Internet of Things

Lecture 12

Security and Privacy 2

Encryption and firmware security

Michael Engel

Encryption in the IoT

Requirement:

Encrypt privacy- and security-critical data stored on, and processed, transmitted or received by IoT edge devices

Challenge:

Enabling strong cryptographic security guarantees on devices constrained in computing power, memory and energy

Questions:

- Can we enable security and privacy on IoT edge devices?
- What is the tradeoff between the ***non-functional requirements*** security on the one hand and power, memory, energy consumption on the other hand?
- How can we enable secure operation of edge devices over long timespans?

Cryptography background [2]

Cryptography involves the study of mathematical techniques that allow the practitioner to achieve or provide the following objectives or services [1]:

- **Confidentiality:** keep the content of information accessible to only those authorized to have it. This service includes both protection of all user data transmitted between two points over a period of time as well as protection of traffic flow from analysis.
- **Integrity:** computer system assets and transmitted information can be modified only by authorized users. Modification includes writing, changing, changing the status, deleting, creating, and the delaying or replaying of transmitted messages. It is important to point out that integrity relates to active attacks and therefore, it is concerned with detection rather than prevention. Moreover, integrity can be provided with or without recovery, the first option being the more attractive alternative.
- **Authentication:** assuring that the origin of a message is correctly identified. That is, information delivered over a channel should be authenticated as to the origin, date of origin, data content, time sent, etc. For these reasons this service is subdivided into two major classes: entity authentication and data origin authentication. Notice that the second class of authentication implicitly provides data integrity.
- **Non-repudiation:** prevent both the sender and the receiver of a transmission from denying previous commitments or actions.

Private-key or symmetric-key Cryptography

- Private-key or symmetric-key algorithms are algorithms where the encryption and decryption key is the same, or where the decryption key can easily be calculated from the encryption key and vice versa.
- The main function of these algorithms, which are also called secret-key algorithms, is encryption of data, often at high speeds.
- Private-key algorithms require the sender and the receiver to *agree on the key prior to the communication taking place*. The *security of private-key algorithms rests in the key*
  - **Divulging the key means that anyone can encrypt and decrypt messages**.
  - Therefore, as long as the communication needs to remain secret, the key must remain secret.
- Two types of symmetric-key algorithms:
  block ciphers and stream ciphers

Private-key cryptography approaches

- Block ciphers
  - convert the plain text into cipher text by taking a plain text's block at a time
  - 64 bit or more block size
  - mix chunks of plaintext bits together with key bits to produce chunks of ciphertext of the same size

- Stream ciphers
  - convert the plain text into ciphertext character by character
  - don't mix plaintext and key bits
    - instead, they generate pseudorandom bits from the key and encrypt the plaintext by XORing it with the pseudorandom bits

Public-key Cryptography

- Public-key (PK) cryptography is based on the idea of separating the key used to encrypt a message from the one used to decrypt it.
- Anyone that wants to send a message to party A can encrypt that message using A's *public key* but only A can decrypt the message using her *private key*.
- In implementing a public-key cryptosystem, it is understood that A's **private key should be kept secret at all times**.
- Furthermore, even though A's public key is publicly available to everyone, including A's adversaries, it is impossible for anyone, except A, to derive the private key (or at least to do so in any reasonable amount of time).

Public-key Cryptography approaches

- Algorithms based on the **integer factorization** problem:
  - given a positive integer n, find its prime factorization.
  - RSA, the most widely used public-key encryption algorithm, is based on the difficulty of solving this problem.
- Algorithms based on the **discrete logarithm** problem:
  - given α and β find x such that $\beta = \alpha^x \bmod p$.
  - The Diffie-Hellman key exchange protocol is based on this problem as well as many other protocols, including the Digital Signature Algorithm (DSA).
- Algorithms based on **Elliptic Curves**.
  - Elliptic curve cryptosystems are the most recent family of practical public-key algorithms, but are rapidly gaining acceptance. Due to their *reduced processing needs*, elliptic curves are *especially attractive for embedded applications*.

Approaches for efficient IoT cryptography

- Even when properly implemented, all PK schemes proposed to date are several orders of magnitude slower than the best known private-key schemes. Hence, in practice, cryptographic systems are a mixture of symmetric-key and public-key cryptosystems.
- Usually, a public-key algorithm is chosen for key establishment and authentication through digital signatures, and then a symmetric-key algorithm is chosen to encrypt the communications and the data transfer, achieving in this way high throughput rates.

Which approaches exist to make (symmetric and asymmetric) cryptography algorithms more efficient (in terms of computing power, energy and memory consumption) for embedded and especially IoT devices?

- **Energy-efficient software**
  - Optimize a given implementation using compiler options or hand-written assembly
- **Hardware acceleration instructions**
  - Provide hardware implementations for common operations of a given algorithm that can be used like regular machine instructions
- **Hardware coprocessors**
  - Provide a peripheral device, e.g. connected via SPI, that implements one or more specific algorithms in hardware

Energy-efficient cryptography in software

- Most cryptographic approaches are compute intensive
  - Execute a significant amount of arithmetic and logic operations for en- and decryption

- **Challenge for the IoT:** is it possible to implement energy efficient cryptography in software?
  - If at all possible, do not compromise security for a reduction in energy consumption

- E.g., an energy-efficient variant of SSL/TLS is proposed in [3]
  - However, no universally accepted solution in software so far

Cryptography acceleration instructions

- Specific cryptography algorithms can be accelerated by providing machine instruction set extensions to implement phases of an algorithm
  - e.g. for the AES and SHA algorithms
- Implemented in x86 processors from intel and AMD
- Optional extensions for ARM [5] and RISC-V [6] exist or are proposed
- RISC-V splits the instruction into different optional instruction set extensions, e.g.
  - Zknd – NIST Suite: AES Decryption
  - Zkne – NIST Suite: AES Encryption
  - Zknh – NIST Suite: Hash Function Instructions
  - Zksed – ShangMi Suite: SM4 Block Cipher Instructions
  - Zksh – ShangMi Suite: SM3 Hash Function Instructions
  - Zkr – Entropy Source Extension
  - Zkt – Data Independent Execution Latency
  - Zbkb – Bitmanip instructions for Cryptography

Cryptography acceleration instructions

- Example instruction for the RISC-V Zknd extension
  - Separate instructions to accelerate en/decryption, different phases and special functions of an algorithm – here, AES [7]

2.4. Zknd - NIST Suite: AES Decryption

Instructions for accelerating the decryption and key-schedule functions of the AES block cipher.

RV32	RV64	Mnemonic	Instruction
✓		aes32dsi	AES final round decrypt (RV32)
✓		aes32dsmi	AES middle round decrypt (RV32)
	✓	aes64ds	AES decrypt final round (RV64)
	✓	aes64dsm	AES decrypt middle round (RV64)
	✓	aes64im	AES Decrypt KeySchedule MixColumns (RV64)
	✓	aes64ks1i	AES Key Schedule Instruction 1 (RV64)
	✓	aes64ks2	AES Key Schedule Instruction 2 (RV64)

Hardware crypto support chips

- Hardware coprocessors support a hardware implementation of one of more cryptographic algorithms
- Example: Microchip ATECC608B
  - Cryptographic coprocessor with secure hardware-based key storage
  - Supports ECC-P256, SHA256, and AES128-GC algorithms
  - Protected storage for up to 16 Keys, certificates or data
  - ECDH: FIPS SP800-56A Elliptic Curve Diffie-Hellman (ECDH)
  - NIST standard P256 elliptic curve support (ECC)
  - Hardware support for symmetric algorithms
  - SHA-256 & HMAC hash including off-chip context save/restore
  - AES-128: encrypt/decrypt, Galois field multiply for GCM
- Connected to microcontroller using I2C interface
  - Small 8-pin chip
  - Relatively inexpensive – ~1€ in single pieces
- More information only available under NDA

Firmware updates in the IoT

Requirement:

Keep software on IoT edge devices ("firmware") up to date to enable sustained secure operation and update device features on demand

Challenges:

- Updating firmware over slow and unreliable wireless links
- Ensuring continued operation of devices even when an update fails
- Versioning – networks of devices with differing software versions
- Preventing man-in-the-middle attacks that try to sneak in compromised firmware

Firmware security

Problem:

Prevent an attacker who has **physical access** to an IoT device from reverse engineering the firmware

Challenges: how to keep code and data secret even if the attacker can read the contents of memory chips on the device?

- Encrypted flash enables a secure storage of code and (read-only) data of an IoT application [9,10]
  - Communication either on chip (hard to analyze) or via encrypted bus communication on board possible

Enabling security in hardware

- Additional methods can be applied to protect hardware features from being (ab)used by IoT software (which might be maliciously injected)
- Implementation of a ***trusted execution environment*** (TEE)
  - a secure area of a main processor which guarantees code and data loaded inside to be protected with respect to confidentiality and integrity
  - TEE provides security features such as isolated execution, integrity of applications executing with the TEE, along with confidentiality of their assets
- Typically, sensitive operations are executed in the TEE
  - Content Protection, mobile financial services, authentication
- TEEs require hardware support, e.g. ARM TrustZone [11]
  - Provides barriers and defined interfaces between trusted and untrusted regions of a system's code

Enabling security in system software

- Formal verification of operating systems can also improve embedded system security
  - mathematical proof that an implementation of a system conforms to a given specification
  - verification implies that it is free of implementation bugs such as deadlocks, livelocks, buffer overflows, arithmetic exceptions or use of uninitialised variables
- The seL4 microkernel [12] was the first formally verified OS
  - developed since 2006 in Gernot Heiser's group at UNSW/ NICTA/etc. (see also https://microkerneldude.org)
  - verification started from an executable specification written in Haskell, completed in 2009 (for ARM)
- Open source, available from https://sel4.systems

Conclusion

- Security is still a critically missing feature of many IoT devices
  - We will see examples for attacks in the next lecture

- Approaches to ensure on-device and data transmission security exist
  - Some run counter to IoT non-functional parameter constraints such as energy or cost
  - Tradeoffs between achievable security and non-functional parameter optimization should be avoided

- Hardware support such as TrustZone can enable TEEs
  - Formal verification of system software is an interesting alternative approach

References

[1] T. Wollinger, J. Guajardo, and Ch. Paar, *Cryptography in embedded systems: An overview*, Proc. Embedded World (2003): 735-744

[2] Jean-Philippe Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption*, No Starch Press 2017, ISBN-13: 978-1593278267

[3] Algimantas Venčkauskas et al. *An Energy Efficient Protocol For The Internet Of Things,* Journal of Electrical Engineering. 66. 10.1515/jee-2015-0007, 2015

[4] P. Schwabe and K. Stoffelen, *All the AES You Need on Cortex-M3 and M4*. In: Avanzi, R., Heys, H. (eds) Selected Areas in Cryptography – SAC 2016, Lecture Notes in Computer Science, vol 10532. Springer, Cham.

[5] Arm Cortex-A32 Processor Cryptographic Extension Technical Reference Manual https://developer.arm.com/documentation/100242/0100

[6] B. Marshall, D. Page, and T. Pham, *Implementing the Draft RISC-V Scalar Cryptography Extensions,* In Hardware and Architectural Support for Security and Privacy (HASP '20). ACM, New York, NY, USA

[7] RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions

[8] Microchip ATECC608B CryptoAuthentication Device Summary Data Sheet Document nr. DS40002239B

[9] Meriem Bettayeb, Qassim Nasir, and Manar Abu Talib, *Firmware update attacks and security for IoT devices: Survey,* Proceedings of the ArabWIC 6th Annual International Conference Research Track. 2019.

[10] Joel Rosenberg, *Embedded flash on a CMOS logic process enables secure hardware encryption for deep submicron designs*, IEEE Symp. Non-Volatile Memory Technology 2005

[11] Sandro Pinto and Nuno Santos, *Demystifying ARM Trustzone: A comprehensive survey*, ACM Computing Surveys (CSUR) 51.6 (2019): 1-36

[12] Gerwin Klein et al., *seL4: Formal verification of an OS kernel,* Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009