT or otherwise—cryptographic algorithms are put together in different ways to counter specific threats (for example, MITM) in the protocol's intended usage environment.

Cryptographic module principles

So far, we have discussed cryptographic algorithms, algorithm inputs, uses, and other important aspects of applied cryptography. Familiarity with cryptographic algorithms is not enough, however. The proper implementation of cryptography in what are called cryptographic modules, though a topic not for the faint of heart, is needed for IoT security. Earlier in my (Van Duren) career, I had the opportunity not only to test many cryptographic devices, but also manage, as laboratory director, two of the largest NIST-accredited FIPS 140-2 cryptographic test laboratories. In this capacity, I had the opportunity to oversee and help validate literally hundreds of different device hardware and software implementations, smart cards, hard drives, operating systems, **hardware security modules** (**HSM**), and many other cryptographic devices. In this section, I will share with you some of the wisdom gained from these experiences. But first, we must define a cryptographic module.

A cryptographic implementation can come from device OEMs, ODMs, BSP providers, security software establishments, just about anyone. A cryptographic implementation can be realized in hardware, software, firmware, or some combination thereof, and is responsible for processing the cryptographic algorithms and securely storing cryptographic keys (remember, compromise of your keys means compromise of your communications or other data). Borrowing NIST's term from the US Government's cryptographic module standard, FIPS 140-2, a cryptographic module is "the set of hardware, software, and/or firmware that implements approved security functions (including cryptographic algorithms and key generation) and is contained within the cryptographic boundary" (`http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`). The cryptographic boundary, also defined in FIPS 140-2, is *an explicitly defined continuous perimeter that establishes the physical bounds of a cryptographic module and contains all the hardware, software, and/or firmware components of a cryptographic module.* A generalized representation of a cryptographic module is shown in the following image:



Crypto-modules.graffle

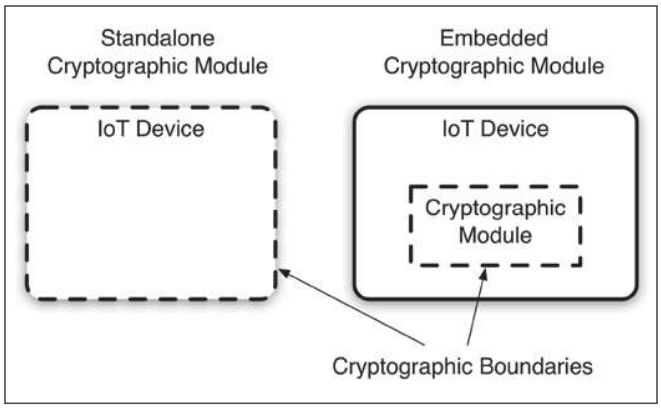
Without creating a treatise on cryptographic modules, the security topics that pertain to them include the following:

- Definition of the cryptographic boundary
- Protecting a module's ports and other interfaces (physical and logical)
- Identifying who or what connects (local or remote users) to the cryptographic module, how they authenticate to it and what services—security-relevant or not – the module provides them
- Proper management and indication of state during self tests and error conditions (needed by the host IoT device)
- Physical security—protection against tampering and/or response to tamper conditions
- Operating system integration, if applicable
- Cryptographic key management relevant to the module (key management is discussed in much more detail from a system perspective later), including how keys are generated, managed, accessed, and used
- Cryptographic self tests (health of the implementation) and responses to failures
- Design assurance

Each of the preceding areas roughly maps to each of the 11 topic areas of security in the FIPS 140-2 standard (note that, at this time, the standard is poised to be updated and superseded).

One of the principal functions of the cryptographic module is to protect cryptographic keys from compromise. Why? Simple. If keys are compromised, there's no point encrypting, signing, or otherwise protecting the integrity of the data using cryptography. Yes, if one doesn't properly engineer or integrate the cryptographic module for the threat environment at hand, there may little point in using cryptography at all.

One of the most important aspects of augmenting IoT devices with cryptography is the definition, selection, or incorporation of another device's cryptographic boundary. Generally speaking, a device can have an internal, embedded cryptographic module, or the device can itself be the cryptographic module (that is, the IoT device's enclosure is the crypto boundary).



crypto-module-embodiments.graffle

From an IoT perspective, the cryptographic boundary defines the cryptographic island on which all cryptographic functions are to be performed within a given device. Using an embedded crypto module, IoT buyers and integrators should verify with IoT device vendors that, indeed, no cryptography whatsoever is being performed outside of the embedded cryptographic module's boundary.

There are advantages and disadvantages to different cryptographic module embodiments. In general, the smaller and tighter the module, 1) the less attack surface and 2) the less software, firmware, and hardware logic there is to maintain. The larger the boundary (as in some standalone crypto modules), the less flexibility to alter non-cryptographic logic, something much more important to vendors and system owners who may be required to use, for example, US Government validated FIPS 140-2 crypto modules (discussed next).

Both product security designers and system security integrators need to be fully aware of the implications of how devices implement cryptography. In many cases, product vendors will procure and integrate internal cryptographic modules that have been validated by independent FIPS testing laboratories.

This is strongly advisable for the following reasons:

- **Algorithm selection**: While algorithm selection can be a contentious issue with regard to national sovereignty, in general, most organizations such as the US government do not desire weak or otherwise unproven cryptographic algorithms to be used to protect sensitive data. Yes, there are excellent algorithms that are not approved for US government use, but in addition to ensuring the selection and specification of good algorithms, NIST also goes to great lengths to ensure old algorithms and key lengths are discontinued when they become outdated from advances in cryptanalytic and computational attacks. In other words, sticking to well established and well-specified algorithms trusted by a large government is not a bad idea. A number of NIST-accepted algorithms are also trusted by the **National Security Agency** (**NSA**) for use in protecting up to top secret data—with the caveat that the cryptographic module meets NSA type standards relevant to assurance levels needed for classified information. Algorithms such as AES (256-bit key lengths), ECDSA and ECDH are both allowed by NIST (for unclassified) and the NSA (for classified) under certain conditions.
- **Algorithm validation**: Test laboratories validate—as part of a crypto module test suite—the correctness (using a variety of known answer and other tests) of cryptographic algorithm implementations as they operate on the module. This is beneficial because the slightest algorithmic or implementation error can render the cryptography useless and lead to severe information integrity, confidentiality, and authentication losses. Algorithm validation is NOT cryptographic module validation; it is a subset of it.
- **Cryptographic module validation**: Test laboratories also validate that each and every applicable FIPS 140-2 security requirement is satisfied at or within the defined cryptographic boundary according to its security policy. This is performed using a variety of conformance tests, ranging from device specification and other documentation, source code, and very importantly, operational testing (as well as algorithm validation, mentioned previously).

This brings us to identifying some of hazards of FIPS 140-2 or any other security conformance test regimen, especially as they relate to the IoT. As a US government standard, FIPS 140-2 is applied incredibly broadly to any number of device types, and as such, can lose a degree of interpretive specificity (depending on the properties of the device to which one attempts to apply the standard). In addition, the validation only applies to a vendor-selected cryptographic boundary—and this boundary may or may not be truly suitable for certain environments and related risks. This is where NIST washes its hands. There were a number of instances when consulting with device vendors where I advised vendors against defining a cryptographic boundary that I knew was disingenuous at best, insecure at worst. However, if the vendor was able to meet all of the FIPS 140-2 requirements at their selected boundary, there was nothing I could do as an independent test laboratory to deny them the strategy. Conformance requirements versus actual security obtained by satisfying them is a never-ending struggle in standards bodies and conformance test regimes.

Given the previous benefits (and also hazards), the following advice is given with regard to utilization and deployment of FIPS 140-2 cryptographic modules in your IoT implementations:

- No device should use interfaces to a cryptographic algorithm aside from those provided by its parent crypto module (meaning outside of the cryptographic boundary). In fact, a device should not perform any cryptographic functions outside of a secured perimeter.
- No device should ever store a plaintext cryptographic key outside of its crypto module's boundary (even if it is still within the device but outside its embedded crypto module). Better yet, store all keys in encrypted form and then apply the strictest protections to the key-encrypting key.
- System integrators, when integrating cryptographic devices, should consult the device vendors and check the publicly available database on how the crypto module was defined prior to integration into the device. The definition of its cryptographic boundary, by US regulation, is identified in the module's non-proprietary security policy (posted online). Validated FIPS 140-2 modules can be checked at the following location: `http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-all.htm`. It is necessary to understand the degree to which an embedded module secures itself versus relying on its host (for example, with regard to physical security and tampering).

- Select cryptographic modules whose FIPS 140-2 validation assurance levels (1-4) are commensurate with the threat environment into which you plan to deploy them. For example, physical security at FIPS 140-2, level 2 does not require a tamper response mechanism (to wipe sensitive key material upon tamper); levels 3 and 4 do, however. If deploying modules into very high threat environments, select higher levels of assurance OR embed lower-level assurance modules into additionally secured hosts or facilities.
- When integrating a cryptographic module, ensure that the intended operators, host devices, or interfacing endpoints identified in the module's Security Policy map to actual users and non-human devices in the system. Applicable roles, services and authentication to a cryptographic module may be external or internal to a device; integrators need to know this and ensure the mapping is complete and secure.
- When implementing more complicated integrations, consult individuals and organizations that have expertise not only in applied cryptography, but also in cryptographic modules, device implementation, and integration. There are far more ways to get the cryptography wrong than to get it right.

Using validated cryptographic implementations is an excellent practice overall, but do it smartly and don't assume that certain cryptographic modules that would seem to meet all of the functional and performance requirements are a good idea for all environments.

Cryptographic key management fundamentals

Now that we have addressed basic cryptography and cryptographic modules, it is necessary to delve into the topic of cryptographic key management. Cryptographic modules can be considered cryptographically secured islands in larger systems, each module containing cryptographic algorithms, keys, and other assets needed to protect sensitive data. Deploying cryptographic modules securely, however, frequently requires cryptographic key management. Planning key management for an embedded device and/or full scale IoT enterprise is essential to securing and rolling out IoT systems. This requires organizations to normalize the types of cryptographic material within their IoT devices and ensure they work across systems and organizations. Key management is the art and science of protecting cryptographic keys within devices (crypto modules) and across the enterprise. It is an arcane technical discipline that was initially developed and evolved by the US Department of Defense long before most commercial companies had an inkling of what it was or had any need for cryptography in the first place. Now, more than ever, it is a subject that organizations must get right in order to secure connected things in our world.

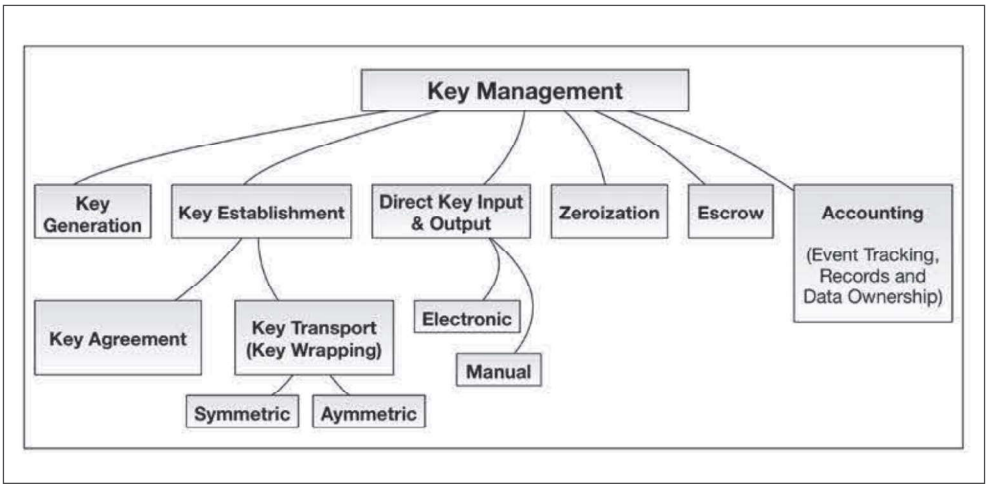
The fallout from the Walker spy ring led to the creation of many of the key management systems and techniques widely used today by the Department of Defense and NSA today. Starting in 1968, US Navy officer John Walker began selling classified cryptographic key material to the Soviet intelligence services. Because this internal compromise was not discovered for many years (he was not caught until 1985), the total damage to US national security was enormous. To prevent crypto key material compromise and maintain a highly accountable system of tracking keys, various DoD services (the Navy and the Air Force) began creating their own key management systems that were eventually folded into what is today known as the NSA's **Electronic Key Management System** (**EKMS**). The EKMS is now being modernized into the **key management infrastructure** (**KMI**) (`https://en.wikipedia.org/wiki/John_Anthony_Walker`).

The topic of cryptographic key management is frequently misunderstood, often more so than cryptography itself. Indeed, there are few practitioners in the discipline. Cryptography and key management are siblings; the security provided by each depends enormously on the other. Key management is often not implemented at all or is implemented insecurely. Either way, unauthorized disclosure and compromise of cryptographic keys through poor key management renders the use of cryptography moot. Necessary privacy and assurance of information integrity and origin is lost.

It is also important to note that the standards that specify and describe PKIs are based on secure key management principles. PKIs, by definition, *are* key management systems. Regarding the IoT, it is important for organizations to understand the basic principles of key management because not all IoT devices will interact with and consume PKI certificates (that is, be able to benefit from third party key management services). A variety of other cryptographic key types—symmetric and asymmetric—will be utilized in the IoT whether it's administering devices (SSH), providing cryptographic gateways (TLS/IPSec), or just performing simple integrity checks on IoT messages (using MACs).

Why is key management important? Disclosure of many types of cryptographic variables can lead to catastrophic data loss even years or decades after the cryptographic transaction has taken place. Today's Internet is replete with people, systems, and software performing a variety of man-in-the-middle attacks, ranging from simple network monitoring to full-scale nation state attacks and compromises of hosts and networks. One can collect or re-route otherwise encrypted, protected traffic and store it for months, years, or decades. In the meantime, the collectors can clandestinely work for long periods of time to exploit people (human intelligence, as in John Walker) and technology (this usually requires a cryptanalyst) to acquire the keys that were used to encrypt the collected transactions. Within IoT devices, centralized key generation and distribution sources or storage systems, key management systems and processes perform the dirty work of ensuring cryptographic keys are not compromised during machine or human handling.

Key management addresses a number of cryptographic key handling topics pertinent to the devices and the systems in which they operate. These topics are indicated in the following relational diagram:



KeyMgmt-hierarchy.graffle

Key generation

Key generation refers to how, when, and on what devices cryptographic keys are generated and using what algorithms. Keys should be generated using a well vetted RNG or DRBG seeded with sufficient min-entropy (discussed earlier). Key generation can be performed directly on the device or in a more centralized system (the latter requiring subsequent distribution to the device).

Key establishment

Much confusion exists in terms of what constitutes cryptographic *key establishment*. Key establishment is simply the act of two parties either 1) agreeing on a specific cryptographic key or 2) acting as sender and receiver roles in the transport of a key from one to the other. More specifically, it is as follows:

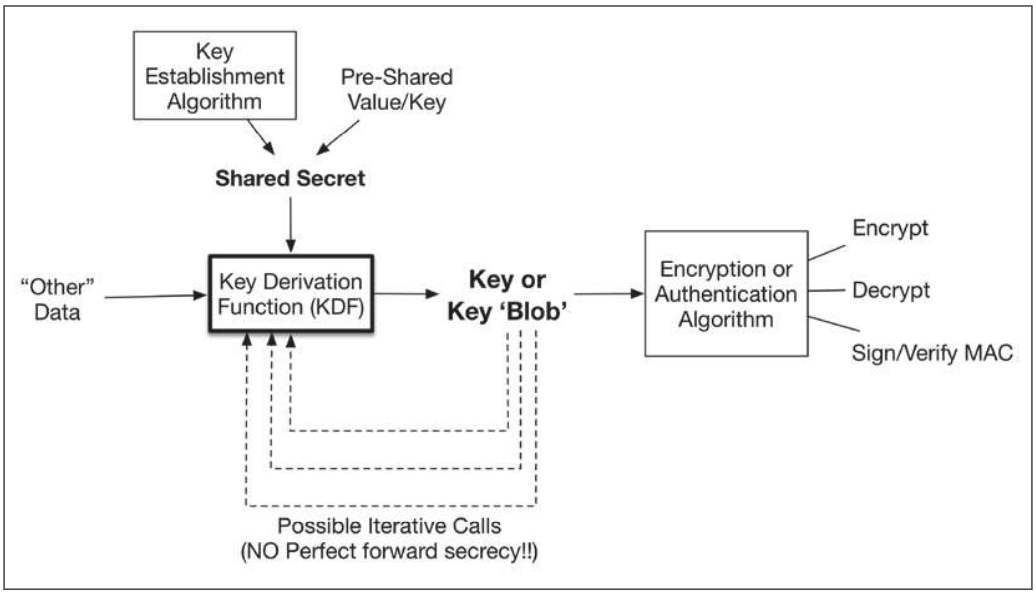
- **Key agreement** is the act of two parties contributing algorithmically to the creation of a shared key. In other words, generated or stored public values from one party are sent to the other (frequently in plaintext) and input into complementary algorithm processes to arrive at a shared secret. This shared secret (in conventional, cryptographic best practices) is then input to a key derivation function (frequently hash-based) to arrive at a cryptographic key or set of keys (key blob).

- **Key transport** is the act of one party transmitting a cryptographic key or its precursor to another party by first encrypting it with a **key encryption key** (**KEK**). The KEK may be symmetric (for example, an AES key) or asymmetric (for example, a RSA public key). In the former case, the KEK must be securely pre-shared with the recipient or also established using some type of cryptographic scheme. In the latter case, the encrypting key is the recipient's public key and only the recipient may decrypt the transported key using their private key (not shared).

Key derivation

Key derivation refers to how a device or piece of software constructs cryptographic keys from other keys and variables, including passwords (so called password-based key derivation). NIST SP800-108 asserts "*….a key derivation function (KDF) is a function with which an input key and other input data are used to generate (that is, derive) keying material that can be employed by cryptographic algorithms.*" Source: `http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf`.

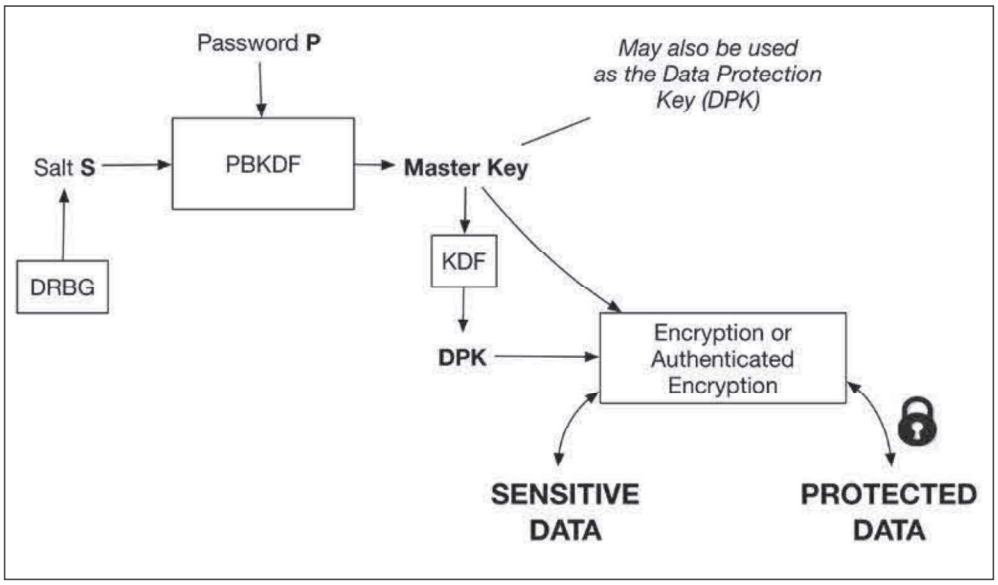
A generalized depiction of key derivation is shown in the following image:



KDF.graffle

Poor practices in key derivation led to the US government disallowing their use with certain exceptions until best practices could be incorporated into the NIST special publications. Key derivation is frequently performed in many secure communication protocols such as TLS and IPSec by deriving the actual session keys from an established shared secret, transported random number (for example, pre-master secret in SSL/TLS), or current key.

Password-based key derivation (**PBKDF**) is the process of deriving, in part, a cryptographic key from a unique password and is specified in NIST SP 800-132. A generalized depiction of this process is shown in the following image:



PBKDF.graffle

Source: `http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf`

Key storage

Key storage refers to how secure storage of keys (frequently encrypted using KEKs) is performed and in what type of device(s). Secure storage may be achieved by encrypting a database (with excellent protection of the database encryption key) or other types of key stores. In enterprise key escrow/storage systems, cryptographic keys should be encrypted using a **hardware security module** (**HSM**) prior to long-term storage. HSMs, themselves cryptographic modules, are specifically designed to be very difficult to hack by providing extensive physical and logical security protections. For example, most HSMs possess a tamper-responsive enclosure. If tampered with, the HSM will automatically wipe all sensitive security parameters, cryptographic keys, and so on. Regardless, always ensure that HSMs are stored in secure facilities. In terms of secure HSM access, HSMs are often designed to work with cryptographic tokens for access control and invoking sensitive services. For example, the SafeNet token—called a PED key—allows users to securely access sensitive HSM services (locally and even remotely).

Example HSM vendors include Thales e-Security and SafeNet.

Key escrow

Key escrow is frequently a necessary evil. Given that encrypted data cannot be decrypted if the key is lost, many entities opt to store and backup cryptographic keys, frequently offsite, to use at a later time. Risks associated with key escrow are simple; making copies of keys and storing them in other locations increases the attack surface of the data protection. A compromised, escrowed key is just as impactful as compromise of the original copy.

Key lifetime

Key lifetime refers to how long a key should be used (actually encrypting, decrypting, signing, MACing, and so on.) before being destroyed (zeroized).
In general, asymmetric keys (for example, PKI certificates) can be used for much longer periods of time given their ability to be used for establishing fresh, unique session keys (achieving perfect forward secrecy). Symmetric keys, in general, should have much shorter key lifetimes. Upon expiration, new keys can be provisioned in myriad ways:

- Transported by a central key management server or other host (key transport, using algorithms such as AES-WRAP—the AES-WRAP algorithm encrypts the key being transported and as such the AES-WRAP key makes use of a KEK)

- Securely embedded in new software or firmware
- Generated by the device (for example, by a NIST SP800-90 DRBG)
- Mutually established by the device with another entity (for example, Elliptic Curve Diffie Hellman, Diffie Hellman, MQV)
- Manually entered into a device (for example, by typing it in or electronically squirting it in from a secure key loading device)

Key zeroization

Unauthorized disclosure of a secret or private cryptographic key or algorithm state effectively renders the application of cryptography useless. Encrypted sessions can be captured, stored, then decrypted days, months, or years later if the cryptographic key used to protect the session is acquired by a malicious entity.

Securely eradicating cryptographic keys from memory is the topic of zeroization. Many cryptographic libraries offer both conditional and explicit zeroization routines designed to securely wipe keys from runtime memory as well as long term static storage. If your IoT device(s) implement cryptography, they should have well-vetted key zeroization strategies. Depending on the memory location, different types of zeroization need to be employed. Secure wiping, in general, does not just dereference the cryptographic key (that is, setting a pointer or reference variable to null) in memory; zeroization must actively overwrite the memory location either with zeroes (hence the term zeroization) or randomly generated data. Multiple overwrites may be necessary to sufficiently render the crypto variables irretrievable from certain types of memory attacks (for example, freezing memory). If an IoT vendor is making use of cryptographic libraries, it is imperative that proper use of its APIs is followed, including zeroization of all key material after use (many libraries do this automatically for session-based protocols such as TLS).

Disposal of IoT devices containing highly sensitive PII data may also need to consider active destruction of memory devices. For example, hard drives containing classified data have been degaussed in strong electromagnetic fields for years to remove secret and top secret data and prevent it from falling into the wrong hands. Mechanical destruction sufficient to ensure physical obliteration of memory logic gates may also be necessary, though degaussing and mechanical destruction are generally necessary only for devices containing the most sensitive data, or devices simply containing massive amounts of sensitive data (for example, hard drives and SSD memory containing thousands or millions of health records or financial data).

Zeroization is a topic some readers may know more about than they think. The recent (2016) conflict between the US Federal Bureau of Investigation and Apple brought to light the FBI's limitation in accessing a terrorist's iPhone without its contents (securely encrypted) being made irretrievable. Too many failed password attempts would trigger the zeroization mechanism, rendering the data irretrievable.

Accounting and management

Identifying, tracking, and accounting for the generation, distribution, and destruction of key material between entities is where accounting and management functions are needed.

It is also important to balance security and performance. This is realized when establishing cryptographic key lifetimes, for example. In general, the shorter the key lifetime, the smaller the impact of a compromise, that is, the less data surface dependent on the key. Shorter lifetimes, however, increase the relative overhead of generating, establishing, distributing, and accounting for the key material. This is where public key cryptography—that enables forward secrecy—has been invaluable. Asymmetric keys don't need to be changed as frequently as symmetric ones. They have the ability to establish a new, fresh set of symmetric keys on their own. Not all systems can execute public key algorithms, however.

Secure key management also requires vendors to be very cognizant of the cryptographic key hierarchy, especially in the device manufacturing and distribution process. Built-in key material may emanate from the manufacturer (in which case, the manufacturer must be diligent about protecting these keys), overwritten, and used or possibly discarded by an end user. Each key may be a prerequisite for transitioning a device to a new state or deploying it in the field (as in a bootstrapping or enrollment process). Cryptographic-enabled IoT device manufacturers should carefully design and document the key management processes, procedures, and systems used to securely deploy products. In addition, manufacturer keys should be securely stored in HSMs within secure facilities and access-controlled rooms.

Access controls to key management systems (for example, HSMs and HSM-connected servers) must be severely restricted given the large ramifications of the loss or tampering of even one single cryptographic key. One will often find key management systems—even in the most secure facility or data center—housed within a cage under lock and key and continuous camera surveillance.

Summary of key management recommendations

Given the above definitions and descriptions, IoT vendors and system integrators should also consider the following recommendations with regard to key management:

- Ensure that validated cryptographic modules securely store provisioned keys within IoT devices—physical and logical protection of keys in a secure trust store will pay security dividends.
- Ensure that cryptographic keys are sufficiently long. An excellent guide is to refer to NIST SP 800-131A (`http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdff`), which provides guidance on appropriate key lengths to use for FIPS-approved cryptographic algorithms. If interested in equivalent strengths (computational resistance to brute forcing attacks), one can reference NIST SP800-57. It is important to sunset both algorithms and key lengths when they are no longer sufficiently strong relative to state-of-the-art attacks.
- Ensure that there are technical and procedural controls in place to securely wipe (zeroize) cryptographic keys after use or expiration. Don't keep any key around any longer than is necessary. Plaintext cryptographic variables are known to exist in memory for long periods after use unless actively wiped. A well-engineered cryptographic library may zeroize keys under certain circumstances, but some libraries leave it to the using application to invoke the zeroization API when needed. Session based keys, for example, the ciphering and HMAC keys used in a TLS session, should be immediately zeroized following termination of the session.
- Use cryptographic algorithms and protocol options in a manner that **perfect forward secrecy** (**PFS**) is provided. PFS is an option in many communication protocols that utilize key establishment algorithms such as Diffie Hellman and Elliptic Curve Diffie Hellman. PFS has the beneficial property that a compromise of one set of session keys doesn't compromise follow-on generated session keys. For example, utilizing PFS in DH/ECDH will ensure that ephemeral (one time use) private/public keys are generated for each use. This means that there will be no backward relationship between adjacent shared secret values (and therefore the keys derived from them) from session to session. Compromise of today's key will not allow forward, adversarial computation of tomorrow's key, thus tomorrow's key is better protected.

- Severely restrict key management system roles, services, and accesses. Access to cryptographic key management systems must be restricted both physically and logically. Protected buildings and access-controlled rooms (or cages) are important for controlling physical access. User or administrator access must also be carefully managed using principles such as separation of duties (not giving one single role or identity full access to all services) and multi-person integrity (requiring more than one individual to invoke sensitive services)
- Use well vetted key management protocols to perform primitive key management functions such as key transport, key establishment, and more. Being an arcane topic, and the fact that many vendors utilize proprietary solutions, there are few key management protocols commonly deployed today. The OASIS group, however, maintains a relatively recently designed industry solution called the **key management interoperability protocol** (**KMIP**). KMIP is now in use by a number of vendors as a simple backbone protocol for performing sender-receiver key management exchanges. It supports a number of cryptographic key management algorithms and was designed keeping multi-vendor interoperability in mind. KMIP is programming language agnostic and useful in everything from large enterprise key management software to embedded device management.

Examining cryptographic controls for IoT protocols

This section examines cryptographic controls as integrated into various IoT protocols. Lacking these controls, IoT point-to-point and end-to-end communications would be impossible to secure.

Cryptographic controls built into IoT communication protocols

One of the primary challenges for IoT device developers is understanding the interactions between different types of IoT protocols and the optimal approach for layering security across these protocols.

There are many options for establishing communication capabilities for IoT devices and often these communication protocols provide a layer of authentication and encryption that should be applied at the link layer. IoT communication protocols such as ZigBee, ZWave, and Bluetooth-LE all have configuration options for applying authentication, data integrity, and confidentiality protections. Each of these protocols supports the ability to create wireless networks of IoT devices. Wi-Fi is also an option for supporting the wireless link required for many IoT devices and also includes inherent cryptographic controls for maintaining confidentiality, integrity and authentication.

Riding above the IoT communication protocols are data-centric protocols. Many of these protocols require the services of lower layer security capabilities, such as those provided by the IoT communication protocols or security-specific protocols such as DTLS or SASL. IoT data centric protocols can be divided into two categories that include REST-type protocols such as CoAP and publish/subscribe protocols such as DDS and MQTT. These often require an underlying IP layer; however, some protocols, such as MQTT-SN, have been tailored to operate on RF links such as ZigBee.

An interesting aspect of publish/subscribe IoT protocols is the need to provide access controls to the topics that are published by IoT resources, as well as the need to ensure that attackers cannot publish unauthorized information to any particular topic. This can be handled by applying unique keys to each topic that is published.

ZigBee

ZigBee leverages the underlying security services of the IEEE 802.15.4 MAC layer. The 802.15.4 MAC layer supports the AES algorithm with a 128-bit key for both encryption/decryption as well as data integrity by appending a MAC to the data frame (`http://www.libelium.com/security-802-15-4-zigbee/`). These security services are optional, however, and ZigBee devices can be configured to not use either the encryption or MAC capabilities built into the protocol. In fact, there are multiple security options available as described in the following table:

ZigBee security configuration	**Description**
No security	No encryption and no data authentication
AES-CBC-MAC-32	Data authentication using a 32-bit MAC; no encryption
AES-CBC-MAC-64	Data authentication using a 64-bit MAC; no encryption
AES-CBC-MAC-128	Data authentication using a 128-bit MAC; no encryption
AES-CTR	Data is encrypted using AES-CTR with 128-bit key; no authentication

ZigBee security configuration	Description
AES-CCM-32	Data is encrypted and data authentication using 32-bit MAC
AES-CCM-64	Data is encrypted and data authentication using 64-bit MAC
AES-CCM-128	Data is encrypted and data authentication using 128-bit MAC

The 802.15.4 MAC layer in the preceding table, ZigBee supports additional security features that are integrated directly with the layer below. ZigBee consists of both a network layer and an application layer and relies upon three types of keys for security features:

- Master keys, which are pre-installed by the vendor and used to protect a key exchange transaction between two ZigBee nodes
- Link keys, which are unique keys per node, allowing secure node-to-node communications
- Network keys, which are shared across all ZigBee nodes in a network and provisioned by the ZigBee trust center; these support secure broadcast communications

Setting up the key management strategy for a ZigBee network can be a difficult challenge. Implementers must weigh options that run the spectrum from pre-installing all keys or provisioning all keys from the trust center. Note that the trust center default network key must always be changed and that any provisioning of keys must occur using secure processes. Key rotation must also be considered since ZigBee keys should be refreshed on a pre-defined basis.

There are three options for ZigBee nodes to obtain keys. First, nodes can be pre-installed with keys. Second, nodes can have keys (except for the master key) transported to them from the ZigBee Trust Center. Finally, nodes can establish their keys using options that include **symmetric key establishment** (**SKKE**) and **certificate-based key establishment** (**CBKE**) (`https://www.mwrinfosecurity.com/system/assets/849/original/mwri-zigbee-overview-finalv2.pdf`).

Master keys support the generation of link keys on ZigBee devices using the SKKE process. Link keys shared between a ZigBee node and the trust center are known as **trust center link keys** (**TCLK**). These keys allow the transport of a new network key to nodes in the network. Link and network keys can be pre-installed; however, the more secure option is to provide for key establishment for link keys that support node-to-node communications.

Network keys are transmitted in an encrypted APS transport command from the trust center.

Although link keys are optimal for node-to-node secure communication, research has shown that they are not always optimal. They require more memory resources per device, something often not available for IoT devices (`http://www.libelium.com/security-802-15-4-zigbee/`).

The CBKE process provides another mechanism for ZigBee link key establishment. It is based on an **Elliptic Curve Qu-Vanstone** (**ECQV**) implicit certificate that is tailored towards IoT device needs; it is much smaller than a traditional X.509 certificate. These certificates are called implicit certificates and their structure provides a significant size reduction as compared to traditional explicit certificates such as X.509 (this is a nice feature in constrained wireless networking) (`http://arxiv.org/ftp/arxiv/papers/1206/1206.3880.pdf`).

Bluetooth-LE

Bluetooth-LE is based on the Bluetooth Core Specification Version (4.2) and specifies a number of modes that provide options for authenticated or unauthenticated pairing, data integrity protections, and link encryption. Specifically, Bluetooth-LE supports the following security concepts (reference: Bluetooth Specification, Version 4.2):

- **Pairing**: Devices create one or more shared secret keys
- **Bonding**: The act of storing the keys created during pairing for use in subsequent connections; this forms a trusted device pair
- **Device authentication**: Verification that the paired devices have trusted keys
- **Encryption**: Scrambling of plaintext message data into ciphertext data
- **Message integrity**: Protects against tampering with data

Bluetooth-LE provides four options for device association:

Model	Details
Numeric comparison	The user is shown a six-digit number and enters *YES* if the numbers are the same on both devices. Note that with Bluetooth 4.2 the six-digit number is not associated with the encryption operations between the two devices.
Just works	Designed for devices that do not include a display. Uses the same model as numeric comparison however the user is not shown a number.
Out of band	Allows use of another protocol for secure pairing. Often combined with **near-field communications** (**NFC**) to allow for secure pairing. In this case, the NFC protocol would be used to exchange the device Bluetooth addresses and cryptographic information.
Passkey entry	Allows a six-character passkey to be entered on one device and displayed on another for confirmation.

Bluetooth-LE makes use of a number of keys that are used together to provide the requested security services. The following table provides a view into the cryptographic keys that play a role in Bluetooth-LE security.

Key type	Description
Temporary key (**TK**)	Determined by the type of Bluetooth pairing used, the TK can be different lengths. It is used as an input to the cipher-based derivation of the **short-term key** (**STK**).
Short-term key (**STK**)	STK is used for secure distribution of key material and is based on the TK and a set of random values provided by each device participating in the pairing process.
Long-term key (**LTK**)	The LTK is used to generate a 128-bit key employed for link-layer encryption.
Connection signature resolving key (**CSRK**)	The CSRK is used for signing data at the ATT layer.
Identity resolving key (**IRK**)	The IRK is used to generate a private address based on a device public address. This provides a mechanism for device identity and privacy protection.

Bluetooth-LE supports cryptographically signed data through the use of the CSRK. The CSRK is used to apply a signature to a Bluetooth-LE **protocol data unit** (**PDU**). The signature is a MAC that is generated by the signing algorithm and a counter that increments for each PDU sent. The addition of the counter provides additional replay protections.

Bluetooth-LE also supports the ability to provide privacy protections for devices. This requires the use of the IRK which is used to generate a special private address for the device. There are two options available for privacy support, one where the device generates the private address and one where the Bluetooth controller generates the address.

Near field communication (NFC)

NFC does not implement native cryptographic protection; however, it is possible to apply endpoint authentication across an NFC negotiation. NFC supports short-range communication and is often used as a first-step protocol to establish out-of-band pairings for use in other protocols, such as Bluetooth.

Cryptographic controls built into IoT messaging protocols

We will discuss here the various controls that are built into the messaging protocols.

MQTT

MQTT allows sending a username and password. Until recently, the specification recommended that passwords be no longer than 12 characters. The username and password are sent in the clear as part of the CONNECT message. As such it is critical that TLS be employed when using MQTT to prevent MITM attacks on the password. Ideally, end-to-end TLS connectivity between the two endpoints (vice gateway-to-gateway) should be used along with certificates to mutually authenticate the TLS connection.

CoAP

CoAP supports multiple authentication options for device-to-device communication. This can be paired with Datagram TLS (D-TLS) for higher-level confidentiality and authentication services.

CoAP defines multiple security modes based on the types of cryptographic material used: `https://tools.ietf.org/html/rfc7252#section-9`.

Mode	Description
`NoSec`	There is no protocol-level security as DTLS is disabled. This mode *may* be sufficient if used in cases where alternate forms of security can be enabled, for example, when IPsec is being used over a TCP connection or when a secure link layer is enabled; however, the authors do not recommend this configuration.
`PreSharedKey`	DTLS is enabled and there are pre-shared keys that can be used for nodal communication. These keys may also serve as group keys.
`RawPublicKey`	DTLS is enabled and the device has an asymmetric key pair without a certificate (a raw public key) that is validated using an out-of-band mechanism. The device also has an identity calculated from the public key and a list of identities of the nodes it can communicate with.
`Certificate`	DTLS is enabled and the device has an asymmetric key pair with an X.509 certificate (RFC5280) that binds it to its subject and is signed by some common trust root. The device also has a list of root trust anchors that can be used for validating a certificate.

DDS

The Object Management Group's **Data Distribution Standard** (**DDS**) security specification provides endpoint authentication and key establishment to enable message data origin authentication (using HMAC). Both digital certificates and various identity/authorization token types are supported.

REST

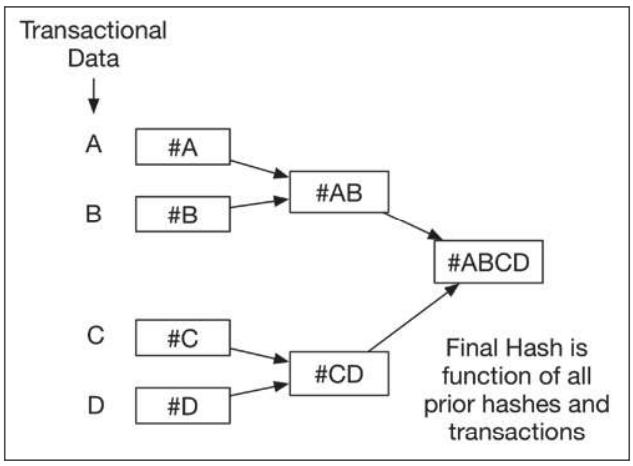
HTTP/REST typically requires the support of the TLS protocol for authentication and confidentiality services. Although basic authentication (where credentials are passed in the clear) can be used under the cover of TLS, this is not a recommended practice. Instead, attempt to stand up a token-based authentication (and authorization, if needed) approach such as OpenID identity layer on top of OAuth2. Additional security controls should be in place when using OAuth2, however. References for these controls can be found at the following websites:

- `http://www.oauthsecurity.com`
- `https://www.sans.org/reading-room/whitepapers/application/attacks-oauth-secure-oauth-implementation-33644`

Future directions of the IoT and cryptography

The cryptography used in the IoT today comprises the same cryptographic trust mechanisms used in the broader Internet. Like the Internet, however, the IoT is scaling to unprecedented levels that require far more distributed and decentralized trust mechanisms. Indeed, many of the large-scale, secure IoT transactions of the future will not be made of just simple client-server or point-to-multipoint cryptographic transactions. New or adapted cryptographic protocols must be developed and added to provide scalable, distributed trust. While it is difficult to predict what types of new protocols will ultimately be adopted, the distributed trust protocols developed for today's Internet applications may provide a glimpse into where things may be going with the IoT.

One such protocol is that of blockchain, a decentralized cryptographic trust mechanism that underlies the Bitcoin digital currency and provides a decentralized ledger of all legitimate transactions occurring across a system. Each node in a blockchain system participates in the process of maintaining this ledger. This is accomplished automatically through trusted consensus across all participants, the results of which are all inherently auditable. A blockchain is built up over time using cryptographic hashes from each of the previous blocks in the chain. As we discussed earlier in this chapter, hash functions allow one to generate a one-way fingerprint hash of an arbitrary chunk of data. A Merkle tree represents an interesting application of hash functions, as it represents a series of parallel-computed hashes that feed into a cryptographically strong resultant hash of the entire tree.

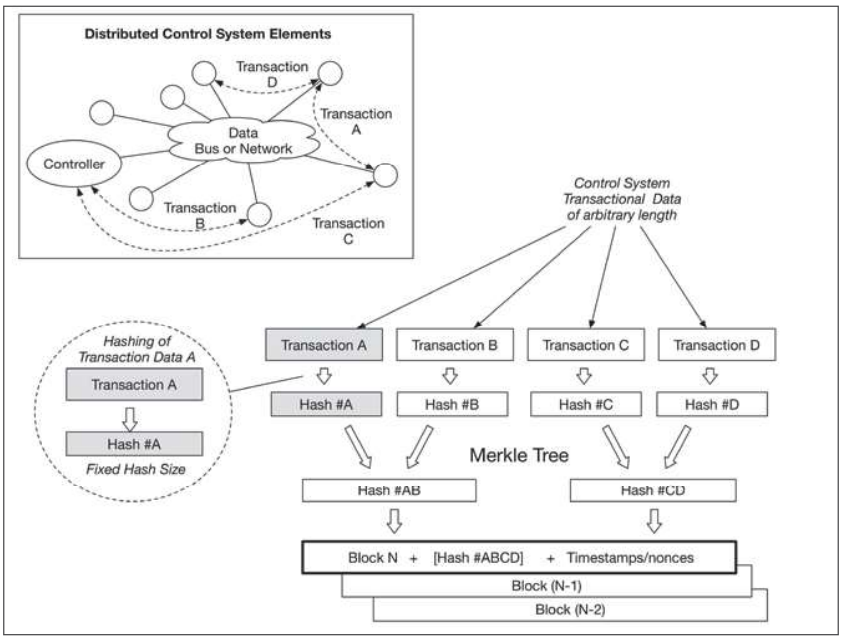


merkle-tree.graffle

Corruption or integrity loss of any one of the hashes (or data elements that were hashed) provides an indication that integrity was lost at a given point in the Merkle tree. In the case of blockchain, this Merkle tree pattern grows over time as new transactions (nodes representing hashable transactions) are added to the ledger; the ledger is available to all and is replicated across all nodes in the system.

Blockchains include a consensus mechanism that is used by nodes in the chain to agree upon how to update the chain. Considering a distributed control system, for example, a controller on a network may want to command an actuator to perform some action. Nodes on the network could potentially work together to agree that the controller is authorized to command the action and that the actuator is authorized to perform the action.

An interesting twist on this, however, is that the blockchain can be used for more than this base functionality. For example, if the controller typically receives data from a set of sensors and one of the sensors begins to provide data that is not within norms or acceptable tolerances (using variance analysis for instance), the controller can update the blockchain to remove authorizations from the wayward sensor. The update to the blockchain can then be hashed and combined with other updated (for example, transactions) hashes through a Merkle tree. The resultant would then be placed in the proposed new block's header, along with a timestamp and the hash of the previous block.



blockchain-trust.graffle

This type of solution may begin to lay the groundwork for resilient and fault-tolerant peer-to-peer networks within distributed, trusted CPS. Such functionality can be achieved in real time and near-real time use cases with appropriate performance requirements and engineering. Legacy systems can be augmented by layering the transactional protocols in front of the system's control, status, and data messages. While we don't ultimately know how such techniques may or may not be realized in future IoT systems, they offer us ideas on how to employ powerful cryptographic algorithms to solve the enormous challenges of ensuring distributed trust at a large scale.

Summary

In this chapter, we touched on the enormously large and complex world of applied cryptography, cryptographic modules, key management, cryptographic application in IoT protocols, and possible future looks into cryptographic enablement of distributed IoT trust in the form of blockchain technology.

Perhaps the most important message of this chapter is to take cryptography and its methods of implementation seriously. Many IoT devices and service companies simply do not come from a heritage of building secure cryptographic systems and it is unwise to consider a vendor's hyper-marketed claims that their "256 bit AES" is secure. There are just too many ways to thwart cryptography if not properly implemented.

In the next chapter, we will dive into **identity and access management** (**IAM**) for the IoT.

6

Identity and Access Management Solutions for the IoT

While society begins to adopt smart home and IoT wearables, IoT devices and applications are diversifying toward broader application in professional, government, and other environments as well. The network connectivity needed to support them is becoming ubiquitous and to that end devices will need to be identified and access provisioned in new and different environments and organizations. This chapter provides an introduction to identity and access management for IoT devices. The identity lifecycle is reviewed and a discussion on infrastructure components required for provisioning authentication credentials is provided, with a heavy focus on PKI. We also examine different types of authentication credentials and discuss new approaches to providing authorization and access control for IoT devices. We address these subjects in the following topic areas:

- Introductory discussion on **identity and access management** (**IAM**)
- Discussion of the identity lifecycle
- A primer on authentication credentials
- Background on IoT IAM infrastructure
- A discussion of IoT authorization and access control