Cryptography in industrial embedded systems: our experience of needs and constraints

Jean-Philippe Aumasson, Antony Vennard

Teserakt AG

A common model is that industrial embedded systems (a.k.a. IoT, M2M, etc.) need small, fast, low-energy crypto primitives—requirements often summarized by the "lightweight" qualificative. For example, a premise of NIST's Lightweight Cryptography standardization project is that AES is not lightweight enough, and more generally that "the majority of current cryptographic algorithms were designed for desktop/server environments, many of these algorithms do not fit into constrained devices"—note the implicit emphasis on size.

In this article we share some observations from our experience working with industrial embedded systems in various industries, on various platforms and using various network protocols. We notably challenge the truism that small devices need small crypto, and argue that finding a suitable primitive is usually the simplest task that engineers face when integrating cryptography in their products. This article reviews some of the other problems one has to deal with when deploying cryptography mechanism on "lightweight" platforms.

We're of course fatally subject to selection bias, and don't claim that our perspective should be taken as a reference or authoritative. We nonetheless hope to contribute to a better understanding of what are the "constrained devices" that NIST refers to, and more generally of "real-world" cryptography in the context of embedded systems.

Few details can be share, alas, about our experience. Let us only say that we've designed, integrated, implemented, or reviewed cryptographic components in systems used in automotive, banking, content protection, satellite communications, law enforcement technology, supply chain management, device tracking, or healthcare.

AES is lightweight enough

Most of the time. Ten years ago we worked on a low-cost RFID product that had to use something else that AES, partially for performance reasons. Today

many RFID products include an AES engine, as specified for example in the norm ISO/IEC 29167-10.

Systems on chips and boards for industrial application often include AES hardware, and when they don't a software implementation of AES comes at acceptable costs. For example, chips from the very popular STM32 family generally provide readily available AES in different modes of operations.

An example perhaps of a very low-cost device that uses a non-AES primitive would be the Multos Step/One, which is EMVCo-compliant. In this case, 3DES was chosen instead. ATM Encrypting-Pin-Pads frequently use 3DES. We believe that the continued use of 3DES has more to do with compatibility and cost of replacement than the prohibitive cost of running a wider block cipher.

Of course an algorithm more compact and faster than AES wouldn't hurt, but the benefits would have to justify the integration costs.

Choosing primitives is a luxury

More than once we faced the following challenge: create a data protection mechanism given the cryptography primitives available on the devices, which could for example be AES-GCM and SHA-256 only. We may have to use AES-GCM and SHA-256 because they're standards, because they're efficiently implemented (for example through hardware accelerators), or for the sake of interoperability. Note that a platform may give you access to AES-GCM, but not to the AES core directly, so you can't use it to implement (say) AES-SIV.

If you want to use another algorithm than the ones available, you have to justify that it's worth the cost of implementing, integrating and testing the new primitive. AES-GCM is not perfect (risk of nonce reuse, etc.), but the risk it creates is usually negligible compared to other risks. The situation may be different with AES-ECB.

Statelessness

Not all platforms are stateful, or reliably stateful. This means that you can't always persistently store a counter, seed, or other context-dependent on the device. The software/firmware on the platform may be updatable over-the-air, but not always. The keys stored on the device may not be modifiable after the personalization phase of the production cycle.

Randomness

The platform may not offer you a reliable pseudorandom generator, or it may only have some low-entropy non-cryptographic generator, anyway that can be

a severe limitation, especially if you realize this after proudly completing an implementation of ECDSA.

There are well-known workarounds of course, such as deterministic ECDSA and EdDSA for ECC signatures, or AES-SIV for authenticated encryption (this robustness to weak/non-randomness is the main reason why we chose to make it the default cipher in our company's product). But sometimes it can get trickier, when you really need some kind of randomness yet can't fully trust the PRNG (it's more fun when the platform is stateless).

It's not only about (authenticated) encryption

When no established standard such as TLS is used—and sometimes even when it is—the security layer is typically implemented at the application layer between the transport and business logic. (Authenticated) encryption is a typical requirement, but seldom the only one: you may have to worry about replay attacks or have to "obfuscate" some metadata or header information, for example to protect anonymity.

In an ideal world, you ought to use a thoroughly-designed, provably-secure, peer-reviewed protocol. But 1) such a thing likely doesn't exist, and 2) even when it does it would probably not be suitable to your use case, for example if the said protocol requires a trusted third party or three network round-trips.

Message size limitations

True story: "Our clear payload is N bytes; the protected payload must be N bytes too, and must be encrypted and authenticated." That's when you have to be creative. Such a situation can occur with protocols such as Bluetooth Low-Energy or WAN protocols such as LoRaWAN or Sigfox (where the uplink payloads are 12 bytes and downlink payloads 8 bytes).

Even when you can afford some overhead to send a nonce and a tag, this may come at prohibitive cost if we're talking of millions of messages and a per-volume pricing model. In other contexts, additional payload size can increase the risk of packet loss.

Network unreliability

It's not just about TCP being reliable (guaranteed, in-order packet delivery) and UDP being not. Protocols running on top of TCP can have their own reliability properties caused by the way they transmit messages. For example, MQTT (when running over TCP) guarantees message delivery, but not in-order.

Whatever protocol is used, devices may have no way to transmit nor receive messages for a certain period of time. For example, communicating with satellites in non-geostationary orbit, or devices that are out of range for periods of time, such as aircraft, ships or smart meters as a measuring driver passes by.

An excellent engineering question is to ask how one would reliably transmit data from Mars, particularly where that data should be processed as quickly as possible on receipt. If not Mars, then further away. At such great distances, what is instantaneous for us starts to take seconds or minutes of time. The answer is to resend on a broadcast channel repeatedly, usually as an illustrative example of UDP/broadcast protocols. In these cases, packets are expected to be lost and round-trips are impossible—there is no way to confirm receipt.

Such limitations often prevent the use of crypto protocols adding RTTs, or requiring even a moderate level of synchronization with other devices. Unreliable network becomes particularly fun when implementing key rotation or key distribution mechanisms.

When crypto is too big

Crypto can be too big (in code size, or RAM usage) for certain platforms; the main cases we've encountered are when public-key operations are impossible, or when a TLS implementation takes too much resources. Although there are good TLS implementations for constrained platforms (such as ARM's mbedTLS, wolfSSL, or BearSSL), they may include a lot of code to support the TLS standards and operations such as parsing certificates. Even the size of a TLS-PSK stack can prohibitive—and not because of AES.

Sometimes public-key cryptography is even possible within the limited capacity of the device, but the limiting factor is the protocol. An example of such a problem can be found in BearSSL's documentation. Quoting Thomas Pornin's TLS 1.3 Status, when streaming ASN.1 certificates, the usual order is end-entity first, CAs later. In any given object, the public key follows the certificate. EdDSA combines the public key and data when signing. Thus, in order to validate a signature, the entire certificate must be buffered until the public key can be extracted.

Now while a highly constrained device may simply use PSK and avoid this problem entirely, it is also true that the device may be capable of Ed25519 signatures even without sufficient RAM to buffer large certificates. This problem arises entirely from the choice of PureEdDSA rather than HashEdDSA in TLS 1.3.

Untrusted infrastructure

More often than you might expect, the infrastructure we use should not be trusted. In the MQTT context, this means brokers. In other context this means wireless repeaters, conversion gateways between protocols such as communication via SMS and so on. In the context of currently proposed IoT standards, these nodes are often assumed trusted and capable of re-encrypting for each hop using TLS where possible, or some other point-to-point protocol.

We believe that the implicit trust in the infrastructure by having it handle keys invites a far greater risk than the challenges of underpowered devices.

Conclusions

Cryptography on constrained devices can pose many problems, but the speed and size of symmetric primitives (ciphers, hash functions) is rarely one, at least in our experience (YMMV).

We can't ignore that economics and risk management play into cryptography. Standard NIST cryptography primitives and NSA Suites A & B, for example, were designed to provide the US Government with an assurance that data is protected for the lifetime of the relevant classified information—on the order of magnitude of 50 years. It took time for the community to gain confidence in AES, but it's now widely and globally trusted—anyway, safe block ciphers are easy to design; even DES nor GOST have never really been broken.

Lightweight cryptography might be suitable where such expectations of long-term security do not hold, and would allow the use of a very "lightweight" component. An extreme example is that or memory encryption, or "scrambling", where only a handful of high-frequency cycles can be allocated.

The open question is whether we can design algorithms to match this requirement, bearing in mind that we have no ability to predict future developments. Looking back at history, requirements are driven by applications on which the public research community has little view. As highlighted in this article, said requirements often involve various components and technologies, which make the engineering problem difficult to approach to outsiders.