Lecture Notes

Chapter 8:
Choosing and Optimizing Cryptographic Algorithms for Resource-Constrained Systems

CYENG 351: Embedded Secure Networking

Instructor: Dr. Shayan (Sean) Taheri

Gannon University (GU)

Overview

- Cryptography presents the biggest challenge to any security solution to be implemented for an embedded device because of the requirements of these computationally complex algorithms.
- Cryptography is notoriously expensive when it comes to clock cycles, and many algorithms are not too friendly to smaller memory devices.
- Some applications can actually get away without using traditional cryptography.
- These applications utilize other mechanisms, such as hashing algorithms, in order to provide some assurance about data.
- The big advantage here is that hashing algorithms are typically many orders of magnitude faster than symmetric or public-key cryptography.
- One classification that will help us make the decision is whether or not we care about eavesdropping.
- If we only care that the data is reliable, and do not care who gets it, we can definitely avoid cryptography.
- For those applications that require absolute secrecy, there are many options that will help the embedded developer meet the goals of performance, cost, and security.
- RSA and other public-key algorithms are extremely slow, requiring hardware assistance on many platforms - not just small embedded systems. → However, these algorithms are essential to certain protocols (primarily SSL and SSH).

Do We Need Cryptography?

- One of the first steps in building a secure embedded system is to see if cryptography is actually needed.
- Whenever security is discussed, many engineers will immediately think of cryptography as the solution, when in fact, many options may exist that do not strictly require cryptography.
- Cryptography is an important part of securing applications, but it is not a security panacea, nor is it always necessary for building a secure system.
- A stream of inaccurate portrayals of cryptography in Hollywood and television combined with unrelenting advertising campaigns for various security products have served to only keep those not involved directly in computer science in the dark.
- Given the confusion surrounding cryptography, it can be extremely difficult for a systems engineer to determine what type of cryptography, if any, is needed.
- The truth is that in some circumstances, no cryptography is needed at all.
- We can just use a simple hash algorithm to verify the integrity of the data being transported.
- Not being a full-blown encryption scheme, the system's requirements can be lowered to support just the hashing (or those resources could be used for improving the performance of other parts of the system).
- In order to see what type of security an application will need, we can divide applications into several distinct categories based upon the type of information each application deals with.

Do We Need Cryptography? (Cont.)

- The following categories should be sufficient for most applications, ordered from the lowest level of security to the highest level of security:
  - No security required (may be optional)—applications such as streaming video, noncritical data monitoring, and applications without networking in a controlled environment.
  - Low-level security (hashing only, plaintext data)—applications delivering publicly available information such as stock market data, data monitoring, networked applications in a controlled environment, and applications requiring an extra level of robustness without concern about eavesdropping. Another example of this would be the distribution of (usually open-source) source code or executables with an accompanying hash value for verifying integrity, usually an MD5 or SHA-1 hash of the fi les that is to be calculated by the end user and compared to the provided hash.
  - Low-medium security (hashing plus authentication)—general industrial control, some web sites, and internal corporate network communications.
  - Medium-level security (cryptography such as RC4 with authentication, small key sizes)—applications dealing with general corporate information, important data monitoring, and industrial control in uncontrolled environments.

Do We Need Cryptography? (Cont.)

- The following categories should be sufficient for most applications, ordered from the lowest level of security to the highest level of security:
  - Medium-high security (SSL with medium key sizes, RSA and AES, VPNs)—applications dealing with e-commerce transactions, important corporate information, and critical data monitoring.
  - High-level security (SSL with maximum-sized keys, VPNs, guards with guns)—applications dealing with important financial information, noncritical military communications, medical data, and critical corporate information.
  - Critical-level security (physical isolation, maximum-size keys, dedicated communications systems, one-time pads, guards with really big guns)—used to secure information such as nuclear launch codes, root Certificate Authority private keys (hopefully!), critical financial data, and critical military communications.
  - Absolute security (one-time pads, possibly quantum cryptography, guards with an entire military behind them)—used to secure information including Cold War communications between the United States and the Soviet Union regarding nuclear weapons, the existence of UFOs, the recipe for Coca-Cola, and the meaning of life.

Do We Need Cryptography? (Cont.)

- The above categories are intended as a rule-of-thumb, not as strict guidelines for what level of security you will want for a particular application.
- Obviously, your particular application may fit into one of the above categories and require a higher level of security than indicated.
- If your application falls into the medium-high category or higher, you will definitely want to do some serious research before jumping into implementation.
- In fact, you should do a lot of research if your application needs security at all.

Hashing - Low Security, High Performance

- Ideally, we would not need any security in our applications at all, instead having the luxury to focus all resources on performance.
- Unfortunately, this is not reality and we have to balance security with performance.
- Fortunately, the security-performance tradeoff is relatively linear—higher security means some performance loss, but the balance can be adjusted fairly easily by choosing different algorithms, methods, and key sizes.
- For the best performance with some level of security, you can't beat the hash algorithms.
- As far as resources are concerned, they use far less than other types of cryptographic algorithms.
- Not strictly cryptography in the classic sense, hash algorithms provide integrity without protection from eavesdropping - in many applications this is all the security that is really needed.
- How do we actually make hashing work for us in our applications? → Probably the most common use of hashing is to provide a guarantee of integrity.
- The concept behind cryptographically secure hashing is that hash collisions are very difficult to create.
- Therefore, if you hash some information and compare it to a hash of that information provided to you by a trusted source and the hashes match, you have a fairly high certainty that the information has not changed or been tampered with in transit.

Hashing - Low Security, High Performance (Cont.)

- Another mechanism that employs hashing to provide a level of security is called “challenge-response,” after the way the mechanism works.
- In a challenge-response operation, the system or user requesting the information provides a “challenge” to the sender of the requested information.
- The sender must then provide an appropriate “response” to the requestor, or the information is deemed invalid.
- The challenge is typically some cryptographically secure random number that is difficult to predict, and it is sent by the client to the server (or from the server to the client, as is the case with some HTTP applications).
- Once the server receives the challenge, it calculates a hash of the challenge and the secret, and sends the hash back to the client.
- The client also performs the same operation on its copy of the secret, so when it receives the hash it will know that the server knows the same secret.
- The reason the mechanism works is because the server could not produce the correct hash value unless it knew the secret.
- The challenge-response hashing described here does suffer from a man-in-the-middle vulnerability, since an attacker could intercept the message in transit from the server back to the client.
- This attack requires the attacker to have access to the routing system of the network in order to spoof the address of the server and the client so it can intercept and retransmit the messages.

Hashing - Low Security, High Performance (Cont.)

- There are several hashing algorithms available, but the most common are MD5 and SHA-1.
- Unfortunately, the MD5 algorithm has been shown to have some weaknesses, and there is a general feeling of uneasiness about the algorithm.
- SHA-1 has been the focus of some suspicion as well, but it has not had the same negative press received by MD5.
- In any case, both algorithms are still heavily used in the absence of anything better.
- In practice, many embedded development tool suites provide libraries for MD5, SHA-1, or both hashing algorithms.
- Hashing algorithms are fairly easy to optimize as well, so it is quite likely that the provided implementations will already be fairly optimal for your target hardware.

Hashing - Low Security, High Performance (Cont.)

- Using hash algorithms is quite easy, since they have only three basic operations that are provided in the user API:
  1. Initialization, which sets up the state data structure used to actually perform the hash.
  2. Hashing, which operates on the incoming data, typically in raw bytes (may also be text).
  3. Finalization, which finishes up the hash, and copies the result into an output buffer.
- The basic operation of hashing algorithms uses a buffer, in C an array of char type, as a workspace for the hashing.
- The algorithms utilize a structure that provides state information across hashing operations, allowing for the hash to be added to in multiple operations, rather than all at once in a single hashing operation.

Hashing - Low Security, High Performance (Cont.)

- Hashing is a very simple operation in code.
- Notice that we do some defensive programming when using the strlen function.
- Unfortunately, the C programming language does not have very good standard library support for protecting against buffer overflow.
- In the program example, if the user entered enough data to fill up the buffer to the end (more than 127 characters), we are relying on scanf to be sure that the last element of the array contains a null-terminator.
- In the program, the scanf "%s" type is used in the format string with the optional width format parameter, so it should not cause any issues for the call to strlen later.
- The hashing example illustrates the use of a cryptographic algorithm to protect data, but it also highlights the fact that anything in the program can become a security issue.
- The use of standard C library functions such as strlen can lead to unexpected and unintentional behavior.
- Sometimes this behavior goes unnoticed; sometimes it leads to a crash.
- All it takes is one malicious attacker to find the flaw in your program and exploit it somehow.

Hashing - Low Security, High Performance (Cont.)

```
#include <sha1.h>
#include <stdio.h>
main () {
 char input_buf[128], output_buf[20];
 struct SHA1_state sha_state;
 int i, input_len;
// Initialize the state structure, requires only a
reference to the struct
SHA1_init(&sha_state);
// Get user input, make sure buffer is cleared first
memset(input_buf, 0, sizeof(input_buf));
scanf("%127s", input_buf);
// Hash the input, with a length equal to the size of
the user input. Note that
// the input to the SHA1_hash function can be of any
length
// !!! Danger, strlen can overflow, so we terminate
the buffer for safety
input_buf[127] = 0;
input_len = strlen(input_buf);
 SHA1_hash(&sha_state, input_buf, input_len);
 // Finally, finalize the hash and copy it into a
buffer and display
 SHA1_finish(&sha_state, output_buf);
 for(i = 0; i < 20; ++i) {
 printf("%X ", output_buf[i]);
 }
 printf("\n");
} // End program
```

Hashing with SHA-1

Is Hashing Considered Dangerous?

- In the past few years, cryptography has come into a lot of visibility, and the old faithful algorithms that have served us well for years are now being put to the test.
- The vast amount of information on the public Internet that needs to be protected has lead to a virtual stampede of corporations, governments, organizations, and individuals studying the security mechanisms that form the foundation of that protection.
- People on both sides of the law (and with varying levels of ethics) are racing to discover flaws in the most commonly used algorithms.
- After all, boatloads of money can be made if a security breach is discovered.
- For the "good" guys, the rewards are recognition and (hopefully) prompt fixing of the issue.
- The "bad" guys profit in numerous different ways.
- The end result is always the same, however: If an algorithm is broken, it usually means it's useless from that point on.
- This insane virtual arms race has revealed that it is extremely hard to develop secure cryptographic algorithms (it's easy to write broken cryptographic algorithms), and it appears that hashing may be among the most difficult.
- The two major hash algorithms in use today (notably by SSL and TLS) are MD5 and SHA-1.
- MD5 is considered "mostly broken" and SHA-1 is "sorta broken."

Is Hashing Considered Dangerous? (Cont.)

- There are various ways a hash algorithm could be broken from a cryptographic standpoint:
  - Take two arbitrary but different messages and hash them. If you can easily calculate a hash value that is the same for these different messages (a "hash collision"), then the algorithm is somewhat broken, and potentially seriously broken.
  - Given a hash value, compute an arbitrary message to hash to that value. If this is easy, then the algorithm is a little more broken, since this starts to get into the area where the flaw can be exploited.
  - Generate a meaningful message that generates a hash collision with another meaningful message. If this is easy, then the algorithm is mostly broken, and it is highly likely it provides no security whatsoever. If this is true for an algorithm, it is very easy to fool someone into accepting incorrect information (or worse, damaging information such as a virus or Trojan horse).

Is Hashing Considered Dangerous? (Cont.)

- Each of the above levels of compromise is based on the idea that performing these operations on the algorithm is “hard” (infeasible given current technology and future technology for at least a few years).
- They all feed into one another as well, so if you can find an arbitrary hash collision, it is often easier to discover the other attacks.
- Unfortunately, both MD5 and SHA-1 have been discovered to have vulnerabilities.
- For MD5 there are several demonstrations of ways to generate different meaningful messages that generate the same MD5 hash value.
- MD5 should not be used in new applications whenever possible.
- The MD5 algorithm is fairly broken, but fortunately for us (and the rest of the world), SHA-1 is not as broken (yet).
- Researchers have discovered something akin to the first vulnerability (the arbitrary hash collision) in SHA-1, but as yet, there does not seem to be a way to translate that vulnerability into

To Optimize or Not to Optimize …

- While hashes are extremely useful for a wide variety of applications, they really cannot provide the same level of data protection that a "true" cryptographic algorithm, such as AES, can provide.
- One of the problems with hashes is that they produce a fixed-size output for arbitrary-length data.
- It doesn't take much thought to realize that if the message is larger than the size of the hash (and maybe even if it is smaller), some information is lost in processing.
- While hashes can work to give you a guarantee (large or small) that the data is intact and represents the original information, they cannot be used (at least directly) to encode the data such that it is extremely difficult for an attacker to get at it, but also can be decoded back into the original information given the proper "key."
- Hashes are, by nature, one-way operations, there is no way to build the original data from a hash value (in general, anyway—you may be able to guess at it if the message was small enough).
- To be able to effectively communicate information securely, it is vital that the data remains intact, albeit obfuscated, in the encrypted message.
- For this we need "true" cryptography as is found with symmetric and asymmetric encryption algorithms.

To Optimize or Not to Optimize … (Cont.)

- The useful properties (and consequences of using) each of the classes of algorithms:
  - Hashes—fast, efficient algorithms generally useful for verifying the integrity of data but provide no means to otherwise protect information.
  - Symmetric—fast (relatively slow compared to hashes in general though), general purpose encryption algorithms. The problem is that the keys need to be shared between receiver and sender somehow (and that sharing obviously cannot be done using a symmetric algorithm—keys must be shared physically or using another method).
  - Asymmetric—slow, special-purpose algorithms that provide the useful ability to facilitate communications without having to share secret keys (public-keys obviously need to be shared, but that's generally pretty easy).

Optimization Guidelines: What NOT to Optimize

- What NOT to optimize? → This is an extremely important concept.
- The reason there are parts of these algorithms that cannot be optimized is that, in order to do their job correctly, they require some significant processing.
- If you go into optimization with your virtual machete blindly, you are more likely than not to remove something important.
- Optimization is not to be taken lightly, so the rule to remember is: “if you aren’t sure about it, don’t do it.”
- A corollary to that rule is to always check with an expert if security is truly important.

Optimization Guidelines: What NOT to Optimize (Cont.)

- So, what are these things that should never be touched?
- Primarily, we want to be sure that any optimization or performance tweaking does not affect the proper functioning of the algorithm itself.
- Almost all cryptographic algorithms have test vectors that can be found to check basic functionality of implementations (be suspicious of algorithms that aren't easily tested).
- You can also use another implementation (Open Source software is always a good place to start since it is free and you can look at source code - just be sure to heed all the legalities inherent in using it).
- If you have broken the algorithm in your efforts to make it faster, it usually shows up quickly - one of the properties of cryptographic algorithms is that they are usually either broken or not, there are few cases where an implementation is "partially functional."
- The problems in implementations are usually with the handling of data once it has been decrypted (this is where buffer overflow and other issues come into play).
- The one case to watch for is if you are implementing the algorithm on both ends of the communication channel - you may break the algorithm, but since you are implementing both sides in the same fashion, you may not notice.
- It is always a good idea to check your algorithm against a known good (if you cannot, then it is a good sign you need to rethink what you are doing).
- Another important rule for optimizing cryptography is that you should never try to tweak the algorithm itself.
- Also, cryptographic primitives need to be properly implemented or you might as well not even use cryptography.

Optimization Guidelines: What NOT to Optimize (Cont.)

- So what is a cryptographic primitive? Cryptographic primitives are the random number generators, entropy sources, and basic memory or math operations that are required by the crypto graphic algorithms.
- For example, the Pseudo-Random Number Generator (PRNG) functions that generate random numbers from some seed value are extremely important to the security of your cryptography.
- Random numbers better be as close to unpredictable true randomness as you can get them—a physical source is usually best.
- If you do need truly random numbers, some manufacturers produce entropy-generating hardware, usually based on interference in electronic circuits.
- For the PRNG, it must have the property of not decreasing the entropy provided by the random seed.

Optimization Guidelines: What NOT to Optimize (Cont.)

- A brief description of the rules:
  1. The implementation should not affect the basic functionality of the algorithm.
  2. You should never try to optimize a cryptographic algorithm by reducing rounds or changing properties of the algorithm's design.
  3. Obey basic cryptographic rules such as using an appropriate random source.
- These rules are of course very generic, but should serve to protect you from most of the common mistakes that can be made when attempting to optimize cryptographic algorithms.
- Most importantly, if you are not comfortable with the optimization (or even simply unsure), don't do it!

Optimization Guidelines: What Can We Optimize?

- The basic rules of optimization apply to cryptography as they do in all other computer engineering disciplines - in other words, there are no rules!
- Unfortunately, optimization is somewhat cryptic (it is similar to cryptography in many ways) and often is considered “black magic” or “voodoo” by many engineers.
- The first thing to think about when looking to optimize a cryptographic algorithm is the math employed by the algorithm.
- Most cryptographic algorithms are very math-heavy, and utilize a number of mathematical primitives that may or may not be implemented in the most efficient manner possible.
- Two of the most common operations are the modulus and XOR, and though the latter is usually quite fast, the modulus can be quite expensive - especially if the hardware does not natively support division operations.
- The real trick in optimizing individual operations is to know what the compiler you are using will do to optimize the code, as well as what the underlying hardware is capable of.
- Since we are dealing with binary machines (try finding a non-binary machine in production), anything that can be decomposed into powers of 2 (which is essentially everything) can benefit from some reworking of the code to utilize the inherent binary capabilities of the underlying hardware.

Optimization Guidelines: What Can We Optimize? (Cont.)

- This convenient property means that we do not care hat is represented by the upper portions of the result since the result of the division will never have any bits set beyond the eighth bit (an astute reader will notice that we could simply assign the integer to the character and rely on the inherent truncation that occurs when assigning to a character in C, but it is usually a bad idea to rely on the language like that). → We can omit the division operation entirely.
- In an algorithm like RC4 where this operation or something similar is performed on every byte of the data being processed, the performance improvement can be staggering.
- In many cases, a function call to a complicated division routine will be replaced by one or two instructions, at a savings of potentially large numbers of clock cycles.
- Generally speaking, anything that you can optimize that is related to the repetitive and mathematical nature of cryptography will be a boon for developing a secure but efficient application.
- Unrolling loops and other generic optimization techniques can also be quite effective, but as with anything there is usually a tradeoff between efficiency and code size (which can be a big deal).
- If you are implementing a cryptographic algorithm from a specification, you can make some design decisions about how the algorithm is implemented, but remember to make sure that none of the optimizations cuts any corners that may lead to a breach of the integrity of the security provided by the algorithm.
- If you are using a prewritten library or source code, you may be limited to the options for optimization provided by the original author.
- When we look at building a secure application using a Microchip PIC processor, we actually utilize an open-source implementation of AES ported to the PIC.

Optimization Guidelines: What Can We Optimize? (Cont.)

```
char index;
int data;
…
// Derive new index from data
index = data % 256;
```

Power-of-2 Optimization Example Part 1

```
char index;
int data;
…
// Derive new index from data using bit-
mask in place of modulus
index = data & 0xFF;
```

Power-of-2 Optimization Example Part 2

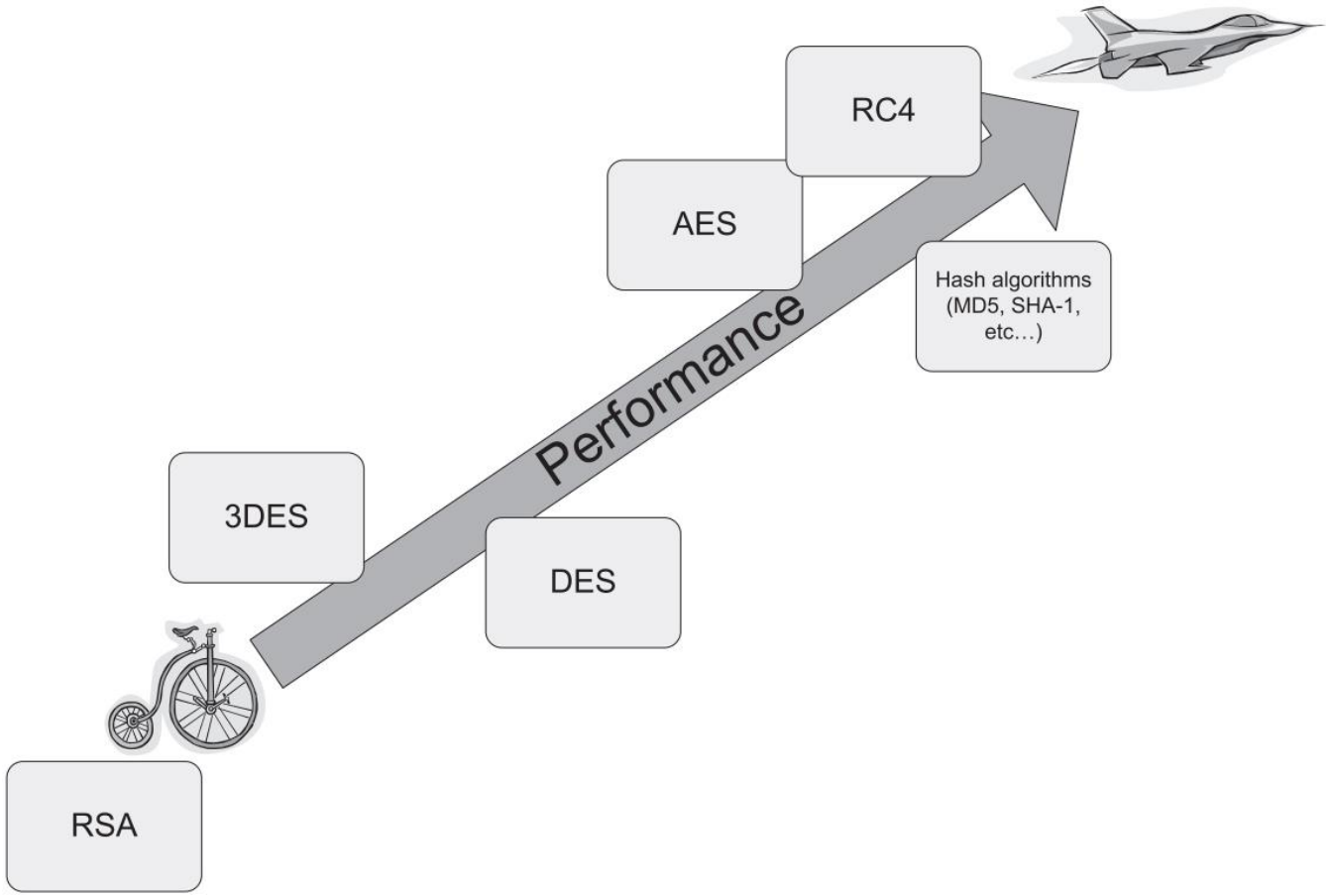
Choosing Cryptographic Algorithms

- You can always move the critical cryptographic operations into hardware, but this may not always be possible, especially considering the additional expense and the fact the design is locked into a particular scheme for encryption.
- In fact, this was exactly the problem with the Wired-Equivalent Privacy (WEP) protocol originally implemented for the 802.11 wireless network protocols.
- Many 802.11 applications utilized specialized hardware to speed up WEP, but once WEP was discovered to have some serious security flaws, it was impractical to update all that hardware.
- The solution (WPA) was implemented to utilize the existing WEP hardware, but this limited the capabilities of the new protocol and took a rather significant effort to implement.
- There is a way to get the performance you need for your application without some of the drawbacks of a hardware-based solution.
- The answer is to design algorithms with a natural performance advantage into your application.

Choosing Cryptographic Algorithms (Cont.)

- Hardware assistance is really the only way to really speed up the algorithm, but there is a method that utilizes a property of the modular math - it is based on an ancient concept called the Chinese Remainder Theorem, or CRT.
- CRT basically divides the RSA operation up using the prime factors p and q used to derive the public and private keys.
- Instead of calculating a full private exponent from p and q, the private key is divided amongst several CRT factors that allow the algorithm to be divided up into smaller operations.
- This doesn't translate into much performance gain unless it is implemented on a parallel processor system, but any gain can be useful for a relatively slow, inexpensive embedded CPU.

Choosing Cryptographic Algorithms (Cont.)



Relative Performance of Common Cryptographic Algorithms

Tailoring Security for Your Application

- Generally speaking, SSL/TLS is going to be your best bet for a general-purpose protocol, since it provides a combination of authentication, privacy, and integrity checking.
- For some applications, however, the security requirements do not include all three security features provided by SSL.
- The properties embodied by the SSL and TLS protocols are inherent in any good secure application.
- When implementing your own security solution around a cryptographic algorithm, instead of using a publicly available protocol, you should remember the lessons provided by SSL and look to implement privacy, authentication, and integrity checking as a sound engineering practice, even if it does not seem that they are all needed.
- One last thing to consider when tailoring security to your particular application is the network in which the device will be deployed.

Assignment

- **Reading Assignment:**
  - Stapko, T., 2011. **Practical embedded security: building secure resource-constrained systems**. Elsevier.
    - ✓"Chapter 8: Choosing and Optimizing Cryptographic Algorithms for Resource-Constrained Systems", Pages 149-171.

Questions?