Security for the internet of things : a bottom-up approach to the secure and standardized internet of things

Timothy Claeys

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Timothy Claeys

Thèse dirigée par **Bernard Tourancheau**
Professeur, Université Grenoble Alpes
et coencadrée par **Franck Rousseau**
Maitre de Conférence, Grenoble INP

préparée au sein de **Laboratoire d'Informatique de Grenoble (LIG)** dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (EDMSTII).**

Sécurité pour l'Internet des Objets:
Une approche de bas en haut pour un Internet des Objets sécurisé et normalisé

Security for the Internet of Things:
A bottom-up approach to the secure and standardized Internet of Things

Thèse soutenue publiquement le **19 décembre 2019**,
devant le jury composé de :

Bernard Tourancheau
Professeur, Université Grenoble Alpes, Directeur de thèse
Marine Minier
Professeure, Université de Lorraine, Rapporteur
Laurent Toutain
Professeur, IMT Atlantique Bretagne-Pays de la Loire, Président
Congduc Pham
Professeur, Université de Pau et des Pays de l'Adour, Examinateur
Mathieu Cunche
Maître de Conférences, INSA Lyon, Examinateur
Franck Rousseau
Maître de Conférences, Grenoble INP, Co-Encadrant de thèse

Abstract

The rapid expansion of the IoT has unleashed a tidal wave of cheap Internet-connected hardware. For many of these products, security was merely an afterthought. Due to their advanced sensing and actuating functionalities, poorly-secured IoT devices endanger the privacy and safety of their users.

While the IoT contains hardware with varying capabilities, in this work, we primarily focus on the constrained IoT. The restrictions on energy, computational power, and memory limit not only the processing capabilities of the devices but also their capacity to protect their data and users from attacks. To secure the IoT, we need several building blocks. We structure them in a bottom-up fashion where each block provides security services to the next one.

The first cornerstone of the secure IoT relies on hardware-enforced mechanisms. Various security features, such as secure boot, remote attestation, and over-the-air updates, rely heavily on its support. Since hardware security is often expensive and cannot be applied to legacy systems, we alternatively discuss software-only attestation. It provides a trust anchor to remote systems that lack hardware support. In the setting of remote attestation, device identification is paramount. Hence, we dedicated a part of this work to the study of physical device identifiers and their reliability.

The IoT hardware also frequently provides support for the second building block: cryptography. It is used abundantly by all the other security mechanisms, and recently much research has focussed on lightweight cryptographic algorithms. We studied the performance of the recent lightweight cryptographic algorithms on constrained hardware.

A third core element for the security of the IoT is the capacity of its networking stack to protect the communications. We demonstrate that several optimization techniques expose vulnerabilities. For example, we show how to set up a covert channel by exploiting the tolerance of the Bluetooth LE protocol towards the naturally occurring clock drift. It is also possible to mount a denial-of-service attack that leverages the expensive network join phase. As a defense, we designed an algorithm that almost completely alleviates the overhead of network joining.

The last building block we consider is security architectures for the IoT. They guide the secure integration of the IoT with the traditional Internet. We studied the IETF proposal concerning the constrained authentication and authorization framework, and we propose two adaptations that aim to improve its security.

Finally, the deployment of the IETF architecture heavily depends on the security of the underlying communication protocols. In the future, the IoT will mainly use the object security paradigm to secure data in flight. However, until these protocols are widely supported, many IoT products will rely on traditional security protocols, i.e., TLS and DTLS. For this reason, we conducted a performance study of the most critical part of the protocols: the handshake phase. We conclude that while the DTLS handshake uses fewer packets to establish the shared secret, TLS outperforms DTLS in lossy networks.

Résumé

La rapide expansion du marché de l'IoT a permis de relier de plus en plus de matériels bon marché à l'Internet. Pour bon nombre de ces objets, la sécurité ne constitue pas une priorité. En raison de leurs fonctionnalités avancées de détection et de manipulation, ces produits IoT mal sécurisés mettent en danger la vie privée et la sécurité de leurs utilisateurs.

Bien que l'IoT englobe des objets connectés de capacités variables, dans ces travaux, nous nous concentrons sur les équipements contraints en énergie, en ressources mémoires, et à faible puissance de calcul. Ces restrictions limitent non seulement la possibilité de traitements, mais aussi la capacité à protéger les données et les utilisateurs. Afin de sécuriser l'IoT, nous identifions plusieurs éléments de bases permettant de fournir des services de sécurité sur l'ensemble d'un équipement.

L'implémentation des mécanismes de sécurité au niveau matériel constitue un premier pilier pour l'IoT sécurisé. Diverses fonctions, telles que le démarrage sécurisé, l'attestation à distance et les mises à jour "over-the-air", dépendent en effet fortement de son support. Comme l'implémentation de la sécurité matérielle est souvent coûteuse et ne peut être appliquée aux systèmes existants, nous étudions l'attestation purement logicielle. Cette méthode fournit une racine de confiance aux systèmes distants qui ne supportent pas la sécurité au niveau matériel. Dans le cadre de l'attestation à distance, l'identification de l'appareil est primordiale. Une partie de ce travail est donc consacrée à l'étude des identificateurs physiques des dispositifs et de leur fiabilité.

L'IoT sécurisé repose sur un deuxième élément clé: la cryptographie. Cette dernière est abondamment utilisée par tous les autres mécanismes de sécurité et largement étudiée. Nous étudions les performances des algorithmes cryptographiques récents pour les dispositifs contraints.

Un troisième élément central pour sécuriser l'IoT est la capacité de la pile protocolaire à sécuriser les communications. Nous montrons par exemple qu'il est possible d'exploiter la tolérance du BLE à la dérive d'horloge pour établir un canal couvert. D'autre part, il est possible de monter une attaque de déni de service en exploitant les phases énergivores du réseau, notamment la phase d'attache. Nous proposons dans ces travaux un algorithme défensif qui réduit quasiment à néant les surcoûts liés à la connexion au réseau.

Les architectures de sécurité constituent le dernier pilier pour la sécurité de l'IoT. Elles permettent en effet de guider le déploiement d'un IoT sécurisé à grande échelle. Après avoir étudié la proposition de l'IETF de schéma d'authentification et d'autorisation pour l'IoT, nous proposons deux pistes d'amélioration de la sécurité.

Enfin, la mise en place d'une architecture de sécurité implique le choix du protocole. Dans le contexte des réseaux contraints énergétiquement, le critère déterminant sera la consommation. Même si, à l'avenir, l'IoT utilisera principalement le paradigme d'objets sécurisés pour protéger les données, tant que ces derniers ne seront pas largement supportés, de nombreux produits IoT s'appuieront sur les protocoles de sécurité traditionnels tels que TLS et DTLS. C'est pourquoi nous réalisons une étude de performance sur la partie la plus critique de ces protocoles : l'établissement du secret partagé. Nous montrons que, même si le "handshake" DTLS utilise moins de paquets pour établir le secret partagé, TLS obtient des meilleurs résultats dans les réseaux avec pertes.

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Franck Rousseau and Bernard Tourancheau, for their guidance over the past four years. They allowed me to independently develop my research while supporting me with excellent advice. I am grateful for the opportunities they have given me which have lead to many unforgettable experiences around the world.

Additionally, I would like to thank all the members of the jury for their compelling questions and comments during the defense. I am grateful to Prof. Laurent Toutain and Prof. Marine Minier, for having reviewed my entire manuscript, and Mr. Mathieu Cunche and Prof. Congduc Pham, for their helpful remarks on my work and presentation.

A major part of my research was done in the context of the IoTize project. I would like to thank Vincent and Francis from the eponymous company who aided me in understanding many complex topics related to microcontrollers. Furthermore, I wish to thank all the wonderful people in the Drakkar research group. I have learned from everyone in the team and their help has been invaluable in completing this work. I particularly enjoyed all the time I spent with Henry and Etienne during breaks, talking about a wide variety of scientific topics. More than once they have helped me overcome technical obstacles.

Throughout my Ph.D. I have met countless new people and some have become close friends. From Grenoble, I want to thank Jacques, Lina, David, and Aline with whom I have spent many hours outside of the lab enjoying the French Alps in winter and summer, Makbule, who I met in Berkeley and who has been a big support in the final months, and finally Margaret for her assistance in correcting my English grammar and spelling. In addition, I want to thank my friends in Belgium and the KRSG rowing club who have always enthusiastically welcomed me back whenever I came to visit.

Special thanks go to Elodie, Colin, Pierre, and Mara. Besides being incredible colleagues, Elodie and Pierre have helped me extensively in my daily life in France. I was always welcome at the Morino's home and they were there for me during moments of hardship and joy.

Finally, I want to write down my profound gratitude to my family. Leaving Belgium to start a Ph.D. in a foreign country is not an easy choice, but with the unwavering support, encouragements and love of my parents, Alexandra and Frank, my brother Ruben, and grandparents, it has become an amazing adventure.

Without all these people, I probably would not have succeeded in completing this thesis. Thank you to all who have supported me and believed in me.

Contents

List of Figures

List of Tables

List of Acronyms

6LoWPAN	IPv6 Low-power Wireless Personal Area Networks. 5, 48, 57, 59, 60, 64, 119–121, 124, 126–130, 138, 142, 145, 146
6top	6TiSCH Operation Sublayer. 97, 122, 124
AAD	Additional Authenticated Data. 55
ACE	Authentication and Authorization for Constrained Environments. 3, 4, 6, 51, 52, 56, 132–134, 136, 137, 140, 142, 145–147
ACL	Access Control List. 62
AE	Authenticated Encryption. 25, 32, 34, 35
AEAD	Authenticated Encryption with Authenticated Data. 5, 34, 35, 41–43, 45, 48, 55, 56, 58, 59, 62, 75, 135–138, 140
AES	Advanced Encryption Standard. 2, 5, 26, 27, 29, 30, 32–35, 43, 55, 62, 124
AGC	Automatic Gain Control. 83
AIMD	Additive Increase/Multiplicate Decrease. 123
AMQP	Advanced Message Queuing Protocol. 57
API	Application Programming Interface. 51, 53, 72, 76, 139, 143
AS	Authorization Server. 132–138, 140–142
ASN	Absolute Slot Number. 61, 62, 97, 100, 101
BCH	Bose, Ray-Chaudhuri, Hocquenghem. 94, 108
BFT	Byzantine Fault Tolerance. 52
BLE	Bluetooth Low Energy. 3–5, 86, 96, 105–108, 111, 112, 114, 146
CA	Certificate Authority. 117, 134
CAESAR	Competition for Authenticated Encryption: Security, Applicability, Robustness. 2, 5, 18, 41, 42, 45, 145
CBC	Cipher Block Chaining. 27–30, 32, 34, 35
CBC-MAC	Cipher Block Chaining Message Authentication Code. 30, 31, 35
CBOR	Concise Binary Object Representation. 51, 56, 116, 132, 138
CCA	Chosen-Ciphertext Attack. 21, 25, 34, 45
CCM	Counter and Cipher Block Chaining. 34, 35, 43, 55, 62
CMOS	Complementary Metal–Oxide–Semiconductor. 89

COA	Ciphertext-Only Attack. 20
CoAP	Constrained Application Protocol. 5, 48, 51, 53–57, 61, 63, 116, 138, 139, 147
CoJP	Constrained Join Protocol. 63
CORE	Constrained RESTful Environments. 53, 54, 116
COSE	CBOR Object Signing and Encryption. 48, 55, 56, 116, 132–135, 137, 138, 147
COTS	Commodity Off-The-Shelf. 10, 66, 70, 71
CPA	Chosen-Plaintext Attack. 20, 25, 27, 28, 34, 36
CPU	Central Processing Unit. 1, 5, 10–13, 18, 42, 43, 70, 72–78, 96, 104, 107–109, 111–113, 146
CRHF	Collision-Resistant Hash Function. 32, 33, 39
CSI	Channel State Information. 83
cTLS	compact TLS. 147
CTR	Counter Mode. 27–29, 34, 35
CWC	Carter-Wegman Counter. 35
CWT	CBOR Web Token. 133, 134
DCCP	Datagram Congestion Control Protocol. 123
DDoS	Distributed Denial-of-Service. 1
DES	Data Encryption Standard. 26
DLP	Discrete Logarithm Problem. 24, 38–41, 45, 147
DMA	Direct Memory Access. 108
DNS	Domain Name System. 54
DNSSEC	Domain Name System Security Extensions. 54
DODAG	Destination- Oriented Directed Acyclic Graph. 60
DoH	DNS-over-HTTPS. 54
DoS	Denial-of-Service. 3, 5, 54, 60, 64, 66, 96, 99, 102, 119, 126, 141, 145
DPA	Differential Power Analysis. 42
DRM	Digital Rights Management. 78
DRTM	Dynamic Root-of-Trust for Measurement. 66, 67, 70, 74
DSA	Digital Signature Algorithm. 36, 40
DTLS	Datagram Transport Layer Security. 3–6, 43, 48, 52, 54, 58, 59, 63, 116, 118–121, 123, 124, 126–130, 132, 139, 141, 146
EB	Enhanced Beacon. 62, 63, 97, 98, 100–102, 104, 114
ECB	Electronic Codebook. 27, 28
ECBC	Encrypted-CBC. 30
ECDH	Elliptic Curve Diffie-Hellman. 43, 44
ECDHE	Elliptic Curve Diffie-Hellman Ephemeral. 58
ECDSA	Elliptic Curve Digital Signature Algorithm. 41, 43, 44, 58, 138
ECN	Explicit Congestion Control. 123
EDHOC	Ephemeral Diffie-Hellman over COSE. 5, 48, 56, 132, 134–137, 142, 147
EUI	Extended Unique Identifiers. 59, 63

EVM	Ethereum Virtual Machine. 53
GCM	Galois-Counter Mode. 35, 43
GPIO	General Purpose Input/Output. 12
HIP	Host Identity Protocol. 141
HMAC	Hashed Message Authentication Code. 33, 71, 74, 135–137
HTTP	Hypertext Transfer Protocol. 48, 51, 53, 54, 61, 63
HTTPS	Hypertext Transfer Protocol Secure. 50
I2C	Inter-Integrated Circuit. 12
IC	Integrated Circuit. 10, 89, 90, 92
ICE	Indisputable Code Execution. 69, 70, 74, 78
IDAU	Implementation-Defined Attribution Unit. 73
IDM	Integrated Device Manufacturer. 90
IE	Information Elements. 62, 97
IETF	Internet Engineering Task Force. 2, 3, 10, 48, 51, 53, 57–60, 63, 116, 122, 123, 134, 137, 145, 147
IIoT	Industrial Internet of Things. 1, 10
IO	Input/Output. 13, 76
IoT	Internet of Things. 1–6, 10–15, 18, 40, 42, 45, 48, 49, 51, 53, 54, 57–61, 63, 64, 66, 78, 82, 89, 94, 96, 116, 119, 121, 126, 130, 132, 133, 138, 142, 145–147
IP	Internet Protocol. 48, 54, 57, 59, 61, 63, 118, 119, 123
IPC	Inter-Process Communication. 77, 78
IPSec	Internet Protocol Security. 33–35, 60
IPv4	Internet Protocol version 4. 59
IPv6	Internet Protocol version 6. 14, 48, 57, 59, 60, 63, 64, 118, 119, 121, 124, 126, 127, 138
ISR	Interrupt Service Routine. 77
IT	Information Technology. 1, 10, 18, 64
IV	Initialization Vector. 27–30, 33, 34, 43, 54, 55
JP	Join Proxy. 63
JRC	Join Registrar/Coordinator. 63
JSON	JavaScript Object Notation. 50, 51
JTAG	Joint Test Action Group. 71
JWT	JSON Web Tokens. 50
KA	Keep-Alive. 97–102
KDF	Key Derivation Function. 71
KID	Key Identifier. 55
KIM	Key Identifier Mode. 62
KPA	Known-Plaintext Attack. 20
LFSR	Linear Feedback Shift Register. 21, 42, 108

LoS	Line-of-Sight. 86, 87, 113
LQI	Link Quality Indicator. 14
M2M	Machine-to-Machine. 1, 51
MAC	Message Authentication Code. 24, 29–36, 38, 39, 45, 55, 59, 60, 64, 75, 77, 118, 121
MCU	Microcontroller Unit. 2, 11–13, 74, 89
MFL	Maximum Fragment Length. 118, 121, 126
MITM	Man-In-The-Middle. 137
MMU	Memory Management Unit. 13
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor. 89, 90
MPU	Memory Protection Unit. 11–13, 68, 73, 76–78
MQTT	Message Queuing Telemetry Transport. 57
MSF	Minimal Scheduling Function. 122, 127
MSS	Maximum Segment Size. 119, 121, 123, 126, 130
MTU	Maximum Transmission Unit. 48, 121, 126, 127, 129
NAT	Network Address Translation. 118
NIST	National Institute of Standards and Technology. 2, 24, 26, 41, 42, 44, 45, 147
NSA	National Security Agency. 44
OAuth	Open Authorization. 2–4, 6, 48–51, 132, 133, 142, 145
OCB	Offset Codebook. 34
OS	Operating System. 66–68, 74–77
OSCORE	Object Security for Constrained RESTful Environments. 4–6, 48, 52, 54–56, 63, 64, 116, 132, 134, 138, 139, 141, 147
OTP	One Time Pad. 19, 24
P2P	Peer-to-Peer. 53, 105
PAN	Personal Area Network. 97
PC	Personal Computer. 10, 43, 66, 70, 74, 76, 77, 108, 114, 124
PCB	Printed Circuit Board. 10
PDP	Provable Data Possession. 69
PDR	Packet Delivery Ratio. 102
PEP	Performance-Enhancing Proxy. 120, 121, 127, 130
PGP	Pretty Good Privacy. 54
PGT	Packet Guard Time. 97–101, 106
PKI	Public-Key Infrastructure. 57, 132, 134, 135, 142, 147
PLL	Phase-Locked Loop. 12
PMTU	Path MTU. 119, 121
PoP	Proof-of-Possession. 51, 52, 132, 133, 135–137, 139, 140, 143
POR	Power-On-Reset. 71
PoS	Proof-of-Stake. 52, 53, 138

PoW	Proof-of-Work. 52, 53, 138, 141
ppm	Parts-per-Million. 83, 85, 97, 103, 105
PPT	Probabilistic-Polynomial Time. 19–24, 31, 37
PRF	Pseudorandom Function. 22, 23, 30–34, 71
PRG	Pseudorandom Generator. 21, 24, 25, 68, 70
PRP	Pseudorandom Random Permutation. 23, 25, 30
PSK	Pre-Shared Key. 58, 63, 64, 117
PUF	Physical Unclonable Function. 3, 5, 82, 89, 93, 94, 145, 146
RAM	Random-Access Memory. 10, 13, 43, 48, 67, 68, 74, 116, 123, 126
REE	Rich Execution Environment. 72, 74
RF	Radio Frequency. 83
RFC	Request for Comments. 2, 40, 51, 57, 58, 64, 116–118, 134, 138, 147
RFID	Radio-Frequency Identification. 10
RNG	Random Number Generator. 19
ROLL	Routing Over Low-power and Lossy Networks. 60
ROM	Read-Only Memory. 12, 68, 72, 74
ROP	Return-Oriented Programming. 67, 75, 78
RoT	Root-of-Trust. 66, 71, 73
RPL	Routing Protocol for Low power and Lossy Networks. 5, 60, 102
RSA	Rivest–Shamir–Adleman. 36–38, 44, 45, 57, 58
RSN	Record Sequence Number. 118
RSS	Radio Signal Strength. 5, 83, 85–89, 94, 146
RTC	Real-Time Clock. 12
RTM	Root-of-Trust for Measurement. 66–68, 71, 72, 77
RTO	Retransmission Timeout. 127
RTR	Root-of-Trust for Reporting. 66, 70–72, 82
RTS	Root-of-Trust for Storage. 66, 70–72
RTT	Round Trip Time. 4, 58, 119–121
S/MIME	Secure/Multipurpose Internet Mail Extensions. 54
SACK	Selective Acknowledgment. 120, 129, 130
SAU	Security Attribution Unit. 73
SCA	Side-Channel Attack. 42
SHA	Secure Hash Algorithm. 32, 124
SNR	Signal-to-Noise Ratio. 14
SoC	System-On-Chip. 11, 12, 71, 90, 124
SPI	Serial Peripheral Interface. 12
SR	Status Register. 70
SRAM	Static Random Access Memory. viii, 5, 12, 42, 71, 72, 89–94, 146
SRTM	Static Root-of-Trust for Measurement. 66, 67, 70, 71

SSH	Secure Shell. 33, 35
SSL	Secure Socket Layer. 34
SWD	Serial Wire Debug. 71
TCB	Trusted Computing Base. 5, 66, 71, 72, 74, 76–78
TCP	Transport Layer Protocol. 6, 48, 57, 63, 116, 119–121, 123, 124, 126–130, 146
TCXO	Temperature-Compensated Crystal Oscillator. 109, 111, 112
TEE	Trusted Execution Environment. 72, 74
TLS	Transport Layer Security. 2–6, 18, 21, 29, 32–35, 41–43, 45, 48, 51, 54, 56–59, 61, 63, 64, 116–121, 123–130, 132, 139, 146, 147
TOCTTOU	Time-of-Check-to-Time-Of-Use. 70
TSCH	Time Slotted Channel Hopping. ix, 61–64, 96–99, 101, 104–106, 114, 120–122, 124, 127, 129, 146
TSMP	Time Synchronized Mesh Protocol. 61
UART	Universal Asynchronous Receiver/Transmitter. 12, 71
UDP	User Datagram Protocol. 4, 53, 57–59, 63, 116, 118–120, 123, 126, 127, 129, 130, 138
UHF	Universal Hash Function. 30–32
UOWHF	Universal One-Way Hash Function. 31, 32
URI	Uniform Resource Identifier. 54, 140
URL	Uniform Resource Locator. 50, 134
VM	Virtual Machine. 67
VRB	Virtual Reassembly Buffer. 60
WEP	Wired Equivalent Privacy. 34
WPA2	Wi-Fi Protected Access II. 34
XOR	Exclusive Or. 24–26, 28, 42, 70

List of Publications

International Conferences

Olivier Alphand, Michele Amoretti, Timothy Claeys, Simone Dall'Asta, Andrzej Duda, Gianluigi Ferrari, Franck Rousseau, Bernard Tourancheau, Luca Veltri, and Francesco Zanichelli. IoTChain: A Blockchain Decurity Architecture for the Internet of Things. In *IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2018.

Timothy Claeys, Franck Rousseau, Boris Simunovic, and Bernard Tourancheau. Thermal Covert Channel in Bluetooth Low Energy Networks. In *13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, pages 267–276. ACM, 2019.

Timothy Claeys, Franck Rousseau, and Bernard Tourancheau. Securing Complex IoT Platforms with Token Based Access Control and Authenticated Key Establishment. In *International Workshop on Secure Internet of Things (SIoT)*, pages 1–9. IEEE, 2017.

Timothy Claeys, Franck Rousseau, Bernard Tourancheau, and Andrzej Duda. Clock Drift Prediction for Fast Rejoin in 802.15. 4e TSCH Networks. In *26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2017.

National Conferences in French

Timothy Claeys, Franck Rousseau, Boris Simunovic, and Bernard Tourancheau. Faire évader secrètement des données d'un réseau BLE sécurisé. In *CoRes*, 2019.

Introduction

Context

Since its conception in the late 1960s, the Internet has been an ever-changing and ever-growing technology. The advent of the smartphone, approximately 15 years ago, introduced the first wave of pervasive, Internet-connected, computing devices providing notable computational power and global connectivity from the palm of our hand [1]. The emerging IoT (Internet of Things) further enables the ubiquitous computing paradigm. Fueled by advances in low-power computing and networking, it interconnects numerous everyday objects and attaches them to the Internet. The origins of the IoT trace back to the initial M2M (Machine-to-Machine) communication protocols, used in telemetry. However, the functionalities we associate with today's IoT considerably surpass the original M2M features. Many IoT devices come with advanced sensing and actuating capabilities, allowing for the coordination of actions on local, national, or global levels, using the input of real-time data [2].

The applications enabled by the IoT vary widely. Examples range from small-scale commercial systems such as home automation to public use cases, e.g., *Smart Cities*. The IIoT (Industrial Internet of Things) also forms one of the core components of "Industry 4.0", a.k.a. the 4$^{\text{th}}$ Industrial Revolution, which defines the concept of the *Smart Factory*. The integration of production means with the IIoT makes it possible to gather and analyze data across machines. The pervasive and connected nature of the IIoT can present businesses with crucial insights into their supply chain and production processes, allowing them to create a leaner and more agile global economy. Although the added value of the IoT seems extraordinary, one question remains: what are the security implications?

The interconnection of previously autonomous and isolated systems naturally gives rise to problems in interoperability and security. As the number of deployed IoT devices increases, it becomes more challenging to manage the infrastructure and those devices securely. The consequences of poorly secured IoT devices are not solely affecting the security and privacy of their immediate users. It was not until recently, with the record-breaking DDoS (Distributed Denial-of-Service) attacks by the Mirai botnet, which took down large parts of the Internet, that the full force of the vulnerable IoT became clear [3]. In addition, the actuating capabilities of many IoT devices provide novel attack vectors for systems and infrastructure, such as the power grid. Recent work by Soltan et al. [4] highlighted the potential risks incurred by vulnerable high wattage IoT devices. Through the compromise of a few tens of thousands of air conditioners, water heaters or electric ovens, an attacker could cause a large-scale blackout of a nation's energy grid.

The design of security protocols and mechanisms for the IoT is a daunting task. The low-power nature, limited computational capabilities, and possibly hostile operating environments of the IoT systems impose further challenges. The security mechanisms we can typically find in the traditional IT (Information Technology) infrastructure are ill-suited for the IoT. They frequently require power-hungry computations, high bandwidth, and specialized hardware. Nonetheless, over the course of this Ph.D, significant progress has been made by the research community and industry towards a more secure IoT. Several new trends are appearing in various areas of IoT security.

On the lowest level, chip designers are moving towards the integration of hardware security mechanisms in the low-power CPUs (Central Processing Units). Through the addition of hardware-enforced boundaries, IoT software developers can divide their code by privilege level. The most striking example in this field was the introduction of the Arm TrustZone technology for the immensely popular Cortex-M line of processors back in 2017.

Only last year, the CAESAR (Competition for Authenticated Encryption: Security, Applicability, Robustness) cryptographic competition concluded, proposing two novel encryption algorithms, Ascon, and ACORN, optimized for the IoT. At the same time, NIST (National Institute of Standards and Technology) kicked off a new competition for IoT cryptography, likewise focussing on symmetric encryption. The proposals are currently under evaluation, but since cryptographic algorithms typically need years of scrutiny, they will not be available in the near future. For now, the IoT must rely on long-established cryptographic primitives, such as AES (Advanced Encryption Standard) and elliptic curve cryptography. As a compromise, many new MCU (Microcontroller Unit) designs aimed at the IoT market now include some form of hardware acceleration to relieve the overhead of cryptography partially.

Finally, network security protocols make abundant use of cryptographic protocols to protect data in flight from unauthorized access. Compared to the prevalent security protocol on the traditional Internet, i.e., TLS (Transport Layer Security), which focusses on securing the communication channel, the IoT networking protocols pursue an object security model. The protection of network traffic is relocated from the transport layer to the application layer, allowing for a more fine-grained control over the protection domains. The connectionless security model is also a better fit with the needs of the IoT applications, which often rely on asynchronous traffic patterns, intermediate data caches to allow for duty-cycled devices and group communication. In July of 2019, the IETF (Internet Engineering Task Force) ratified the first of a suite of upcoming RFC (Request for Comments), detailing the new security protocols aimed at the IoT. Besides the object security architectures, the IETF is working on a lightweight authentication and authorization framework for the IoT, inspired by OAuth (Open Authorization) 2.0.

Motivation

This thesis is part of the IoTize project whose goal is to propose a simple turnkey solution to extend new and existing embedded systems with two functionalities: a human-machine interface and a connection with the Internet, see Figure 1. Throughout this Ph.D. we worked with the IoTize development team to investigate the tradeoffs between security and production costs. The challenges and issues faced during the product development inspired many of the research questions presented in this thesis.

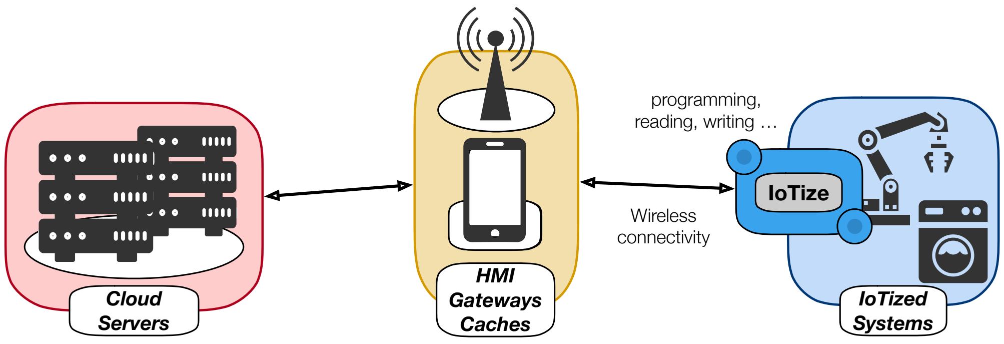


Figure 1: The IoTize architecture [5]. (Icons: [6])

Of vital importance in the design of new IoT products is the choice of appropriate hardware. Although hardware-backed security such as Arm's TrustZone carries many advantages, for some constrained applications, the extra cost of the added silicon might be prohibitive. Additionally, the immutability of the hardware can also be a drawback. In case design flaws are discovered in the hardware architecture, it is not possible to fix the problem through updates. These issues inspired us to study alternative approaches to defend IoT systems from software attacks. While shielding Internet-connected devices from malicious code injection without hardware support is near to impossible, remote attestation protocols provide a valid alternative. Attestation routines attempt to evaluate the integrity of a target system periodically. A significant challenge is to ensure that the routine executes untampered on the chosen target. To this aim, we studied several fingerprinting techniques on low-power hardware. By combining the attestation routine with physical identifiers, the security guarantees of the attestation routines improve notably.

Since the IoTize product attaches itself to existing, likely battery-powered devices, the incurred additional energy should be minimal. In the context of the IoT, communication is expensive; thus, the choice of a fitting protocol stack was imperative. Both BLE (Bluetooth Low Energy) and IEEE 802.15.4e have pushed the boundaries of low-power communication and are excellent candidates. They share a similar design philosophy, relying on time synchronization to minimize the duty cycle and, consequently, the energy consumption of both the transmitter and receiver devices. However, due to imperfections in the hardware, keeping the devices synchronized is not trivial. The link layer synchronization algorithm is critical to the operation of the protocols and is, therefore, an appealing target to adversaries. We discovered two weaknesses intrinsic to low-power, time-synchronized protocols. Further investigation led to the proposal of a defensive algorithm for IEEE 802.15.4e mesh networks and a covert channel attack for BLE connections.

Link layer protocols potentially provide hop-by-hop encryption of the data within the mesh network, but it is the responsibility of higher-layer protocols to encrypt the sensor data destined for the Internet. With the object security protocols still mostly being under construction by the IETF, for now, the IoT application developers fall back on traditional transport layer security protocols, e.g., TLS and DTLS (Datagram Transport Layer Security). The IETF chose DTLS as the designated security protocol for the IoT because of its tolerance for packet losses, but DTLS does not offer any other benefits compared to TLS. Ironically, the DTLS handshake is more complicated than the TLS equivalent and seems poorly adapted to lossy networks. Motivated by this paradoxical observation, we conducted an in-depth study of both the TLS and DTLS handshake over an IEEE 802.15.4e network.

IoTize exposes previously isolated systems to the Internet. The addition of the IoTize hardware allows interested parties to obtain sensor data and interact with the internal configuration of the embedded system remotely, see Figure 1. We directly understand that an access control system should strictly limit these powerful capabilities to authorized parties only. In 2014, the IETF started working on the adoption of the OAuth 2.0 architecture for the IoT. The resulting ACE (Authentication and Authorization for Constrained Environments) framework provides an authentication and authorization mechanism that suits the IoTize use case well. However, the framework foresees the use of unconditionally trusted authorization servers. Since it is unclear which party, vendor or buyer, would operate these trusted servers, we proposed two architectures which strive to reduce the capabilities of the authorization server. The latter helps not only the IoTize business case but also improves the overall security and robustness of the framework.

Contributions

This thesis presents several contributions to various aspects of IoT security. We structure their presentation in a bottom-up fashion:

- At the physical layer, we have studied the different approaches to device identification. The IoT brings with it an explosion of new hardware attached to the Internet. If we have any hope of securely managing these devices, reliable device identification is paramount. This contribution is placed in the context of software-based remote attestation protocols. Software-based attestation is valuable for systems that lack support for security features such as memory protection. We analyzed two main categories of physical fingerprinting techniques: signal-based identifiers and PUF (Physical Unclonable Function)-based identifiers. However, we show that none of the mechanisms can provide a constrained IoT device with a unique, unspoofable identifier. Nonetheless, in scenarios where devices are compromised and secret keys are lost, physical fingerprinting is the only viable approach to distinguish between the compromised devices. Combined with a remote attestation protocol that can detect, and possibly purge, malicious code on the system, we could build a tool to recover hacked IoT devices automatically.
- At the link layer, the contribution is two-fold. First, we show that IEEE 802.15.4e, one of the most promising protocols for mesh networks, is vulnerable to a simple DoS (Denial-of-Service) attack. IEEE 802.15.4e is a time-synchronized protocol. It allows both the sender and receiver to reduce radio usage, obtaining duty cycles of 1% or lower. The synchronization mechanism between the nodes is crucial to the operation of the network. The key insight is that with a short jamming attack, a malicious actor can knock-out large parts of the network for extended periods. The attack is destructive

because there does not exist a fast algorithm to synchronize nodes back to the network. Currently, the synchronization of a node demands a significant amount of time and energy. We proposed an algorithm that minimizes energy consumption and latency for a node to rejoin a network to which it was previously synchronized. The rejoin cost is only slightly higher than what the node would have consumed during regular network operation, which is an improvement of a factor 1000 over the current situation.

The second part of the contribution focusses on the BLE protocol. BLE shares its design philosophy with IEEE 802.15.4E. It also uses time synchronization and channel hopping. In this part of our thesis, we propose a novel technique to build covert channels between BLE-enabled devices. We exploit the temperature sensitivity of the crystal oscillators used by the BLE radio chip to modulate a clock skew on top of the precisely-timed, legitimate BLE traffic. The covert channel does not disrupt the working of the protocol but is measurable by a colluding device. We showed that the attack works on different hardware architectures, with varying success, by implementing it on a Raspberry Pi 3B, a Motorola smartphone, running Android, and an iPhone 5s on iOS 11.

- The network security for the IoT has seen a paradigm shift from the transport layer security to object security at the application layer. Nonetheless, transport layer security will remain widely used in the IoT, at least in the near future, while we wait for the ratification of multiple object security protocols. We have performed an in-depth study of the two most prevalent transport layer security protocols, TLS and DTLS, in the context of the IoT. The performance results show that DTLS, while being tolerant to packet losses once the security context is established, functions poorly when the handshake message are carried over a lossy network compared to TLS. The lack of acknowledgment packets and accurate estimates on the RTT (Round Trip Time) between both endpoints causes uncontrolled retransmissions of oversized UDP (User Datagram Protocol) datagrams, resulting in the DTLS handshake transmitting significantly more bytes than TLS.
- The final contribution centers around security architectures for the IoT. We propose two novel frameworks that are mostly inspired by the recent work on authorization and authentication architectures. Both frameworks adopt the token-based protocol flows of the ACE and OAuth architectures but make improvements in terms of security. In the first part, the proposed scheme follows the security model of OAuth 1.0*a*, which integrates its proper security model, facilitating deployment in complex networking environments. It uses self-securing tokens that are independent of the underlying transport protocols. At the same time, we reduce the amount of trust users must have in third-parties, responsible for access token distribution. The latter was also the primary design goal of the second part of this contribution. We replaced the trusted party with an authorization blockchain which transparently regulates access token generation through smart contracts. Additionally, we combined the blockchain with the OSCORE (Object Security for Constrained RESTful Environments) object security architecture to provide full end-to-end security [7].

Structure of the thesis

Instead of focussing on a single element of the IoT ecosystem, we have attempted to sketch a broad picture of the current state of the IoT security. This document is primarily split into two parts. The first part describes the different building blocks of a secure IoT. While we consider each building block in a separate chapter, they are not independent, and we often refer to other chapters to point out relationships and dependencies. The second part of this thesis contains the various contributions. We present them in a bottom-up fashion, starting from the physical layer of the IoT device, and working our way up through the networking stack to generic security architectures for large-scale deployments.

Part I, Chapter 1 – Define a Thing IoT is a broad domain, we can categorize a plethora of devices as IoT. Therefore, we start the first chapter by narrowing down the actual definition of an IoT device used throughout this thesis. We present the most common hardware components we can find inside an IoT device and discuss how they differ from traditional Internet hosts. The chapter also zooms in on the various security features constrained hardware provides. Finally, we point out the tradeoffs that are typically made by IoT product designers, and we briefly discuss the emerging trends in IoT hardware related to security.

Part I, Chapter 2 – Basics of Cryptography The second chapter provides an exhaustive background on modern cryptography. Cryptography is used abundantly by the other building blocks, and a solid foundation is therefore imperative. The chapter starts by presenting the different notions of security. It explains how to reason about the security of a cipher, and it lists the various threat models considered by cryptographers. Next, we examine the most common symmetric-key constructions. We describe each time the generic format and show a real-life example, e.g., block ciphers and AES. The AEAD (Authenticated Encryption with Authenticated Data) ciphers form the culmination of our discussion on symmetric cryptography. The second part of the chapter focusses on public-key cryptography. We explain how one-way functions such as integer factorization and the discrete logarithm problem lay the foundation of public-key encryption, key establishment protocols, and digital signatures. We also consider elliptic curve cryptography, since it makes public-key constructions feasible for the constrained IoT. The chapter concludes with a performance study of the popular cryptographic algorithms on low-power hardware. We also compare the advanced AEAD ciphers from TLS 1.3 with the winning algorithms from the CAESAR competition.

Part I, Chapter 3 – A Secure IoT Networking Stack We describe the standardized IoT networking stack in Chapter 3. We present the different layers of the networking stack in a top-down manner and provide each time a discussion of the available security mechanisms. First, we consider the issue of authorization and access control in the IoT. We detail the solution developed for a similar problem on the world wide web, and we show the parallels with the IoT's equivalent. Next, we describe CoAP (Constrained Application Protocol) and the upcoming security protocols, i.e., OSCORE, and EDHOC (Ephemeral Diffie-Hellman over COSE), to protect application-layer traffic. We briefly sketch the internals of TLS and DTLS, but a more comprehensive study is presented in Chapter 7. RPL (Routing Protocol for Low power and Lossy Networks) and 6LoWPAN (IPv6 Low-power Wireless Personal Area Networks) are also considered with their associated security model. Finally, we present the IEEE 802.15.4E protocol and link layer security.

Part I, Chapter 4 – System Security for Constrained Devices The final building block we discuss, concentrates on trusted computing for the constrained IoT. The chapter handles two device architectures. Firstly, we consider devices without any form of hardware-backed security boundaries. We present different remote attestation methods that attempt to detect malicious activity on the device with a software-only approach. Secondly, we enumerate various techniques, designed by both the industry and research community to build hardware security into low-power devices. Most designs rely on a combination of trusted hardware and software components, although one architecture has a pure hardware TCB (Trusted Computing Base). We finish the chapter with a discussion on the tradeoffs between software and hardware trusted computing.

Part II, Chapter 5 – Scalable and Secure Physical Device Identification In the first chapter of the contributions, our goal is to analyze simple hardware fingerprinting methods to improve the security guarantees software-based remote attestation protocols can provide. We limit ourselves to signal-based techniques and PUFs since most other methods require specialized equipment. Signal-based fingerprinting tries to extract unique identifiers from overheard radio transmission. We study clock skew patterns and RSS (Radio Signal Strength) features. Next, we discuss the SRAM PUF on constrained hardware. We conclude that none of the techniques suffice to reliable identify IoT devices.

Part II, Chapter 6 – Vulnerabilities in Time Synchronized Link Layer Protocols Chapter 6 presents our findings on vulnerabilities in time-synchronized protocols. The first part gives an in-depth overview of the inner-working of the IEEE 802.15.4E standard. It explains how the network formation and maintenance operates, and how it deals with the unstable crystal oscillators. We show that there is a significant DoS vulnerability, which allows an attacker to take most of the network down. Next, we detail the algorithm we designed that minimizes the latency and energy cost for a node to perform network resynchronization. The algorithm reduces most of the additional energy a node had to spend after desynchronization, which makes the DoS attack less attractive for attackers. In the second part of the chapter, we switch roles and build a covert channel between BLE-enabled devices. It relies on thermal energy emitted by the CPU to modulate data under the form of a clock skew. We perform the attack on three different hardware platforms and study its performance.

Part II, Chapter 7 – Performance of Transport Layer Security over IEEE 802.15.4E Networks We briefly discuss both TLS and DTLS during our overview of the state of the art of IoT networking protocols, but in Chapter 7, we perform an in-depth study. More precisely, we discuss the differences between both protocols in the context of the IoT. We focus on the initial handshake since it is one of the most critical and resource-demanding phases of both TLS and DTLS. Although DTLS is tolerant towards packet losses and thus suitable for a specific range of IoT applications, during the handshake, the protocol performs poorly in lossy networks. DTLS implements some of the TCP (Transport Layer Protocol) features half-heartedly to provide reliability during the handshake, but the lack of roundtrip timing information results in unnecessary retransmissions. Finally, we also consider the overall memory footprint of both protocols and notice that their performance is almost identical.

Part II, Chapter 8 – Security architectures for the Internet of Things The final chapter concerns security architectures for the IoT. In Chapter 3, we gave an overview of ACE, and OAuth 2.0, its equivalent on the world wide web. We build on this knowledge to propose two new frameworks. The first framework takes the ACE model but combines it with the security model of OAuth 1.0a. In both ACE and OAuth 2.0, security is decoupled from the architecture. They entirely rely on the underlying networking stack to provide the necessary protection. The OAuth 1.0a model implements its proper security. Our new framework additionally removes some of the trust in the different endpoints, improving the overall security of the architecture. The second proposal entirely focusses on removing trusted third-parties from the architecture. It uses blockchain technology to authorize Internet hosts to access IoT resources dynamically. The blockchain is combined with OSCORE, the object security framework for the IoT [7], to provide end-to-end security of the exchanged data.

We conclude this work with a general conclusion. It summarizes the contributions from the individual chapters and provides several future research directions concerning security for the constrained IoT.

Part I

Building blocks of the secure Internet of Things

Chapter 1

Define a Thing

Contents

Introduction

Nowadays, a broad assortment of computing devices classify as IoT (Internet of Things). Examples range from comprehensive home automation systems to the tiniest energy-scavenging RFID (Radio-Frequency Identification) chips. The IoT also revolutionizes in the industry. The IIoT (Industrial Internet of Things) can implement supply chain tracking or monitor assembly lines at manufacturing sites through the deployment of wireless sensor networks [8]. Smartdust can even be used to combat global warming by providing us with large-scale environmental measurements in hostile areas [9, 10].

In contrast to traditional IT (Information Technology) systems, i.e., PCs (Personal Computers) and servers, built to run a variety of programs, IoT devices tend to be optimized for a particular application. The latter helps to lower the price and in some cases, energy consumption of the end product. The wide range of diverse IoT applications causes a significant disparity in hardware characteristics. Some IoT systems have high-performance specifications and possess advanced computing capabilities [11], while others must survive years on a single battery [12].

In this thesis, we are concerned with the constrained IoT. We define the constrained IoT as the set of Internet-connected devices that have strict limitations in terms of computational power, total available memory, and energy supply. Although the capabilities of a single device are limited, equipped with a battery and a radio transceiver, they can form a wireless network. Their wireless nature makes the devices mobile, and in multi-hop network formation, they can cover large areas by relaying information.

In this first chapter, we explore the typical hardware components of COTS (Commodity Off-The-Shelf) constrained IoT devices. Detailed knowledge of the various hardware building blocks is crucial to the security of any device. We will see, throughout the subsequent chapters, that hardware support forms the foundation of fundamental security properties such as isolation, device authentication, confidentiality, and integrity. Because of the stringent limitations on the constrained devices, many hardware features, available on more powerful systems, are not implemented. By studying the components available on low-power devices, we can build alternative techniques in an effort to provide similar security properties.

1.1 Terminology

Throughout this document, we refer to tiny devices with limited CPU (Central Processing Unit) performance, memory, and energy supply as constrained devices. We also interchangeably use the terms: smart object, thing, board, platform and embedded system to describe a PCB (Printed Circuit Board) that contains different electronic components, e.g., ICs (Integrated Circuits), energy supply, and a wireless interface. When the constrained devices form a network, we refer to them as nodes or motes.

1.1.1 A Low-Power Device Classification

Despite the overwhelming variety of existing constrained IoT devices, the IETF (Internet Engineering Task Force) [13] has proposed a classification based on the performance characteristics of the devices. Typically, these characteristics are available code memory, RAM (Random-Access Memory), and the CPU speed.

- **Class 0 devices** are the most constrained. They have less than 10 KiB of RAM and less than 100 KiB of code memory. These devices will most likely have one preconfigured setup which persists throughout their lifetime. Class 0 devices generally cannot be secured or managed in the traditional sense [13].
- **Class 1 devices** are more powerful than Class 0 (i.e., 10 KiB of RAM and 100 KiB of code memory), but it remains a challenge to deploy a traditional secure communication stack. The current research majorly efforts focus on adapting communication protocols and security schemes to suit this device class.
- **Class 2 devices** are the most powerful devices in the classification. They have around 50 KiB of RAM and 250 KiB of code memory. These devices are capable of running the traditional Internet protocol suite, but for reasons such as bandwidth efficiency, available code space for the applications and overall energy consumption of the device, they also use the adapted IoT protocol stack.

The boundaries between the classes coincide with distinct clusters of commercially available chips. Each higher class corresponds to a bump in computational capabilities and memory, but also means the devices require additional power. During this thesis, we principally focused on devices from Class 1 and 2.

1.2 The Internals of IoT hardware

To more accurately define *a Thing*, we have a closer look at the usual hardware of a constrained IoT device. We can divide the hardware into four subsystems: the microcontroller unit, the wireless transceiver, sensors and actuators, and the energy source. On some hardware designs, these subsystems are distinguishable, while other designs integrate multiple components in a single SoC (System-On-Chip).

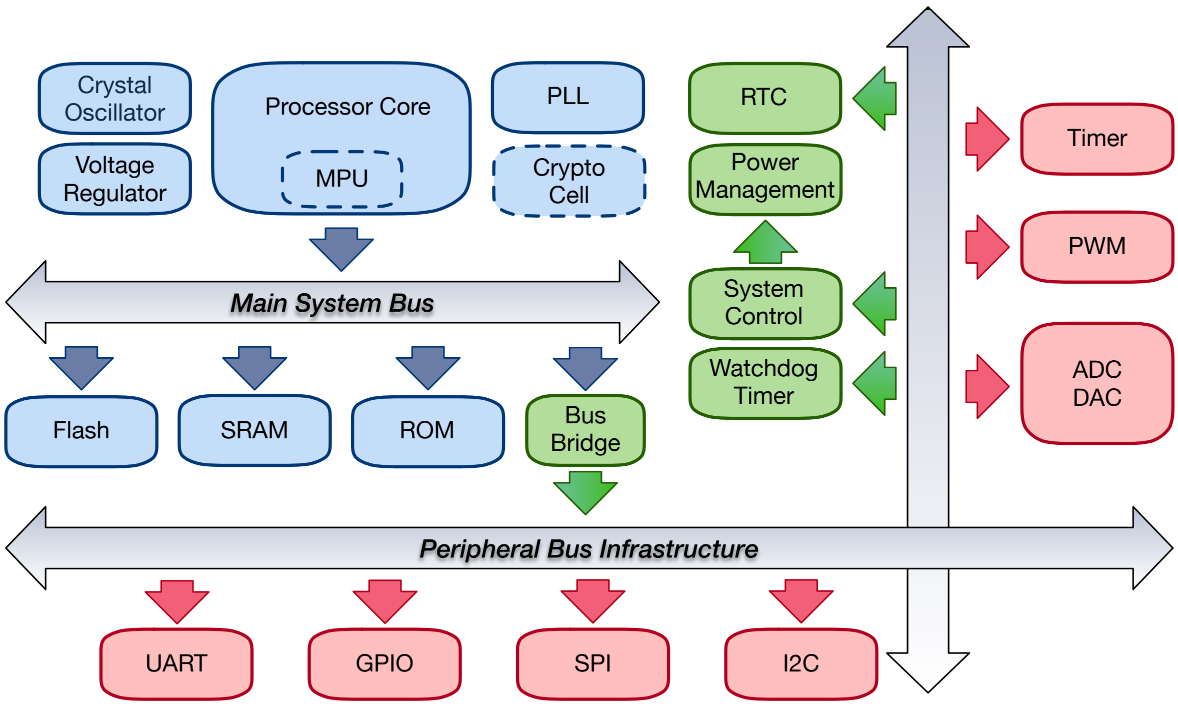


Figure 1.1: Main components of a modern microcontroller [5].

1.2.1 The Microcontroller Unit

At the center of the IoT device, we find the MCU (Microcontroller Unit). Depending on the requirements of the final product, the microcontroller has different performance characteristics and components. The simplified block diagram in Figure 1.1 shows the most common components inside a microcontroller [5].

At the heart of the MCU, we can find the CPU, it performs the gross of the calculations. It has several internal registers, some of which are generic and used to store intermediate results, others describe the current CPU configuration. The CPU contains an interrupt controller. When an interrupt occurs, the CPU jumps to the appropriate interrupt vector and executes the interrupt service routine. An interrupt allows the CPU to execute a specific section of code when a particular internal or external event transpires. A variety of MCU subsystems can be configured to generate interrupts. The CPU is also directly connected to several memory systems through the main system bus. Optionally, CPU accesses to memory and MCU subsystems can be monitored and verified by an MPU (Memory Protection Unit). Low-end CPUs typically do not feature a cache; however, the MCU designer might choose to add a system-level cache to accelerate flash look-ups. For example, the STM32F4 family of MCUs employs the ART Accelerator cache to circumvent the speed mismatch between the processor and the flash memory. On each instruction fetch from flash, the ART cache additionally loads 4 to 5 instructions. Preloading instructions keeps the CPU running at full speed without having to install more high-end flash. Some MCUs also come with a cryptographic coprocessor. This component provides hardware acceleration for cryptographic primitives.

The predominant memory types are flash, ROM (Read-Only Memory), and SRAM (Static Random Access Memory). The first two are non-volatile memories. They are used to store program code, with the difference that the latter can only be programmed once (usually by the device manufacturer). It usually holds the boot code, the first instructions executed by the CPU during the device's start-up sequence. The SRAM is volatile memory. It can hold both program code as well as data. It is mainly used as a scratchpad by the CPU when executing program code. SRAM holds among other things the stack and heap segments of an executing program.

Other components directly linked to the CPU are the crystal oscillator and the PLL (Phase-Locked Loop) system. The crystal oscillator is an external component that provides a stable clock signal to the system. The crystal and its surrounding electrical components determine the exact frequency of the oscillator. Some microcontrollers also have internal oscillators based on RC-circuits (however, their output frequency can be somewhat inaccurate) [5]. The PLL allows the software to control the frequencies of the clock signals fed into the digital circuits. Different digital logic requires distinct frequencies. As a result, the MCU can have an external crystal of just 12 MHz, while the processor is running at a much higher clock speed (e.g., 100 MHz), and some of the peripherals running at a divided clock speed. The PLL uses the crystal oscillator as a reference clock while generating the various operating frequencies. Timer circuits use the clock signals to tell time. A timer consists of a register that increments for each elapsed *clock tick*. The RTC (Real-Time Clock) is a particular type of timer that counts seconds. It uses a low-power crystal oscillator and keeps track of time even when the device is powered off.

MCUs often possess various energy modes. The power management system controls the power mode of the device. In standard mode, an MCU may draw a substantial amount of energy. Increasingly aggressive sleep modes shut down additional parts of the MCU to save energy. The most aggressive modes only keep a few subsystems of the MCU active. In these configurations, data retention in SRAM is not always guaranteed [14]. When the sleep mode is active, memory leakage and the crystal oscillator are the primary contributors to the overall energy consumption. Decreasing energy consumption in sleep mode is thus of utmost importance as the IoT devices spend most of their time sleeping. The MCU can be configured to return to standard mode on the arrival of an interrupt.

Finally, a regular MCU implements several interfaces to the external world. These come under the form of GPIO (General Purpose Input/Output), UART (Universal Asynchronous Receiver/Transmitter), I2C (Inter-Integrated Circuit) and SPI (Serial Peripheral Interface) systems. They provide a parallel or serial data interface to control external devices and to read external signals.

In general, production cost and the available energy supply limit the MCU's capabilities. The addition of hardware, to support additional features, requires more silicon, which raises the price of the end product and the overall energy consumption. The energy constraints of the MCU directly impact the supported total memory size and the processor frequency [2]. Larger memory sizes imply larger cell and transistor count, which directly influences leakage currents. Consequently, MCU sleep modes that retain data memory in sleep modes have higher power consumption compared to those where data memory is not retained. A higher processor frequency consumes additional energy, but it also allows the CPU to complete its tasks more rapidly, resulting in more time spent in low power modes. In Table 1.1, we depict some characteristics of commercially available MCU systems.

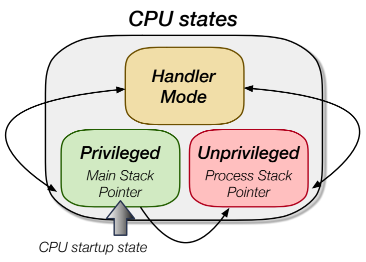
TABLE 1.1: Specifications of commercially available MCUs.

MCU Identifier	**Instruction size [bits]**	**SRAM [KiB]**	**Max. CPU freq. [MHz]**	**CPU on [mA/MHz]**	**CPU sleep [µA]**	**MPU**	**Crypto support**
ATmega8A [15]	8	1	16	0.56	1.5		
MSP430F16x [16]	16	10	8	1.8	5.1		
CC2538 SoC [14]	32	32	32	0.185	0.44	✓	✓
STM32F401xD/E [17]	32	96	84	0.146	12	✓	

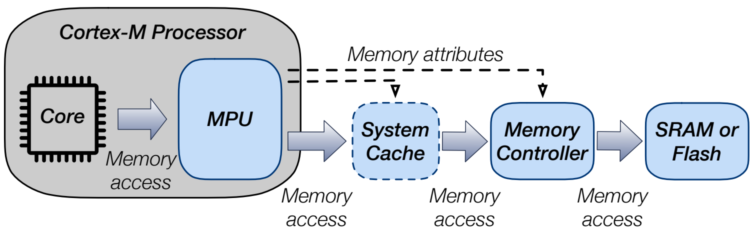
1.2.1.1 Security Features of the Microcontroller

The limitations put on hardware due to energy and cost restrictions also directly affect the security of the constrained IoT. Security critical features common to in high-end systems have been made optional, allowing to trade security for production cost and energy-efficiency. A prime example is memory protection. More powerful commodity systems possess an MMU (Memory Management Unit). The MMU is a hardware component that contains each application to its own virtual memory space. The MMU prevents any attempt of a program to access the memory of a different application or the kernel. It sits between the processor and the RAM, and it checks any memory accesses performed by the processor. The MMU drastically limits the damage a rogue or malicious application can inflict on other applications and the system.

In the constrained IoT, the (silicon) cost of a full-fledged MMU is prohibitive, but protecting the memory of safety and security-critical functions is still an indispensable feature, definitely for Internet-connected devices. The Arm Cortex-M family of processors, prominent in designs for low-power devices, optionally includes an MPU[1]. The MPU is a programmable component that defines multiple memory ranges and attaches specific memory attributes and memory access permissions. The system designer can protect parts of memory from non-privileged software, by configuring the MPU in the start-up code of the device. Memory zones used as communication buffers for possibly untrusted interfaces, e.g., a networking interface, are particularly vulnerable and should be isolated from the rest of the system. The developer should mark the code that accesses these untrusted memory areas as unprivileged. The Arm architecture uses memory-mapped IO (Input/Output) to communicate with peripheral devices. The MPU can thus additionally shield the peripherals from unprivileged code.



(a) Cortex-M0+ CPU states.



(b) MPU checks CPU memory accesses and triggers a *MemFault* if the access is illegitimate.

Figure 1.2: Security features in low-power MCUs.

Devices that contain an MPU have two CPU privilege levels. The code running in unprivileged mode has restrictions on memory and register accesses, while privileged code can access the entire memory and all CPU registers. Privileged code can easily switch to unprivileged mode, but the inverse is not true. The code must go through an exception handler, which can enforce certain restrictions, to return the CPU to privileged mode.

Hardware accelerators for cryptographic primitives form another class of vital hardware components for the security of the constrained IoT. Security protocols employed on the Internet require expensive cryptographic operations, see Chapter 3. For the low-end MCUs used in IoT devices, the cryptographic functions result in a notable overhead. Implementing the algorithms in software not only requires a lot of flash memory, but they also hog the CPU and consume a lot of battery. Hardware accelerators free space in flash memory, and they allow the CPU to offload expensive operations. They also facilitate the implementation of the networking protocols by providing a fast and more energy-efficient way of encrypting and decrypting data. Hardware implementations of cryptographic functions can also increase the overall security of the system. In the next chapter, we present some essential cryptographic primitives and show that while they can provide powerful security properties, small mistakes in the implementations can lead to fatal flaws in the security protocols. Providing vetted hardware implementations of the primitives reduces the chance of critical security bugs.

[1]Starting from the Cortex-M0+. The least powerful Cortex-M0 does not include an MPU to reduce silicon area and power consumption

1.2.2 The Wireless Transceiver

To become part of a network and by extension the entire Internet, IoT devices are frequently equipped with a wireless transceiver. The role of the wireless transceiver or radio is twofold. On the one hand, it encodes the digital information into an electromagnetic wave emitted by the radio antenna. On the other hand, a radio needs to be able to capture such an emitted electromagnetic wave and decode it back to digital data [2]. Table 1.2 presents the characteristics of some state-of-the-art transceivers.

TABLE 1.2: Energy consumption of IEEE 802.15.4 radios.

Radio Identifier	**Tx @ 0 dBm [mA]**	**Rx [mA]**
CC2538 SoC [14]	24.0	20.0
CC2520 [18]	25.8	18.5
CC2420 [19]	19.5	21.8
AT86RF231 [20]	11.6	10.3

The ability of a radio to demodulate and decode an electromagnetic wave depends on several factors such as the strength and quality of the incoming signal and the radio modulation techniques. The SNR (Signal-to-Noise Ratio) metric represents the ratio of the strength of the received signal to the background noise at the receiver side. A low SNR makes it harder for the receiving radio to decode the incoming signal. The SNR value depends on the power of the emitted signal, receiver sensitivity, and interference on the channel. The LQI (Link Quality Indicator) is a metric that tries to gauge the quality of the received transmission. It uses either energy detection, the SNR or both [21]. A low LQI indicates a complicated demodulation process.

Due to energy constraints, IoT devices use limited transmission power when communicating. If we compare current consumption values in Table 1.2 with those in Table 1.1, we can conclude that radio usage dominates the energy consumption of constrained IoT devices. This is a key insight that has played an fundamental role in the design of the low-power communication protocols. The limited output power impacts the range over which IoT devices can communicate. Depending on the environment, the distances range from a few meters to tens of meters. Devices can form multi-hop networks by establishing peer-to-peer links between neighboring devices. In these formations, data is relayed over long distances to a central device using a specialized protocol stack. Characteristic for the IoT, the protocol stack supports IPv6 (Internet Protocol version 6), allowing the individual devices in the network to be uniquely addressed on the Internet. We discuss the constrained Internet stack further in chapter 3.

1.2.3 Sensors and Actuators

Sensors in IoT devices provide measurements from their immediate surroundings. The most basic sensors are temperature, humidity, light, pressure, accelerometer, and gyroscope. IoT devices can also interact with their environment through the use of an actuator. Examples of simple actuators that are controlled by an IoT device are smart light bulbs, smart locks, valves, or switching boards.

1.2.4 The Energy Source

One can use constrained devices in scenarios where the conventional energy sources are unavailable, or the devices have to be mobile, and they can therefore not use a wired energy source. The predominant energy source in these scenarios is a battery. Because of the vast number of devices, the battery life must be as long as possible to prevent frequent battery renewal. For systems that require very little energy, energy scavenging techniques are an option. Examples are solar cells, piezoelectric elements, and temperature gradients. The harvested energy can power the device directly (when there is no battery), or it can recharge a battery.

Discussion

Production cost and energy supply are the primary factors guiding the design of constrained IoT devices. Both factors influence the overall device capabilities, not only in terms of computational power and available memory but also in its potential to protect users and their data from attackers. The hardware design of the device directly affects the security guarantees the end product can offer. Regrettably, the past years have shown that product designers have all too often sacrificed the security-critical components in order to lower the overall production cost. Prior to the IoT, the impact of such vulnerabilities was limited as embedded devices were part of closed and isolated systems. Attackers almost always required physical access to the system to exploit the flaws. However, the current IoT evolution pushes the product designers to provide networking capabilities to anything with elemental computational power, exposing the devices to attack vectors they were never designed to withstand.

Luckily, the tide is turning. Chip vendors and system designers have become aware of the new threats their products are facing, and they are building security into the hardware. The integration of security mechanisms in the early stages of the product development phase ensures that the hardware can act as a trusted foundation. Upon this foundation, we can create the building blocks that will help us protect the IoT. Features such as secure boot, network security, secure over-the-air updates, intellectual property protection, etc. can help in mitigating the bulk of the attacks that the IoT is currently facing. Hardware assistance not only supports us in building secure systems; it permits us to do this efficiently. For example, hardware acceleration for cryptographic algorithms almost eliminates its overhead and dramatically improves the adoption of the newly developed secure networking protocols. Hardware-enforced memory protection can efficiently separate critical system components from untrusted software, allowing for the deployment of secure boot, firmware updates, and patching, damage control, etc. New emerging technologies such as Arm TrustZone are state of the art in hardware security support for the IoT. They implement the discussed security features and much more in an attempt to provide system-wide protection. We discuss Arm TrustZone and similar technologies in Chapter 4.

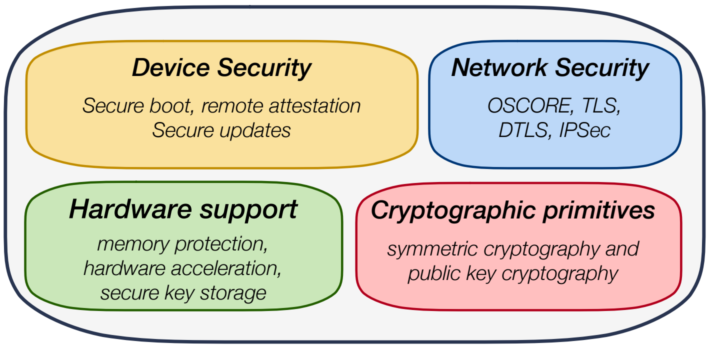


Figure 1.3: Building blocks of a secure IoT device.

Although appropriate hardware support seems almost indispensable for the future of the IoT, it is only a part of the solution. In the subsequent chapters, we discuss the different security building blocks that must flawlessly work together to build a secure IoT device. We start our discussion with a chapter on cryptography, a powerful but delicate tool that appears as a crucial component in the other building blocks.

Chapter 2

Basics of Cryptography

Contents

Introduction

Cryptography is ubiquitous. These days, the bulk of computing systems use it extensively. Historically, cryptography was the art of inventing codes to hide messages from the prying eyes of the enemy. Modern cryptography, however, is a science that encompasses much more than only secret communication. It includes, among others, data integrity and authentication, secret key exchange algorithms, and digital signatures. The most common security properties cryptosystems strive to provide are:

Confidentiality is the property whereby information is not revealed to unauthorized parties [22]. To obtain confidentiality, we use encryption. Encryption algorithms convert the *plaintext* to some illegible form, known as the *ciphertext*. The mapping of the plaintext to the ciphertext, for a given encryption algorithm, depends on the encryption key. Only the entities in possession of the decryption key can retrieve the original data through the process of decryption.

Data Integrity protects the data from being altered. Cryptographic primitives such as cryptographic hash functions are applied to the data to detect, with very high probability, malicious modifications or accidental changes to the data.

Data Origin Authentication guarantees that only a party with knowledge of a specific secret key can create some data. Data authentication is considered to be a stronger property than data integrity. Data origin authentication directly implies the data integrity property since a modified message has a new source [23].

Entity Authentication and Non-Repudiation ensure that a party cannot deny that they created some data. Systems achieve non-repudiation through the use of a public key cryptography algorithm (i.e., digital signatures). This property is, for example, particularly crucial in electronic voting schemes, where a voter should not be able to deny having already cast a vote. Non-repudiation seems similar to data origin authentication, but there is a fundamental difference between both properties. Data origin authentication uses a symmetric key, a secret key shared by two or more entities. A recipient of a protected message can verify the message originated from an entity with access to the key, but the recipient cannot, however, prove which of the entities in possession of the key created the message. In a cryptographic scheme that offers non-repudiation, we can uniquely identify the creator of the message.

While powerful, cryptography remains highly brittle. Over the past decades, many, seemingly innocuous mistakes in cryptographic specifications and implementations lay at the foundation of disastrous vulnerabilities in security systems. The substantial amount of flaws found in the past version of networking protocols such as TLS (Transport Layer Security) are living proof that cryptography is notoriously hard to get right. With the emergence of the IoT, the security community is now facing an additional challenge. Cryptographic schemes need to consume fewer CPU cycles and energy while still providing a sufficient level of security. An increasing number of IT systems support elliptic curve cryptography to accommodate the utilization of less potent devices on the Internet. Asymmetric cryptography based on elliptic curves is faster, requires less memory, and is overall more secure than its counterpart based on modular arithmetic. At the same time, recent public competitions target the design of new lightweight authenticated encryption schemes suitable for the IoT.

Throughout this chapter, we outline the essential concepts of modern cryptography. We introduce the basic notions of information security and present the various threat models for cryptosystems, allowing us to reason about their security. We build on this knowledge to describe the most widely used cryptographic algorithms. First, we discuss the different algorithms that belong to the family of symmetric cryptography: stream ciphers, block ciphers, message authentication codes, and authenticated encryption ciphers. Secondly, we consider the cryptographic mechanisms we can build with asymmetric cryptography. We present public-key encryption, key exchange algorithms, and digital signatures. We distinguish between algorithms based on modular arithmetic over the integers and elliptic curve cryptography.

A part of this chapter is devoted to lightweight cryptography. We review the finalists of the CAESAR (Competition for Authenticated Encryption: Security, Applicability, Robustness) competition for authenticated encryption schemes briefly. We conclude this chapter with a performance analysis of several state-of-the-art primitives on constrained hardware.

2.1 Notions of Security

The basic algorithm used in cryptography to provide confidentiality is called a *cipher*.

Definition 2.1.1. Cipher: A cipher $\mathcal{E}$ defined over a triple $(\mathcal{K},\mathcal{M},\mathcal{C})$, denoting the key space, message space and ciphertext space, is a pair of efficient[1] algorithms (E,D) where $E:\mathcal{M}\times\mathcal{K}\rightarrow\mathcal{C}$ and $D:\mathcal{K}\times\mathcal{C}\rightarrow\mathcal{M}$ such that,

$$\forall m\in\mathcal{M},\ \forall k\in\mathcal{K}:\ D(k,\ E(k,\ m))=m \tag{2.1}$$

One of the more famous ciphers is the OTP (One Time Pad), invented by Gilbert Vernam in 1917. The OTP owes its fame to its property of *perfect secrecy*. Informally stated, a cipher provides perfect secrecy if an intercepted ciphertext reveals no information about the plaintext, even if the attacker, denoted as $\mathcal{A}$, has unbounded computational power.

Definition 2.1.2. Perfect Secrecy: A cipher $\mathcal{E}(E,D)$ over $(\mathcal{K},\mathcal{M},\mathcal{C})$ has perfect secrecy if $\forall m_0,m_1\in\mathcal{M}$ with $|m_0|=|m_1|$, $\forall k\in\mathcal{K}$ and $\forall c\in\mathcal{C}$ such that,

$$\Pr\Big[\mathcal{A}(E(k,m_0)=c)\Big]=\Pr\Big[\mathcal{A}(E(k,m_1)=c)\Big]\quad\text{with}\quad k\overset{R}{\leftarrow}\mathcal{K} \tag{2.2}$$

Equation 2.2 states that the probability distributions obtained by encrypting m_0 and m_1 with randomly selected keys from $\mathcal{K}$ must be equal. An attacker looking at these distributions does not gain any additional information.

The main drawback of the OTP is the length of the encryption key. It must have the same size as the plaintext message. Additionally, every bit of the key must be sampled from a truly random source. The secret key, which can stretch to gigabytes in size, depending on the message, must be shared over a secure channel with the recipient for decryption of the ciphertext. We can directly remark that if such a secure channel exists, then we could use it to transfer our plaintext message without encrypting it.

In practice, finding a strong source of randomness is challenging. The Linux operating system provides a software RNG (Random Number Generator) implemented as a device called `/dev/random`. It obtains entropy from several hardware sources such as keyboard events, mouse events, and hardware interrupts. Additionally, one can use an external device that can generate randomness [24]. These systems use quantum random processes, the time between emissions during radioactive decay, inherent semiconductor thermal noise, shot noise from Zener diodes, or free-running oscillators [25]. In 2012, starting with the Ivy Bridge processor family, Intel introduced a hardware random number generator in its processors. Output from the generator is read using the `RdRand` instruction that is intended to provide a fast uniformly random bit generator. Because of aging, the hardware might malfunction, causing predictable bit sequences. The output of the hardware circuit is, therefore, continuously tested. Lack of strong randomness is also a significant issue on constrained hardware, as discussed by Yan et al. [26].

Due to the above reasons, the OTP is not a practical cipher. Modern cryptography does not consider adversaries with unlimited computational resources. It assumes instead that the adversary's computational power is bounded in some reasonable way. In particular, the adversary is limited to PPT (Probabilistic-Polynomial Time) algorithms.

Definition 2.1.3. Probabilistic Polynomial Time Algorithm: A PPT algorithm is an algorithm that takes an input x and makes random decisions during its execution. It terminates after $|p(x)|$ steps, for some polynomial p.

The assumption of a bounded adversary permits us to use a weaker, albeit more practical, definition of security called *computational security*. We can use Equation 2.2 to motivate the definition of computational security. Instead of insisting that distributions in Equation 2.2 are equal, we only require that they are very close; that is

$$\left|\Pr\Big[\mathcal{A}(E(k,m_0)=c)\Big]-\Pr\Big[\mathcal{A}(E(k,m_1)=c)\Big]\right|\le\epsilon\quad\text{with}\quad k\overset{R}{\leftarrow}\mathcal{K} \tag{2.3}$$

[1]In theory, efficient means an algorithm that runs in polynomial time, in practice some time constraints can be placed on the runtime of the algorithm.

for some very small, or negligible, value of ϵ. In a practical scenario, the definition of negligible is often a scalar, e.g., $\epsilon \leq 2^{-80}$, but in theory, it is expressed as a function of a security parameter n [27]. By picking a larger security parameter, we can obtain a higher level of security. For the most part, the security parameter determines the length of the key used by the cryptographic algorithm.

Definition 2.1.4. Negligible A function $f : \mathbb{Z}_{\geq 1} \to \mathbb{R}$ is called negligible if $\forall c \in \mathbb{R}_{>0}$ there exists $n_0 \in \mathbb{Z}_{\geq 1}$ such that $\forall n \in \mathbb{Z} : n \geq n_0$, we have $|f(n)| < 1/n^c$.

To reason about the security of ciphers, cryptographers use “attack games”. They formalize the interactions between an adversary and the cipher, denoted as the *challenger*. The challenger challenges the adversary to break the cipher. The attack game for encryption security of a cipher $\mathcal{E}(E, D)$, defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$, and a given PPT adversary $\mathcal{A}$ can be defined as follows:

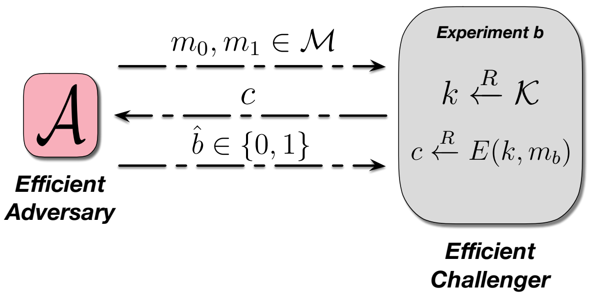


Figure 2.1: Definition of indistinguishable encryption with the attack game.

1. The adversary picks two messages, $m_0, m_1 \in \mathcal{M}$, and sends them to the challenger.
2. The challenger picks a random $b \in \{0,1\}$ and computes $k \xleftarrow{R} K$, $c \xleftarrow{R} E(k, m_b)$. The challenger sends c to the adversary.
3. The adversary outputs a bit $\hat{b} \in \{0,1\}$. The adversary outputs $\hat{b} = 0$ if he thinks the challenger encrypted m_0 and similarly he outputs $\hat{b} = 1$ if he thinks the challenger encrypted m_1.

The encryption scheme is now secure if the success probability of any PPT adversary in the attack game, also denoted as the attacker's *advantage*, is at most negligible. We say that the cipher has indistinguishable encryptions.

$$\text{INDadv}[\mathcal{A}, \mathcal{E}] = \left| \Pr\left[\mathcal{A}(E(k, m_0)) = 1\right] - \Pr\left[\mathcal{A}(E(k, m_1)) = 1\right] \right| \tag{2.4}$$

Definition 2.1.5. Secure Cipher A cipher $\mathcal{E}(E, D)$ is secure if for all PPT adversaries $\mathcal{A}$, INDadv$[\mathcal{A}, \mathcal{E}]$ is negligible.

Informally, a secure cipher is a cipher that only leaks a negligible amount of additional information about the plaintext, regardless of any prior information that the attacker has obtained about the plaintext. The indistinguishable encryption definition can be used to reason about the cipher's strength in the presence of different types of attackers. If a cipher provides indistinguishability in the presence of an attacker with capabilities x, we say it is an x-secure cipher. For example, the above definition describes indistinguishable encryption in the presence of a passive eavesdropping attacker. This cipher is thus a ciphertext-only secure cipher. We now list all the attacker types in order of the “increasing strength”. Each subsequent attacker model incorporates all the previous attacker capabilities.

- ***COA (Ciphertext-Only Attack)***: the attacker can only observe ciphertexts (the values of m_0 and m_1 are not known by $\mathcal{A}$ in the attack game).
- ***KPA (Known-Plaintext Attack)***: the attacker knows the plaintext values of the observed ciphertext messages, but it does not control plaintext. A natural scenario for this model is the encryption of known header information in messages, e.g., e-mail.
- ***CPA (Chosen-Plaintext Attack)***: the attacker can obtain the ciphertext of plaintext messages of its choosing. The attacker thus specifically chooses m_0 and m_1 in the attack game.

- ***CCA (Chosen-Ciphertext Attack)***: the attacker additionally has the capability to decrypt messages of its choice. Although this attack model seems unrealistically strong, many padding oracle attacks on the TLS protocol [28–30] belong to this attack model.

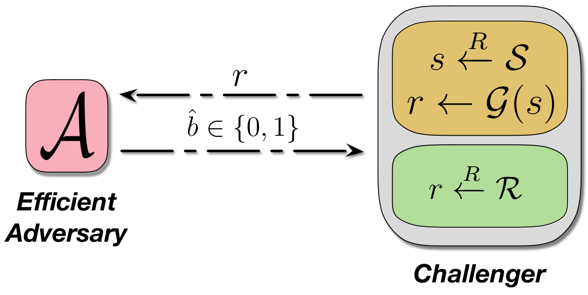
To build systems that are secure in these different threat models, we need *primitives* with certain computational hardness as a property. Examples of these primitives are *pseudorandom generators*, *pseudorandom function families*, and *one-way functions*.

2.1.1 Pseudorandom Generators

A PRG (Pseudorandom Generator) is an efficient, deterministic algorithm $\mathcal{G}$ that picks as input a seed $s : \{0,1\}^n$, uniformly at random from the finite seed space $\mathcal{S}$ and outputs a long random-looking sequence of bits, $r : \{0,1\}^m$, belonging to finite space $\mathcal{R}$.

$$\mathcal{G} : \{0,1\}^n \rightarrow \{0,1\}^m \quad \text{with} \quad m > n \tag{2.5}$$

Many of the classical techniques for building PRGs are unsuitable for cryptographic applications. For example, a standalone LFSR (Linear Feedback Shift Register) is well-known to be cryptographically insecure. We can solve for the feedback pattern given a small number of output bits. A PRG is secure, and suitable for cryptography if its output is *computationally indistinguishable* from a truly random sequence. We use an attack game to provide the security definition of the PRG, see Figure 2.2.



$\mathcal{A}$
Efficient Adversary
r
$\hat{b} \in \{0,1\}$
$s \xleftarrow{R} \mathcal{S}$
$r \leftarrow \mathcal{G}(s)$
$r \xleftarrow{R} \mathcal{R}$
Challenger

Figure 2.2: The PRG is computationally indistinguishable from a true random seed if the distinguishing probability of $\mathcal{A}$ is negligible.

For a PRG denoted as $\mathcal{G}$ and defined over $(\mathcal{S},\mathcal{R})$, and for an adversary $\mathcal{A}$, we present the following attack game.

- The challenger picks at random a value $b \in \{0,1\}$ and generates r. It then sends r to the adversary. The value r is calculated as follows:
  - if $b = 0$: . The challenger picks uniformly at random a seed $s \xleftarrow{R} \mathcal{S}$ and generates a pseudorandom sequence $r \leftarrow \mathcal{G}(s)$.
  - if $b = 1$:. The challenger picks bitstring $r \xleftarrow{R} \mathcal{R}$ from a truly random source.
- Given r, the adversary tries to detect if he has been given a trult random bitstring or a pseudorandom bitstring. He outputs $\hat{b} \in \{0,1\}$.

The attacker wins the attack game if he succeeds with a non-negligible probability. Let W_0 be the event that $r \leftarrow \mathcal{G}(s)$, and W_1 be the event that $r \xleftarrow{R} \mathcal{R}$ [31].

$$\text{PRGadv}[\mathcal{A},\mathcal{G}] = \left| \Pr\left[\mathcal{A}(W_0) = 1\right] - \Pr\left[\mathcal{A}(W_1) = 1\right] \right| \tag{2.6}$$

Definition 2.1.6. Secure PRG A PRG $\mathcal{G}$ is secure if the value PRGadv$[\mathcal{A},\mathcal{G}]$ is negligible for all PPT-bounded adversaries $\mathcal{A}$.

Intuitively, we cannot say that a single fixed bitstring is pseudorandom. Instead, we define pseudorandomness over a distribution of bitstrings. When we say that a specific distribution is pseudorandom, it means that we cannot distinguish it from the uniform distribution over all the bitstrings.

2.1.2 Pseudorandom Functions

A PRF (Pseudorandom Function) is a deterministic, efficient algorithm $\mathcal{F}$ that has two inputs: a key k and an input data block x. The output $y = \mathcal{F}(k,x)$ is called an output data block. The PRF $\mathcal{F}$ is defined over $(\mathcal{K},\mathcal{X},\mathcal{Y})$, where $\mathcal{K}$ is the key space of k, $\mathcal{X}$ is the input data block space of x and $\mathcal{Y}$, the output data block space of y. The notion of security for a PRF states that for a uniformly at random chosen key k, the function $\mathcal{F}(k,\cdot)$ should be indistinguishable from a true random function for a PPT adversary.

Similarly to pseudorandom bitstrings, it does not make sense to say that a single function is random. We define a function family that takes inputs from $\mathcal{X}$ and maps them to $\mathcal{Y}$. We then pick uniformly at random a function from this family and compare it to our pseudorandom function $\mathcal{F}(k,\cdot)$. The family of all functions mapping $\mathcal{X}$ to $\mathcal{Y}$ can be denoted as Funs$[\mathcal{X},\mathcal{Y}]$.

TABLE 2.1: The function family Funs$[\mathcal{X},\mathcal{Y}]$ mapping 1 bit inputs to 1 bit outputs.

function family $f:\mathcal{X}\to\mathcal{Y}$		f_1	f_2	f_3	f_4
		outputs: y			
input: x	0	0	0	1	1
	1	0	1	0	1

This function family in Table 2.1 only contains 4 functions but for larger input and output spaces the size of the function family grows drastically: $|\text{Funs}[\mathcal{X},\mathcal{Y}]| = |\mathcal{Y}|^{|\mathcal{X}|}$, e.g., for $\mathcal{X} : \{0,1\}^3$ and $\mathcal{Y} : \{0,1\}^4$, the family already holds 2.81e14 functions. Note that the input and output do not necessarily have to be of the same size. The PRF's security depends on its indistinguishability from a function picked at random from this vast function family.

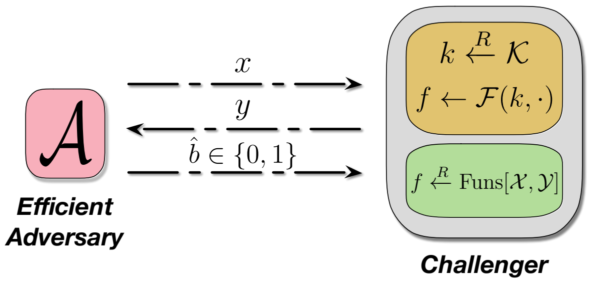

x
y
$\hat{b} \in \{0,1\}$
$\mathcal{A}$
Efficient Adversary
$k \xleftarrow{R} \mathcal{K}$
$f \leftarrow \mathcal{F}(k,\cdot)$
$f \xleftarrow{R}$ Funs$[\mathcal{X},\mathcal{Y}]$
Challenger

Figure 2.3: A PRF, for a given key k, must be computationally indistinguishable from a function taken uniformly at random from Funx$[\mathcal{X},\mathcal{Y}]$ for any efficient algorithm $\mathcal{A}$.

Again, to define security of the PRF more formally, we introduce a new attack game. For a given PRF $\mathcal{F}$, defined over $(\mathcal{K},\mathcal{X},\mathcal{Y})$, and for a given adversary $\mathcal{A}$, we define the following game:

1. The challenger randomly picks b, with $b \in \{0,1\}$ and computes $f \in$ Funs$[\mathcal{X},\mathcal{Y}]$:
   - if $b = 0$: The challenger picks at random a key $k \xleftarrow{R} \mathcal{K}$, and instantiates a function $f \leftarrow \mathcal{F}(k,\cdot)$
   - if $b = 1$: The challenger picks at random a function from the family $f \xleftarrow{R}$ Funs$[\mathcal{X},\mathcal{Y}]$.
2. Next, the adversary submits a query $x \in \mathcal{X}$ to the challenger
3. The challenger evaluates the function $y \leftarrow f(x)$ with $y \in \mathcal{Y}$ and sends the result y to the adversary.
4. The adversary computes and outputs $\hat{b} \in \{0,1\}$.

The attacker wins the game if he can distinguish with a non-negligible probability the output generated by the pseudorandom function from the output generated by the truly random function. The event that $f \leftarrow \mathcal{F}(k,\cdot)$ is the denoted asW_0, and W_1 is the event that $f \xleftarrow{R}$ Funs$[\mathcal{X},\mathcal{Y}]$.

$$\text{PRFadv}[\mathcal{A},\mathcal{F}] = \left|\Pr\left[\mathcal{A}(W_0)=1\right] - \Pr\left[\mathcal{A}(W_1)=1\right]\right| \tag{2.7}$$

Definition 2.1.7. Secure PRF A PRF $\mathcal{F}$ is secure if the value PRFadv$[\mathcal{A},\mathcal{F}]$ is negligible for all efficient adversaries $\mathcal{A}$.

2.1.3 Pseudorandom Permutations

A PRP (Pseudorandom Random Permutation) is an efficient, deterministic permutation $\mathcal{P}$ defined over the spaces $(\mathcal{K}, \mathcal{X})$. It is very similar to a PRF, but there are some additional constraints. The input data block space and output data block space are the same finite set $\mathcal{X}$. On top of being computationally indistinguishable from the permutation picked at random from the permutation family, a PRP is a bijective function for which exists an efficient inversion algorithm. The PRP primitive is the abstract model of a block cipher, which we discuss later in the chapter.

The attack game for a given PRP $\mathcal{P}$, and for a given adversary $\mathcal{A}$ is formalized as follows:

1. Again, the challenger picks b at random with $b \in \{0,1\}$ and selects $p \in \text{Perms}[\mathcal{X}]$ as follows:
   - if $b = 0$: The challenger picks at random a key $k \overset{R}{\leftarrow} \mathcal{K}$, and instantiates a permutation $f \leftarrow \mathcal{P}(k, \cdot)$
   - if $b = 1$: The challenger picks at random a permutation from the family $f \overset{R}{\leftarrow} \text{Funs}[\mathcal{X}, \mathcal{Y}]$.
2. The adversary submits a query $x \in \mathcal{X}$ to the challenger
3. The challenger computes $y \leftarrow p(x)$ with $y \in \mathcal{X}$ and sends y to the adversary.
4. The adversary computes and outputs $\hat{b} \in \{0,1\}$.

The attacker wins if he can distinguish with a non-negligible probability the pseudorandom permutation from the truly random permutation. The event that $p \leftarrow \mathcal{P}(k, \cdot)$, is denoted as W_0 and W_1 is the event that $p \overset{R}{\leftarrow} \text{Perms}[\mathcal{X}]$.

$$\text{PRPadv}[\mathcal{A}, \mathcal{P}] = \left| \Pr\left[\mathcal{A}(W_0) = 1\right] - \Pr\left[\mathcal{A}(W_1) = 1\right] \right| \tag{2.8}$$

Definition 2.1.8. Secure PRP A PRP $\mathcal{P}$ is secure if the value $\text{PRPadv}[\mathcal{A}, \mathcal{P}]$ is negligible for all efficient adversaries $\mathcal{A}$.

2.1.4 One-Way Functions

Pseudorandom generators, functions, and permutations form the building blocks of all the symmetric cryptography we will further discuss in this chapter. Symmetric cryptography can be proven secure under the assumption that the pseudorandom generators and functions exist. Currently, an unconditional proof of the existence of these primitives is not possible[2]. To work around this problem cryptographers formulated a minimal assumption: the existence of one-way functions [31]. If one-way functions exist, the pseudorandom constructions are achievable.

One-way functions are functions that are easy to compute but hard to invert. The latter means that a PPT adversary only has negligible probabilty on success when trying to invert the function within a reasonable time. Let $\mathcal{H}$ be an efficient one-way function defined over $(\mathcal{M}, \mathcal{T})$. We define the following attack game to reason about the one-wayness of $\mathcal{H}$:

1. The challenger chooses $m \in \mathcal{M}$ at random, computes $t \leftarrow \mathcal{H}(m)$ and sends t to $\mathcal{A}$
2. The adversary computes $m' \in \mathcal{M}$
3. The adversary wins if $\mathcal{H}(m') \rightarrow t$

$$\text{OWadv}[\mathcal{A}, \mathcal{H}] = \left| \Pr\left[\mathcal{A}(\mathcal{H}(m)) \in \mathcal{H}^{-1}(\mathcal{H}(m))\right] \right| \tag{2.9}$$

Definition 2.1.9. Secure one-way function We say that $\mathcal{H}$ is a secure one-way function if the adversary's advantage over the one-way function, denoted as $\text{OWadv}[\mathcal{A}, \mathcal{H}]$, is negligible for every PPT adversary.

We will now briefly present two problem statements that are believed to be one-way.

[2]It would require a proof for the famous mathematical problem $\mathcal{NP} \neq \mathcal{P}$.

2.1.4.1 The Factoring Assumption

The factoring problem can be described as finding positive integers p, q, where p and q are prime[3], such that $pq = N$, for a given N. Naive algorithms can solve the problem in exponential time by using a trial division approach: check if p divides N for all p in $[2, \sqrt{N}]$. Although better algorithms are known, nobody has succeeded in finding a PPT algorithm that correctly solves the factoring problem.

2.1.4.2 Discrete Logarithm Assumption

The DLP (Discrete Logarithm Problem) can be defined in any cyclic group, although it is only considered hard in a few [31]. Groups are basic algebraic structures. We do not provide their definition but refer the interested reader to Shoup et al. [33].

If $\mathbb{G}$ is a cyclic group of order q, then there exists a generator $g \in \mathbb{G}$ such that $\{g^0, g^1, \dots g^{q-1}\} = \mathbb{G}$. For every $h \in \mathbb{G}$ there is a unique $x \in \mathbb{Z}_q$, such that $g^x = h$. We call x the discrete logarithm of h with respect to g. The problem to solve can be defined as follows:

- An efficient algorithm $\mathcal{G}$ generates a cyclic group $\mathbb{G}$ with order q and generator g.
- The challenger picks $x \leftarrow \mathbb{Z}_q$ and calculates $h = g^x$.
- The adversary gets $\mathbb{G}, q, g, h$ and outputs $x' \in \mathbb{Z}_q$

The adversary wins the game if $g^{x'} = h$. The discrete logarithm assumption is the assumption that there exists an efficient algorithm $\mathcal{G}$ for which the discrete logarithm is hard. The DLP is the core component of the famous Diffie-Hellman key establishment algorithm.

2.2 Symmetric-Key Cryptography

In symmetric-key cryptosystems, all the communicating parties possess the same secret key. This key is used in conjunction with a cipher to convert, i.e., encrypt and decrypt, between plaintext and ciphertext. Additionally, the secret can be used to generate and verify a MAC (Message Authentication Code) to protect the message from tampering.

In the previous chapter, we saw that the length of the key is related to the security parameter of the cryptosystem. Thus, the length of the key plays vital role in the cryptosystem's resilience against brute force attacks. In a brute force attack, the adversary tries to decrypt the plaintext message by trying all the possible keys. Because the computational power of computers increases year after year, the key length of the encryption scheme should be chosen conservatively if the ciphertext must resist brute force attacks in the upcoming decade. Currently, NIST (National Institute of Standards and Technology) recommends key lengths of 128 bits for securing data whose "security life" extends the year 2031 and beyond [22].

2.2.1 Stream Ciphers

Stream ciphers individually encrypt the bits of the plaintext message. It calculates the XOR (Exclusive Or) operation between a bit from the key and a plaintext bit. Decryption is the inverse operation; the cipher recovers the plaintext by calculating the bitwise XOR between the key bits and the ciphertext. The most famous stream cipher is undoubtfully the OTP, which we shortly discussed in the previous Section 2.1. The length of the key and the requirement for it to be genuinely random make the OTP impractical. In practice, we pick a short uniformly random key, from which a secure PRG can produce a keystream. Encryption and decryption are defined as follows,

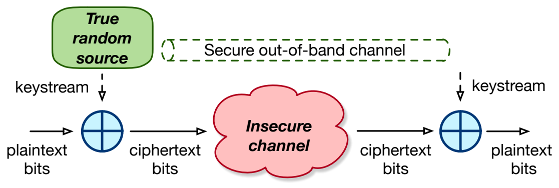
$$E(s, m) \rightarrow \mathcal{G}(s)[0 \dots v-1] \oplus m \quad \text{with } |m| = v \tag{2.10}$$

and decryption,

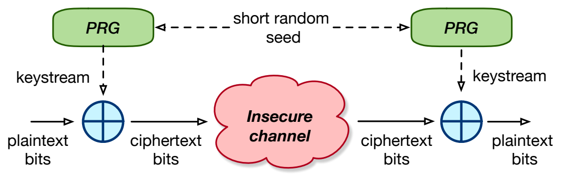
$$D(s, c) \rightarrow \mathcal{G}(s)[0 \dots v-1] \oplus c \quad \text{with } |c| = v \tag{2.11}$$

[3]To generate the large prime numbers, we can use an efficient algorithm such as the Miller-Rabin primality test [32].

It is important to note that the use of a PRG reduces but does not eliminate the need for a strong source of random bits. The PRG is a "randomness expander", it must be fed a truly random seed to generate a pseudorandom sequence.



(a) Block diagram of a one-time pad cryptographic system.



(b) Block diagram of a stream cipher.

Figure 2.4: The difference between the one-time pad and a stream cipher.

Stream ciphers instantiated from a secure PRG are CPA-secure. For a chosen-plaintext, the corresponding ciphertext allows the attacker to recover a part of the keystream. However, if $\mathcal{G}$ is a secure PRG, the attacker does not learn anything about previous or successive keystream bits. Hence, the attacker cannot learn anything extra about remaining encrypted message parts.

Even when using a secure stream cipher, an adversary still has powerful attacks at his disposal. If an attacker can force a stream cipher to encrypt two messages with the same part of the keystream, the attacker can break the cipher. If the attacker intercepts two ciphertexts c_1 and c_2, both encrypted with the same seed s, he can obtain the XOR of both original plaintext messages:

$$\Delta = c_1 \oplus c_2 = (m_1 \oplus \mathcal{G}(s)) \oplus (m_2 \oplus \mathcal{G}(s)) = m_1 \oplus m_2 \tag{2.12}$$

One solution would be to change the seed for every encrypted message. In practice, stream ciphers, such as ChaCha20, often take an additional input called a *nonce*. The underlying PRG can then be defined as a function $\mathcal{G} : \mathcal{S} \times \mathcal{N} \rightarrow \mathcal{R}$. The space $\mathcal{N}$ is called the nonce space. By adding a nonce to the algorithm we can generate multiple pseudorandom outputs from a single seed s. The value of the nonce can be arbitrary, but it should never be used twice for the same seed.

A second vulnerability arises when the communication parties send each other unauthenticated ciphertext. An adversary can change bits in the ciphertext at will without being detected. This attack is called a bit-flipping attack (and belongs to the CCA threat model). Suppose an attacker intercepts a ciphertext $c = E(s, m) = m \oplus \mathcal{G}(s)$. He can change c to $c' = c \oplus \Delta$ for some Δ of the attacker's choice. The receiver then gets the following plaintext message

$$D(s, c') = c' \oplus \mathcal{G}(s) = (c \oplus \Delta) \oplus \mathcal{G}(s) = m \oplus \Delta. \tag{2.13}$$

The attacker has, without knowledge of either m or s, manipulated the plaintext message predictably. This unwanted property is called *malleability*. In Section 2.2.5, we introduce AE (Authenticated Encryption) ciphers, which are specifically designed to prevent this type of attack.

2.2.2 Block Ciphers

Block ciphers operate on chunks of data, denoted as blocks. A block cipher takes a key k and a block of plaintext b, both with a fixed size, and outputs a block of ciphertext. As noted in Section 2.1, a PRP is an abstract model of a block cipher. A practical PRP consists of two functions:

1. A simple non-linear function that takes a key and a block of data as input
2. A key expansion function, which is a PRG that is used to expand the key k into n keys $k_1, \ldots, k_n$.

In the context of block ciphers, the simple non-linear function is called a *round function*, and multiple of those are chained together to form a secure encryption algorithm. The round functions take each one of the expanded keys, called *round keys*. The encryption $E(k, x)$ and decryption $D(k, y)$ algorithms are defined as follows:

- **Key expansion:** use the key expansion function $\mathcal{G}$ to stretch the key k to n keys, one for each round function $\hat{E}$:
$$(k_1 \ldots, k_n) \leftarrow \mathcal{G}(k) \tag{2.14}$$

- **Encryption:** for $i \ldots, n$ apply $\hat{E}(k_i, \cdot)$:
$$y \leftarrow \hat{E}(k_n, \hat{E}(k_{n-1} \ldots, \hat{E}(k_1, x) \ldots)) \tag{2.15}$$

- **Decryption:** for $i \ldots, n$ apply $\hat{D}(k_i, \cdot)$:
$$x \leftarrow \hat{D}(k_1, \hat{D}(k_2 \ldots, \hat{D}(k_n, y) \ldots)) \tag{2.16}$$

Interestingly, the round function $\hat{E}$ on its own is not a secure block cipher, but heuristic evidence suggests that security of a block cipher comes from iterating it many times. There is no rigorous method that measure security of a round function and derives how many times it must be iterated before it becomes a secure block cipher. We only know that certain functions, like linear functions, never lead to secure block ciphers, while simple non-linear functions appear to give a secure block cipher after a few iterations [27].

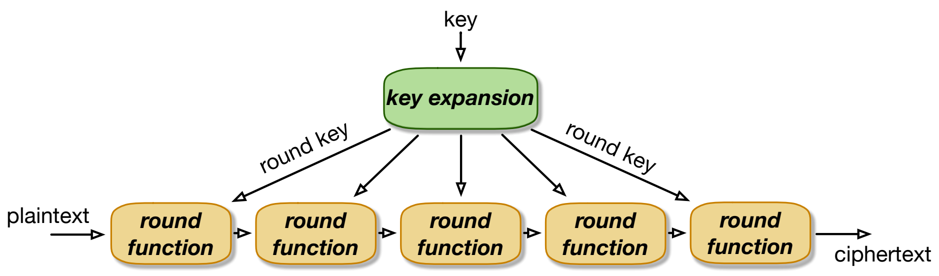


Figure 2.5: General construction principles of a block cipher using the iterated cipher paradigm.

Advanced Encryption Standard is currently the most widely deployed block cipher. AES (Advanced Encryption Standard) was chosen in October 2000 by NIST as the successor of DES (Data Encryption Standard). The original algorithm, named Rijndael after its inventors Vincent Rijmen and Joan Daemen, was picked after a three-year-long public competition. AES uses a series of permutations and substitutions and therefore executes fast when implemented in both hardware and software. It has a 16-byte block size and uses keys of length 128, 192, and 256 bits. Following the design principles we described above, AES uses a chained round function to encrypt plaintext. Depending on the key length, it either uses 10, 12, or 14 combinations of the AES round function. A high-level overview of the AES function is shown in Algorithm 1.

Algorithm 1 AES block encryption.

```
1: procedure ENCRYPTION(State, Key)
2:     KeyExpansion(Key, RoundKey)
3:     AddRoundKey(State, RoundKey[0])
4:     for $0 < i < N_R$ do
5:         RoundFunc(State, RoundKey[$i$])
6:     FinalRound(State, RoundKey[$N_R$])
```

At the start of the encryption function, a key expansion takes place, which creates $N_R + 1$ round keys, where N_R is the number of round functions. After the expansion, the encryption algorithm applies an initial `AddRoundKey`. The `AddRoundKey` consists of a bytewise XOR between the current internal AES state and the round key. Next, the algorithm executes N_R times the round function, see Algorithm 2.

The encryption finishes with one application of the `FinalRound`, see Algorithm 3. Note that `FinalRound` differs from `RoundFunc` only in the absence of `MixColumns` step.

Internally, the `RoundFunc` applies subsequently, `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`. The `SubBytes` step is the substitution step of the cipher, where each byte in the state is substituted

Algorithm 2 AES round function.

```
1: procedure ROUNDFUNC(State, ExpandedKey[$i$])
2:     SubBytes(State)
3:     ShiftRows(State)
4:     MixColumns(State)
5:     AddRoundKey(State, ExpandedKey[$i$])
```

Algorithm 3 AES final round function.

```
1: procedure ROUNDFUNC(State, ExpandedKey[$N_R$])
2:     SubBytes(State)
3:     ShiftRows(State)
4:     AddRoundKey(State, ExpandedKey[$N_R$])
```

using a Rijndael box (S-Box). Practically, this results in a table lookup operation for each byte of the 16 bytes in the AES state. The main property of the `SubBytes` step is non-linearity, and it is the only non-linear transformation of the cipher. S-box values are carefully constructed and are paramount for the security of the cipher. The `ShiftRows` step cyclically shifts rows of the state represented as a 4×4 matrix over different offsets. The four offsets have to be different to achieve resistance against differential and linear cryptanalysis. The first row of the state matrix is not shifted, while the second, the third, and the fourth rows are shifted with offsets one, two, and three, respectively. The `MixColumns` step is another linear transformation where each column of the state matrix is multiplied modulo $x^4 + 1$ with a fixed polynomial. The coefficients of the polynomial were selected in a way the multiplication can be implemented very efficiently even on 8-bit processors. The AES decryption algorithm performs the inverse steps to recover the original plaintext.

2.2.3 Block Cipher Modes of Operation

While a secure stream cipher allows us to encrypt messages of arbitrary length, block ciphers only encrypt data blocks of a fixed size. In reality, we often want to encrypt messages that are much longer than the block size. To do so, we split the message in chunks of the appropriate block size, potentially by adding padding bytes to the last block, and then we encrypt each block separately with the same key.

In Section 2.2.1, we saw that we need to be extremely careful when reusing keys. Reusing the same key to encrypt multiple message chunks with any block cipher is insecure. To make key reuse possible, we need to use a probabilistic cipher. Probabilistic ciphers introduce randomness in the encryption process. The randomness often comes under the form of an IV (Initialization Vector). Next, we discuss the different block cipher modes of operation, which allow us to build CPA-secure block ciphers. We present three common modes: ECB (Electronic Codebook), CTR (Counter Mode), and CBC (Cipher Block Chaining).

2.2.3.1 Electronic Codebook

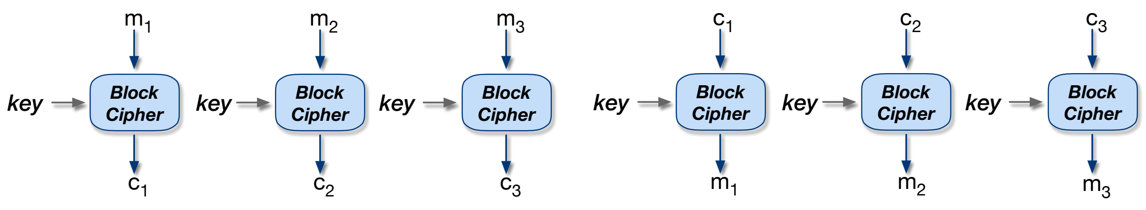
The ECB mode, shown in Figure 2.6, is the most straightforward mode of operation; however, it is not CPA-secure. The cipher cuts the message M into blocks, $(m_1, m_2, m_3, \ldots, m_k)$ with a length that corresponds to the block size of the cipher. Padding is applied when the plaintext is not a multiple of the block size. The block cipher encrypts each block of plaintext message individually.

$$c_i = E(k, m_i) \quad \text{with } i \in [1, k] \tag{2.17}$$

The ECB decryption mode takes a similar approach. The ciphertext is cut up in blocks of the appropriate length, and the procedure decrypts each block independently.

$$m_i = D(k, c_i) \quad \text{with } i \in [1, k] \tag{2.18}$$

A block cipher should never use plain ECB. The ECB mode has the undesired property that identical plaintext result in identical ciphertext. As fragments of plaintext tend to repeat (network protocol headers, specific application data) an attacker could mount an attack based on statistical analysis.



(a) Encryption. (b) Decryption.

Figure 2.6: A block cipher configured in ECB mode.

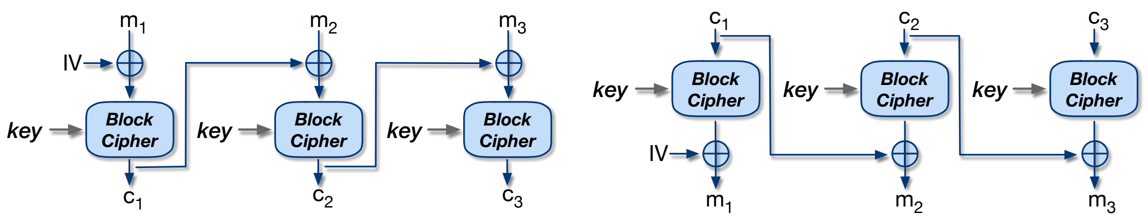
2.2.3.2 Cipher Block Chaining

The CBC mode is a CPA-secure mode of operation for block ciphers, depicted in Figure 2.7. In order to remove the property of ECB that identical plaintext blocks result in identical ciphertext, CBC introduces a dependency between subsequent ciphertext blocks: the i^{th} ciphertext block is obtained by encrypting the i^{th} plaintext block with $(i-1)^{\text{th}}$ ciphertext block.

$$c_i = E(k, x_i \oplus m_i) \quad \text{with } x_i = \begin{cases} \text{IV} & \text{if } i = 1 \\ c_{i-1} & \text{if } i > 1 \end{cases} \tag{2.19}$$

The inter-block dependency ensures that two identical plaintext blocks in a message result in different ciphertext blocks. However, two identical messages will still result in an identical ciphertext, which leaks information about the plaintext. The CBC mode overcomes this issue by introducing some randomness in the encryption process. The algorithm calculates the XOR between the first plaintext block and a randomly chosen IV. When using different IVs, the same plaintext will produce different ciphertext when encrypted under the same key k. To decrypt a probabilistically encrypted message, the recipient recovers the IV from the received message and calculates the XOR between the IV and the first decrypted block. The IV is not a secret, and it is sent in the clear to the receiver for the decryption process.

$$m_i = x_i \oplus D(k, c_i) \quad \text{with } x_i = \begin{cases} \text{IV} & \text{if } i = 1 \\ c_{i-1} & \text{if } i > 1 \end{cases} \tag{2.20}$$



(a) Encryption. (b) Decryption.

Figure 2.7: A block cipher configured in CBC mode.

2.2.3.3 Counter Mode

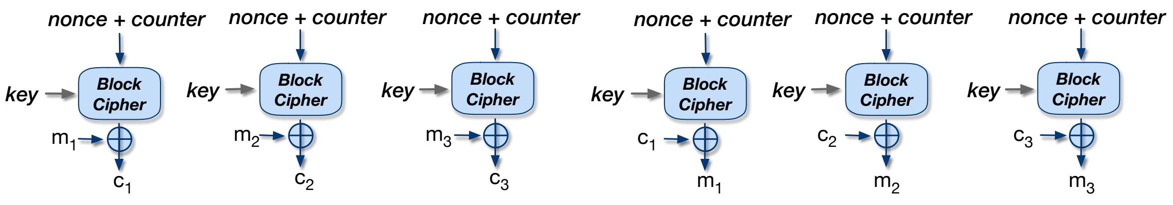
The CTR mode is a CPA-secure block cipher mode that allows for parallel encryption and decryption, see Figure 2.8. CTR mode uses the IV as the initial value of a monotonic counter to encrypt/decrypt successive blocks. CTR mode does not pass the plaintext blocks through the block cipher encryption primitive. Instead, it merely calculates the XOR of the plaintext block and the encrypted monotonic counter.

$$c_i = m_i \oplus E(k, IV_i) \quad \text{with } i \in [1, k] \tag{2.21}$$

Decryption is simply the inverse operation.

$$m_i = c_i \oplus E(k, IV_i) \quad \text{with } i \in [1, k] \tag{2.22}$$

The CTR mode emulates a stream cipher. The output of the AES algorithm is used as a keystream. The message is bitwise XORed with the output of the algorithm. The counter is typically constructed by taking a random nonce and concatenating it with a counter that increments for each encrypted block. CTR mode allows for parallel encryption and decryption because each block is independent.



(a) Encryption.

(b) Decryption.

Figure 2.8: A block cipher configured in CTR mode.

It is imperative that for both the CBC and CTR modes, the same IV and key pair is never repeated. More specifically, for the CBC mode, the IV must be completely unpredictable[4]. For CTR mode the initial IVs must be sufficiently spread out in the IV space such that two different counters never collide for a same key.

2.2.4 Message Authentication Codes

Looking back at the four properties we described at the start of this chapter, we notice that all previously described constructions only provide confidentiality. To protect our message from malicious alteration by an active adversary, we turn our attention to the next two properties: data integrity and data origin authentication. An algorithm that provides data integrity and data origin authentication uses a shared key between the sender and the receiver and is called a MAC.

Definition 2.2.1. MAC system A MAC system $\mathcal{I}(S,V)$ consists of a pair of efficient algorithms, S and V. The algorithm S generates a tag and the algorithm V verifies the tag.

- S is a probabilistic algorithm. It is invoked as $t \overset{R}{\leftarrow} S(k,m)$, where k is a key, m is a message.
- V is a deterministic algorithm. It is invoked as $r \leftarrow V(k,m,t)$, where k is a key, m is a message and t is the tag generated by S. The output of the algorithm is either `accept` or `reject`.

To reason about the security of MAC systems, we can describe a new attack game. During the game, the attacker can request the challenger many tags $t \overset{R}{\leftarrow} S(k,m)$ on messages of his choice. This attack is called the *chosen message attack*. Using the chosen message attack, the attacker must come up with a new valid pair (m,t). The attacker can freely choose m. If the attacker succeeds, he has found an *existential MAC forgery*. The formal description of the attack game goes as follows:

1. The challenger picks a secret key at random, $k \overset{R}{\leftarrow} \mathcal{K}$.
2. $\mathcal{A}$ queries the challenger i times. For $i = 1,2,\ldots$, the i^{th} signing query is the message $m_i \in \mathcal{M}$. The challenger returns the tag $t_i \overset{R}{\leftarrow} S(k,m_i)$.
3. After a while $\mathcal{A}$ outputs a candidate forgery pair $(m_f,t_f) \in \mathcal{M} \times \mathcal{T}$. The candidate forgery cannot appear in the list of previously specified pairs.

$$(m_f,t_f) \notin \Big\{(m_1,t_1),(m_2,t_2),\ldots\Big\} \tag{2.23}$$

The attaker $\mathcal{A}$ wins the game if (m_f,t_f) is a valid pair under the key k (i.e., $V(k,m,t) =$ `accept`). His advantage with respect to the MAC system $\mathcal{I}$ is denoted as MACadv[$\mathcal{A}$,$\mathcal{I}$]. It expresses the probability that $\mathcal{A}$ wins the attack game.

Definition 2.2.2. Secure MAC system We say that a MAC system $\mathcal{I}$ is existentially unforgeable under a chosen message attack if for all efficient adversaries $\mathcal{A}$, the value MACadv[$\mathcal{A}$,$\mathcal{I}$] is negligible.

[4]BEAST is a well-known attack on TLS1.0 which exploits a predictable IV in CBC mode to recover the plaintext

We can build MAC systems in several ways: based on pseudorandom functions, by using universal hashing and from collision resistant hash functions. The MAC systems can be deterministic or randomized. In a deterministic system, for a given key k and a given message m, there is only one valid tag. A randomized MAC uses a randomized signing algorithm and for a given key k and message m, there are many valid tags.

2.2.4.1 Message Integrity using Pseudorandom Functions

We can build a secure MAC from any secure PRF. As discussed in Section 2.1, a PRF is an algorithm $\mathcal{F}$ that takes two inputs, a key k and an input data block x, and outputs a value $y \leftarrow \mathcal{F}(k,x)$. The PRF switching lemma [27] states that any secure PRP is a secure PRF. Thus, we can use block ciphers such as AES as a real-life instantation of a PRF. For a given PRF $\mathcal{F}$, we define the deterministic MAC system as follows:

$$S(k,m) = \mathcal{F}(k,m) \quad \text{and} \quad \mathcal{F}(k,m,t) = \begin{cases} \texttt{accept} & \text{if } \mathcal{F}(k,m) = t \\ \texttt{reject} & \text{otherwise} \end{cases} \tag{2.24}$$

A problem arises if we want to create integrity tags for messages that are larger than the block size of the chosen PRF, i.e., AES has a block size of 128 bits. To extend the signing domain of the MAC algorithm S, we can reuse the CBC mode as described in Section 2.2.3. The encryption of the last block of plaintext is dependent, in a complicated way, on the encryption of all the preceding plaintext blocks. In some sense, the last block of ciphertext provides a cryptographic accumulation of the entire encryption process [23]. The CBC-MAC (Cipher Block Chaining Message Authentication Code) primitive builds on this intuition.

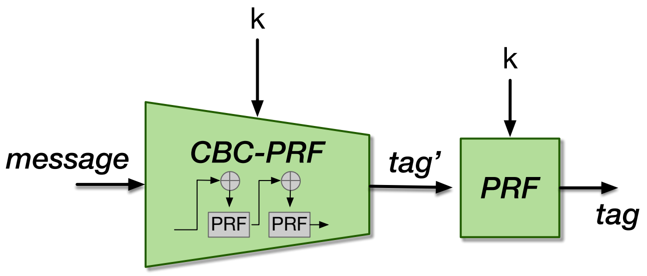


Figure 2.9: The ECBC MAC protects against length-extension attacks by appending a final iteration of the PRF.

CBC-MAC computes a message authentication tag by encrypting the plaintext with a block cipher in CBC mode, using an initial all null IV, and throwing away all intermediate ciphertext blocks but the last. A naive implementation of the CBC-MAC, as described above, is only secure for fixed-length messages. When trying to use this primitive for variable-length messages, it becomes vulnerable to a *length extension attack* [27, 34]. The attack goes as follows:

1. First the attacker gets a valid tag t on a message m_1.
2. The attacker calculates $t \oplus m_2$, where m_2 is some arbitrary message chosen by the attacker.
3. The attacker feeds $t \oplus m_2$ into the CBC-MAC algorithm and gets a tag t' on the message $t \oplus m_2$
4. The resulting tag t' is a valid tag for the combined message $m_1 || m_1$.

To protect ourselves from a length extension attack the tag t' can be encrypted by applying the PRF one last time before returning the tag t to the attacker, this construction is known as the ECBC (Encrypted-CBC) and is shown in Figure 2.9. This way, the tag t cannot be used as a prefix to request a new tag on the combination of m_1 and m_2. Alternatively, we could prepend the length of the message to the first block of the plaintext before calculating the CBC-MAC.

2.2.4.2 Message Integrity using Universal Hashing

A second technique to build a secure MAC system uses a UHF (Universal Hash Function), followed by a secure PRF, a.k.a. the *hash-then-PRF* paradigm. Before we can explain this construct any further, we introduce a new primitive, called the *hash function*.

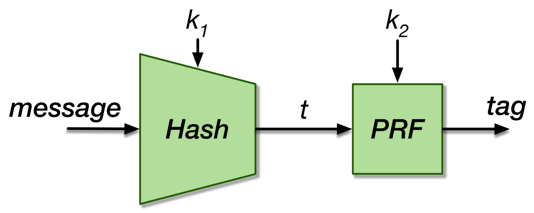


Figure 2.10: The hash-then-PRF paradigm to construct a secure MAC system.

A hash function H is a function that maps inputs from a vast space $\mathcal{M}$ to a smaller output space $\mathcal{T}$. The elements of $\mathcal{T}$ are often called *message digests*. Since the digest space $\mathcal{T}$ is typically much smaller than the message space $\mathcal{M}$, many collisions can occur (by the pigeonhole principle).

Definition 2.2.3. Collision Two messages $m_0, m_1 \in \mathcal{M}$ form a collision for the hash function H if

$$H(m_0) = H(m_1) \quad \text{and} \quad m_0 \neq m_1 \tag{2.25}$$

When considering hash functions, we usually deal with a family of hash functions. The functions are indexed by a key s. This key is not secret. It is merely a means to specify a function H in the family $\mathcal{H}$. A hash function family is called *universal* if for a randomly picked $H \in \mathcal{H}$, the probability that two distinct messages m_0 and m_1 collide is at most $1/|\mathcal{T}|$.

$$\left|\Pr\Big[H(m_0) = H(m_1)\Big]\right| \leq \frac{1}{|\mathcal{T}|} \tag{2.26}$$

In other words, a universal hash function family is equivalent to having a probability distribution on the functions from $\mathcal{M}$ to $\mathcal{T}$ that map elements of $\mathcal{M}$ uniformly at random to elements of $\mathcal{T}$.

For cryptographic applications, we need some stronger properties than the hash function simply "being good" at avoiding collisions. We require that no PPT adversary can find collisions. This property is called *collision resistance*. Collision resistance is a strong security requirement that is difficult to achieve. There exist other levels of security which are useful in cryptographic hash functions. We list them in order of increasing strength:

- **Pre-image resistance or one-wayness:** Given only a message digest $H(m)$, it is difficult to find any message (or pre-image), m' that generates that digest, $H(m) = H(m')$.
- **Second pre-image resistance:** Given a message m and its digest $H(m)$, it is difficult to find another message that has the same message digest, $H(m) = H(m')$.
- **Collision resistance:** It is difficult to find two different messages, $m_1 \neq m_2$ with the same message digest $H(m_1) = H(m_2)$.

A UHF that has second pre-image resistance is called a UOWHF (Universal One-Way Hash Function) and can be used to build a MAC. In a UOWHF attack game, the challenger picks a challenge m, then fixes the UOWHF by selecting one randomly from the hash function family, and finally proposes the challenge, $H(m)$, to the adversary. The adversary wins if he finds another input m' that hashes to the same value.

UOWHFs can be constructed from polynomials modulo a prime. Let l be a length parameter and let p be a prime. We define a hash function H_{poly} that hashes a message $m \in \mathbb{Z}_p^{\leq l}$ to a single element $t \in \mathbb{Z}_p$. The key space is $\mathcal{S} \leftarrow \mathbb{Z}_p$. Let m be a message, $m = (a_1, a_2, \ldots a_v) \in \mathbb{Z}_p^{\leq l}$ for some $0 \leq v \leq l$. Let $s \in \mathbb{Z}_p$ be a key. The hash function $H_{\text{poly}}(s, m)$ is defined as follows:

$$H_{\text{poly}}(s, (a_1, \ldots, a_v)) = s^v + a \cdot s^{v-1} + \cdots + a_{v-1} \cdot s + a_v \in \mathbb{Z}_p \tag{2.27}$$

Note that this UOWHF can only be used once. An adversary that evaluates the function at a single point can completely recover the key, s. For example, if $m = (a_1)$, since $H_{\text{poly}}(s, m) = s + a_1$, an adversary who has both m and $H_{\text{poly}}(s, m)$ immediately obtains $s \in \mathbb{Z}_p$. Since we only required second pre-image resistance and not the more powerful collision resistance property, it might be trivial for an attacker to find a collision now that he knows s. Consequently, cryptographic applications of UOWHF hide the output from the adversary, either by encryption or by other means. Compared to other MAC systems, UOWHF provides some speed benefits. The polynomials are faster to evaluate than the iterative application of PRFs, i.e., the CBC-MAC.

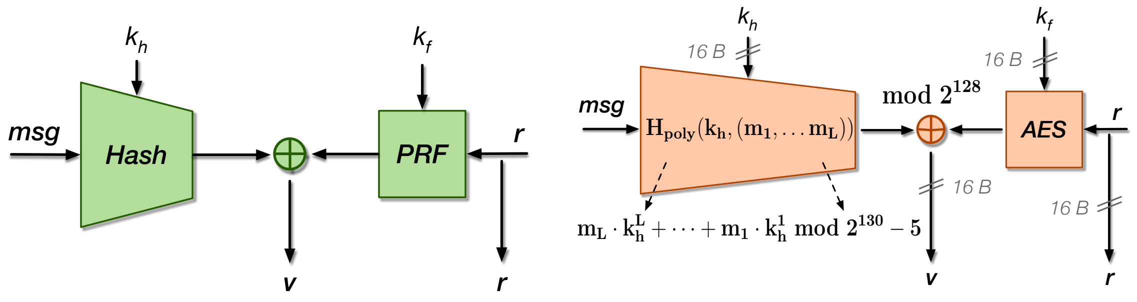
Carter-Wegman MAC is a randomized MAC system based on a UOWHF. There exist multiple valid tags for each message. The Carter-Wegman MAC first fixes some large integer N, e.g., 2^{128}, and sets the tag space $\mathcal{T} \leftarrow \mathbb{Z}_N$, to the (abelian) group of size N where addition is defined modulo N. We use a hash function H and a PRF $\mathcal{F}$ that output values in $\mathbb{Z}_N$:

- H is a UOWHF defined over $(\mathcal{K}_H, \mathcal{M}, \mathcal{T})$
- $\mathcal{F}$ is a PRF defined over $(\mathcal{K}_F, \mathcal{R}, \mathcal{T})$

The Carter-Wegman MAC, shown in Figure 2.11a, takes as input a message $m \in \mathcal{M}$, a pair of keys (k_h, k_f) and a value r, picked uniformly at random from their respective input spaces $(\mathcal{K}_H, \mathcal{K}_F, \mathcal{R})$. The algorithm outputs a tag t as a pair of values in the output space $(\mathcal{T} \times \mathcal{R})$:

$$v \leftarrow H(m, k_h) + \mathcal{F}(r, k_f) \quad \in \mathbb{Z}_N \tag{2.28}$$

The tag t is a tuple of (v, r). To verify a Carter-Wegman tag, the verifier recovers the value r from the tuple and repeats the tag calculation. It compares its result to the received tag. A state-of-the-art example of a Carter-Wegman-style MAC is Poly1305 [35, 36]. The algorithm uses a polynomial modulo the prime $2^{130} - 5$ and sets $N = 2^{128}$. It takes a 256-bit key, and splits it into two parts: the first part is necessary for the evaluation of the polynomial while the second part acts as a key for the PRF, i.e., AES. Poly1305 forms in combination with the stream cipher ChaCha20, an AE cipher. ChaCha20+Poly1305 is included in the latest two versions of the TLS protocol, i.e., version 1.2 & 1.3 [37].


k_h
k_f
msg
Hash
PRF
r
v
r
k_h
16 B
k_f
16 B
msg
$\mathrm{H_{poly}(k_h, (m_1, \dots m_L))}$
mod 2^{128}
AES
r
16 B
16 B
$\mathrm{m_L \cdot k_h^L + \cdots + m_1 \cdot k_h^1 \bmod 2^{130} - 5}$
v
r

(a) Block diagram of the Carter-Wegman MAC system.

(b) Diagram of the Poly1305 MAC, as specified in the original description by Bernstein et al. [36].

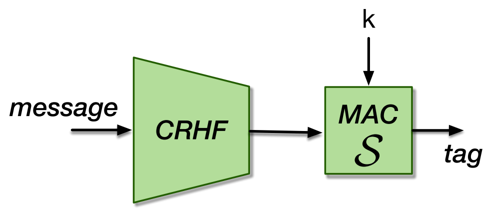
Figure 2.11: Instantiations of the hash-then-MAC paradigm.

2.2.4.3 Message Integrity from Keyless Collision Resistant Hashing

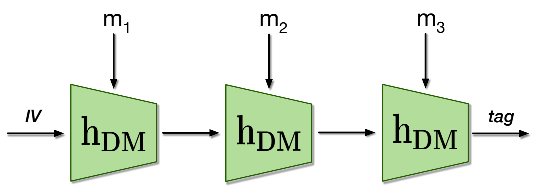
A final method to design MAC systems builds on CRHFs (Collision-Resistant Hash Functions). Similarly to the previous constructions we discussed, the MAC system extends its domain by applying a hash function to the input. While both the UHF and CBC constructions use keyed hash functions, this time, the construction uses a keyless hash function.

The construction is known as the *hash-then-MAC* paradigm. The hash-then-MAC construction looks similar to the hash-then-PRF composition, but there is a significant difference. A CRHF can extend the input domain of any MAC signing algorithm S, while a UOWHF can only use a PRF as a signing algorithm. Recall that we must use a secure PRF as a signing algorithm to hide the details of the key used by the UOWHF; otherwise, the attacker could break the collision resistance of the UOWHF and, consequently, the security of the entire MAC system. In general, the signing algorithm is not required to have this confidentiality property, it should only provide us with tag unforgeability. Keyless CRHFs do not possess any secret that needs to be protected to guarantee the collision resistance. Therefore a simpler signing algorithm could be used.

A keyless CRHF can be designed from number-theoretic primitives such as the discrete logarithm problem or through a block cipher. The most widely used method to turn a block cipher in a CRHF is called Davies-Meyer. The well-known SHA (Secure Hash Algorithm)-family of cryptographic hash functions uses



(a) The hash-then-mac paradigm.



(b) Merkle-Damgård paradigm for variable-input CRHFs.

Figure 2.12: Construction of a CRHF-based MAC function.

Davies-Meyer. The Davies-Meyer hash function derived from the block cipher $\mathcal{E}(E, D)$ maps inputs in $\mathcal{X} \times \mathcal{K}$ to outputs in $\mathcal{X}$. The function is defined as follows:

$$h_{DM}(x, y) = E(y, x) \oplus x \tag{2.29}$$

The Davies-Meyer construction gives us a CRHF, h_{DM}, for short input messages. A frequent approach to construct a CRHF for arbitrary length input messages is to use the Merkle-Damgård paradigm. First, select a CRHF that hashes short messages, e.g., h_{DM}. In this context, h_{DM} is called a *compression function*. Next, chain several instances of the compression function together. Each compression function takes as input the output of the previous compression function and a fixed-sized block of the message, see Figure 2.12b. The first compression function in the Merkle-Damgård construction takes a public, fixed IV as input. Note that the internal block cipher uses the consecutive message blocks m_i as keys. Standard block ciphers, such as AES are built to encrypt long messages with a fixed key. If we change the key on every block it will slow down the CRHF. Block ciphers often have a non-negligible key expansion phase.

Another reason to not use an off-the-shelf block cipher in the Merkle-Damgård construction is that the block size may be too short. Typical block ciphers produce a 128-bit output which is too short for collision resistance. Due to the birthday paradox [38], an attacker can find a collision with only 2^{64} evaluations of the function. In addition, the relatively short keys, result in Merkle-Damgård processing only 128 message bits per round. Instead, Merkle-Damgård uses a custom block cipher designed explicitly for rapid key changes with longer keys (typically, 512-bits or even 1024-bits long) so that the internal block cipher can process many more message bits in every round. For example SHA256 has a message block size of 512 bits and a digest size of 256 bits.

Combining the Merkle-Damgård CRHF with a signing algorithm forms a secure MAC system. In reality, this construct is not widely used because of several reasons:

1. The security of the MAC relies completely on the collision resistance of the hash function. A collision-finding attack, such as a birthday attack, or a more sophisticated one, can be carried out entirely offline.

2. When using the hash-then-MAC paradigm we need two different primitives to calculate a MAC. First, a CRHF and then a signing algorithm, e.g. a PRF.

Hash-Based Message Authentication Codes provide a way to combine a keyless Merkle-Damgård hash function, such as SHA256, with a secret key to implement a secure MAC. An HMAC (Hashed Message Authentication Code) uses a fixed CRHF, H, with a block size of B bytes, i.e., $B = 32$ in case of SHA256 and takes as input a key k and a message m. Additionally, it defines two fixed and different strings of size B called `ipad` and `opad` [39]. An HMAC is computed as follows:

$$H(k \oplus \texttt{opad}, H(k \oplus \texttt{ipad}, m)) \tag{2.30}$$

The HMAC is the most widely deployed MAC on the Internet. Protocols such as TLS, IPSec (Internet Protocol Security), and SSH (Secure Shell) use HMACs abundantly, among others as a means for deriving session keys during session setup.

2.2.5 Authenticated Encryption

An AE scheme combines an encryption algorithm and a MAC algorithm in a single cryptographic construction. AE-secure ciphers provide us with confidentiality, integrity, and authenticity. It is the only scheme that is secure in the strongest threat model, i.e., CCA. There are roughly two mechanisms to build AE ciphers: the first is called the *generic composition* that combines a standalone secure cipher with a standalone secure MAC, the second method builds an AE cipher directly from a block cipher or PRF. These are sometimes called *integrated schemes* or one-pass schemes. They are faster than the composed schemes, which are two-pass, but they are often patented. An example of a one-pass scheme is the OCB (Offset Codebook) mode.

For the generic composition technique three combinations exist:

- **Encrypt-then-MAC** provides a ciphertext with integrity protection. The result of this mode is a ciphertext-tag pair (c, t). Encrypt-then-MAC is supported in TLS version 1.2 and 1.3 as well as in IPSec.
- **MAC-then-Encrypt** computes a MAC over the plaintext, which is subsequently encrypted, resulting in a ciphertext c. Older versions of the TLS protocol use this scheme, but is not always secure. The infamous POODLE [28] attack on SSL (Secure Socket Layer)[5] version 3.0, exploited a flawed MAC-then-Encrypt cipher to recover plaintext messages from the SSL connection.
- **Encrypt-and-MAC** computes a MAC over the plaintext and then encrypts the plaintext, resulting in a ciphertext-tag pair (c, t). The ciphertext is not integrity protected. This method should also be avoided.

Ciphers that are AE-secure posses the following properties: the encryption algorithm must be CPA-secure and the corresponding MAC algorithm must provide ciphertext integrity.

Definition 2.2.4. AE-secure We say that a cipher E is simply AE-secure, if E is CPA-secure and provides ciphertext integrity.

Ciphertext integrity prevents an attacker from exploiting the malleability of a cipher. We briefly, presented the bit-flipping attack in Section 2.2.1 that exploited the lack of ciphertext integrity.

2.2.5.1 Authenticated Encryption with Associated Data

AE ciphers are often extended to accept a nonce and *additional data* as their input, next to the plaintext and key. These schemes are called, AEAD (Authenticated Encryption with Authenticated Data) ciphers. The integrity of the additional data is protected by the cipher but the data is not encrypted. The need for this construction is apparent in the context of networking protocols. Ciphers protect the content of network packets, but the headers of the packet need to remain legible. An AEAD cipher can encrypt the payload of the packet and concurrently, protect the integrity of the header. The AEAD cipher uses the nonce to derive IVs for the internal encryption and MAC algorithms. The nonce should never be repeated; otherwise, the AEAD scheme loses its security guarantees.

CCM: Counter with CBC-MAC is a widely used composed AEAD mode and was designed as a non-patented alternative to OCB. Its implementation became mandatory for the IEEE 802.11I standard, which defines WPA2 (Wi-Fi Protected Access II), as the replacement of the broken WEP (Wired Equivalent Privacy) security protocol. Currently, it is one of the few AEAD ciphers in TLS 1.3, and it is also the designated cipher of IEEE 802.15.4. The CCM (Counter and Cipher Block Chaining) mode [40] is a combination of CTR and CBC, to provide confidentiality and authenticity while using a single key [41] and a nonce. CCM only supports block ciphers with a 128-bit block length, such as AES-128 [42]. CCM is popular in constrained environments because of the following reasons:

- CCM can be configured to use variable-length authentication tags (from 32-bits to 128-bits), thus allowing varying degrees of protection against unauthorized modifications [43]. By limiting the size of the tag, a constrained device has fewer bytes to transmit. The latter presents a clear case where there is a trade-off between energy consumption and security.

[5]SSL is the deprecated name for the TLS protocol. In 1999 TLS 1.0 was introduced.

- CCM uses one cryptographic construction to provide encryption and confidentiality, limiting the overall code size compared to other AE constructions where an additional hash function is required. Additionally, CCM only requires the encryption functionality. The decryption algorithm can be omitted. When combined with AES, which is sometimes supported in hardware on constrained devices [14], it provides a secure, energy-efficient, and fast encryption scheme.

Figure 2.13 shows the schematic of CCM mode. First, CCM authenticates the message. The first block in the CBC chain consists of the nonce, flags and the message length. Next, the cipher appends blocks of "additional data". Then, the algorithm feeds the plaintext blocks into the CBC-MAC. In the second stage, the plaintext is encrypted with a block cipher in CTR mode [44]. The 16-byte counter values are formed by concatenating the flags, nonce value and an incrementing counter.

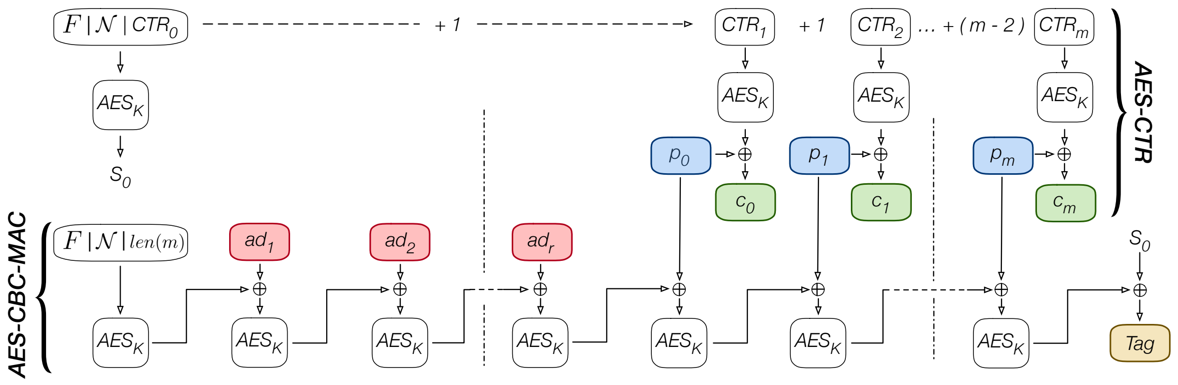


Figure 2.13: Diagram of the CCM authenticated cipher mode.

GCM: Galois Counter Mode is probably the most popular AEAD mode. It is supported in TLS 1.3, IPSec and SSH. It has an Encrypt-then-MAC form. GCM (Galois-Counter Mode) mode is an improvement over an older AEAD mode, called CWC (Carter-Wegman Counter). GCM uses CTR mode encryption with the addition of a Carter-Wegman-style MAC based on the GHASH function, a variant of the H_{poly} function (see Section 2.2.4.2). The input and output are elements in the Galois field $\{0,1\}^{128}$, hence the name GCM. An advantage of GCM over CCM is that the message length does not need to be known in advance. On larger systems, GCM mode is also faster then CCM as both the encryption and the authentication phase can be computed in parallel. Since 2011 Intel introduced a dedicated instruction to quickly compute binary polynomial multiplication, speeding up the GHASH calculations.

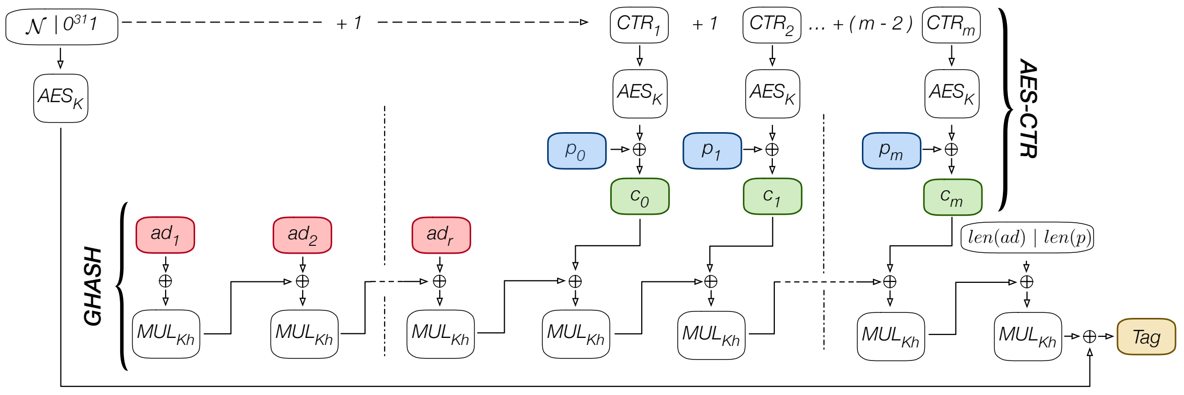


Figure 2.14: Diagram of the GCM authenticated cipher mode.

2.3 Public-Key Cryptography

During our overview of the symmetric-key algorithms we stumbled on some questions for which we have not provided any answers:

- **Key exchange:** Symmetric-key algorithms are well-suited for fast encryption and decryption of data, but both communicating parties need to share a secret key. How can we securely establish a secret key between two entities without relying on some secure out-of-band channel?
- **Non-repudiation:** We discussed several methods to design MAC systems. They provide integrity and authenticity of the message, but fail to pinpoint the source of the message. How can we build a cryptosystem that allows us to identify the source of the data?

These questions can be answered by using public-key cryptography, also known as asymmetric cryptography. We start our discussion on public-key cryptography with a definition of a public-key encryption scheme. We show how to instantiate a public key encryption scheme by using the RSA (Rivest–Shamir–Adleman) trapdoor function. Next, we present the Diffie-Hellman protocol to establish a shared secret between two parties over an insecure channel and, finally the DSA (Digital Signature Algorithm) protocol to provide the non-repudiation property.

2.3.1 Public-Key Encryption

In a public-key encryption system, one party (the receiver) generates a pair of keys (k_p,k_s), denoted as the public key and the private key, respectively. The sender uses the public key to encrypt messages for the receiver. The receiver then uses the private key to decrypt the ciphertext. All interested parties can obtain the public key k_p. This approach stands in stark contrast to symmetric cryptography, where there is a single key, used for both encryption and decryption. The symmetric key must be kept secret at all times.

Definition 2.3.1. Public-key encryption scheme A public-key encryption scheme $\mathcal{E}(G,E,D)$ consists of a triple: a key generation algorithm G, an encryption algorithm E and a decryption algorithm D.

- The probabilistic key generation algorithm is invoked as $(k_p,k_s) \overset{R}{\leftarrow} G$. The keys k_p and k_s are called the public and private key, respectively.
- The probabilistic encryption algorithm is run as $c \overset{R}{\leftarrow} E(k_p,m)$, where m is a message from some finite message space $\mathcal{M}$ and c is the ciphertext from the space $\mathcal{C}$.
- Finally, D is a deterministic decryption algorithm that decrypts the ciphertext as follows, $m \leftarrow D(k_s,c)$.

In the above scheme, we assume that everybody who wants to obtain a legitimate copy of the public key can do so. In practice, this is a non-trivial problem to solve, but we do not discuss it here.

Similarly to symmetric encryption primitives, public-key encryption is CPA-secure when it is probabilistic. The same plaintext encrypted under the same key should not result in the same ciphertext. The main difference with the CPA-secure symmetric ciphers is that the adversary is now in possession of the public key, k_p, allowing him to obtain encrypted plaintexts of his choosing without the need to interact with a challenger. In the symmetric setting, the attacker must ask the challenger for the encryption of a plaintext. The attacker does not possess the secret key. The CPA-security of public encryption schemes can be defined as follows:

1. The probabilistic key generation algorithm generates (k_p,k_s), k_p is given to the adversary.
2. The adversary picks a pair of messages m_0 and m_1 and sends them to the challenger
3. The challenger picks a random bit $b \in \{0,1\}$, computes the ciphertext $c \overset{R}{\leftarrow} E(k_p,m_b)$ and sends the result to the adversary.
4. The attacker still has the power to call the encryption function on other messages of his choice.
5. Finally, The adversary outputs a bit $\hat{b}$.

The public-key encryption scheme $\mathcal{E}(G,E,D)$, is secure if for all PPT adversaries it provides indistinguishable encryptions under chosen-plaintext attacks.

$$\mathrm{INDadv}[\mathcal{A},\mathcal{E}] = \left|\Pr\left[\mathcal{A}(E(k_p,m_0)) = 1\right] - \Pr\left[\mathcal{A}(E(k_p,m_1)) = 1\right]\right| \tag{2.31}$$

Definition 2.3.2. CPA-secure public-key encryption The public-key encryption scheme $\mathcal{E}(G,E,D)$ is secure if for all PPT adversaries $\mathcal{A}$, INDadv$[\mathcal{A},\mathcal{E}]$ is negligible.

RSA named after its inventors Rivest, Shamir, and Adleman, is probably the most famous method to instantiate a public-key encryption scheme. In Section 2.1, we saw examples of one-way functions. Until now, we have not seen any direct applications of these functions. However, at the core of the RSA scheme lies the RSA assumption directly related to the factorization assumption.

If we have a group $\mathbb{Z}_N^*$, where N is the product of two primes p and q, the order of the group can be computed with Euler's totient function $\phi(N) = (p-1)(q-1)$. If the factors, p and q of N are known, we can derive the group order and any computations modulo N can potentially be simplified by working in the exponent modulo $\phi(N)$. If the factorization is not known, finding the group order is difficult. The RSA exploits this asymmetry. The RSA problem can now be described informally as follows: given N, an integer $e > 0$ that is relatively prime to $\phi(N)$, and an element $y \in \mathbb{Z}_N^*$, compute $y^{1/e} \bmod N$. Alternatively, given N, e, y, find x such that $x^e = y \bmod N$. There also exists an integer d that satisfies the equation $ed = 1 \bmod \phi(N)$. This is true if e is invertible modulo $\phi(N)$. The RSA problem thus rests on the assumption that there is no efficient algorithm that can calculate the e^{th} root of m modulo a large composite number N. The fastest known algorithm to calculate the e^{th} root requires the factorization of N. It is possible, however, that there are other ways of solving the RSA problem that do not involve explicit computation of $\phi(N)$, and so we cannot conclude that the RSA problem is as hard as factoring. We now present the RSA algorithm for public-key encryption. The RSA key derivation function is presented in Algorithm 4, and takes as input a security parameter l.

Algorithm 4 RSA key generation.

```
1: procedure RSA KeyGen(l)
2:     Generate $l$-bit prime $p$
3:     Generate $l$-bit prime $q$
4:     Calculate $\phi(N) = (p-1)(q-1)$
5:     Find $e$ such that $\gcd(e, \phi(N)) = 1$
6:     $d \leftarrow e^{-1} \bmod \phi(N))$
7:     return $(d, N)$ and $(e, N)$
```

The pair $k_p \leftarrow (e,N)$ represents the public key, while $k_s \leftarrow (d,N)$ is the private key. The number N is called the RSA modulus, the number e is called the encryption exponent, and d is called a decryption exponent. We can encrypt a message $m \in \mathbb{Z}_N^*$ with a given public key k_p as follows:

$$E(k_p, m) \to c \equiv m^e \bmod n. \tag{2.32}$$

Decryption consists of exponentiation of the ciphertext c with the private key k_s:

$$D(k_s, c) \to c^d \bmod n \equiv (m^e)^d \equiv m^{ed} \equiv m \bmod n \tag{2.33}$$

The above construction is what is called "textbook RSA". This is a deterministic form of the public-key encryption scheme and is thus inherently insecure.

2.3.2 Key Exchange Algorithms

Key exchange protocols allow two parties to establish a shared secret over an insecure channel. This shared secret is then often transformed in a key for a symmetric cipher. We can describe a generic key exchange protocol between two entities, Alice and Bob. The protocol makes use of two functions, E and F. Alice starts the protocol by choosing a random secret value α and computing $E(\alpha)$. Next, she sends the result to Bob.

Similarly, Bob picks his secret β and calculates $E(\beta)$. He also sends the result to Alice. Finally, both Alice and Bob derive the shared secret by computing $F(\alpha,\beta)$. For this scheme to be efficient and secure against an eavesdropping adversary, the following properties must hold:

1. E should be efficiently computable.
2. Given α and $E(\beta)$, or β and $E(\alpha)$, it should be simple to compute $F(\alpha,\beta)$.
3. Given only $E(\alpha)$ and $E(\beta)$, it should be hard to compute $F(\alpha,\beta)$.

The functions E and F must be easy to compute to build an efficient protocol. The protection against an eavesdropping adversary comes from the third property. An adversary that monitors the communication channel sees only $E(\alpha)$ and $E(\beta)$. He should not be able to compute the shared secret efficiently from this information.

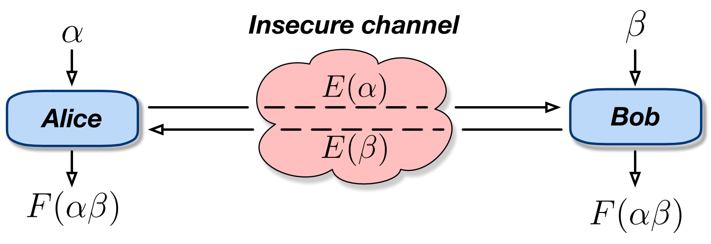


Figure 2.15: Generic key exchange protocol over an insecure channel.

Diffie-Hellman is a well-known key establishment scheme. The protocol is based on the hardness of the DLP problem presented in Section 2.1. The protocol between two parties called Alice and Bob starts with algorithm $\mathcal{G}$ generating a group $\mathbb{G}$ and generator g with order m. These parameters are publicly known and often available ahead of time. Alice chooses a random $x \in \{1,\dots m-1\}$ and computes $h_1 = g^x$. Alice sends h_1 to Bob. Bob chooses a random value $y \in \{1,\dots m-1\}$ and computes $h_2 = g^y$. Bob sends h_2 to Alice. Alice computes $h_2^x = g^{yx}$ and Bob computes $h_1^y = g^{xy}$. Because $g^{xy} = g^{yx}$, both parties now share a secret value $k = g^{xy}$. An eavesdropper, Eve, cannot carry out the same operation as Alice or Bob in order to compute the key k. Eve does not know x or y and cannot compute them. The latter would involve solving the DLP, which is assumed to be hard. However, just as there might be an alternative approach to solving the RSA problem (without relying on integer factorization), there might exist an alternative method to calculate k, which does not depend on solving discrete logarithms. The assumption that no other method exists is known as the computational Diffie-Hellman assumption. It is assumed that finding k from g^x and g^y is hard. Unfortunately, even the computational Diffie-Hellman assumption is not enough to guarantee security in the key exchange protocol. The security of the Diffie-Hellman key exchange needs an even stronger assumption called the *decisional Diffie-Hellman assumption*. This assumption states that g^{xy} is not just hard to compute for eavesdroppers, but also hard to distinguish from a random element in the group. In case, g^{xy} could be easily distinguished, an attacker could recover up to half of the bits of the secret k [31].

2.3.3 Digital Signatures

The last public-key construct we discuss is digital signatures. Functionally, a digital signature is similar to a MAC. They allow a signer who has established a private key k_s to sign a message in such a way that any other party who knows k_p can verify that the message originated from the signer. The signature authenticates and protects the integrity of the message.

Additionally, the signature provides non-repudiation. While everyone with the public key can verify the signature, only the sender could have created the signature, under consideration that the private key was not stolen. Once signed, the signer cannot deny having done so.

Definition 2.3.3. Signature scheme A signature scheme $\mathcal{S}(G,S,V)$ is a triple of efficient algorithms, where G is called a key generation algorithm, S is called a signing algorithm, and V is called a verification algorithm. Algorithm S is used to generate signatures and algorithm V is used to verify signatures.

- G is a probabilistic algorithm that outputs a pair (k_p, k_s), where k_s is called a secret signing key and k_p is called a public verification key.
- S is a probabilistic algorithm that is invoked as $\sigma \xleftarrow{R} E(k_s, m)$, where k_s is a secret key (as output by G) and m is a message. The algorithm outputs a signature σ.
- V is a deterministic algorithm invoked as $V(k_p, m, \sigma)$. It outputs either `accept` or `reject`.

The definition of a secure signature scheme is similar to the definition of a secure MAC. We give the adversary the power to mount a chosen message attack, namely the attacker can request the signature on any message of his choice. The attacker should not be capable of creating an existential forgery. The attack game for a given signature scheme $\mathcal{S}(G, S, V)$, defined over $(\mathcal{M}, \Sigma)$, and a given adversary $\mathcal{A}$, goes as follows:

- The challenger creates a key pair $(k_p, k_s) \xleftarrow{R} G$ and sends k_p to $\mathcal{A}$.
- $\mathcal{A}$ queries the challenger several times. For $i = 1, 2, \ldots$ the i^{th} signing query is a message $m_i \in \mathcal{M}$. Given m_i, the challenger computes $\sigma_i \xleftarrow{R} S(k_s, m_i)$, and then gives σ_i to $\mathcal{A}$.
- $\mathcal{A}$ outputs a candidate forgery $(m_f, \sigma_f) \in \mathcal{M} \times \Sigma$, where

$$m_f \notin \left\{m_1, m_2, \ldots\right\} \tag{2.34}$$

We say that the adversary wins the game if $V(k_p, m_f, \sigma_f) = \texttt{accept}$. We define $\mathcal{A}$'s advantage with respect to S, denoted SIGadv$[\mathcal{A}, S]$, as the probability that $\mathcal{A}$ wins the game.

Definition 2.3.4. We say that a signature scheme S is secure if for all efficient adversaries $\mathcal{A}$, the quantity SIGadv$[\mathcal{A}, S]$ is negligible.

To extend the input domain of the signature scheme, allowing signing of arbitrary long messages, the scheme can be extend with a CRHF. This construct is very similar to the ones presented during our discussion on MAC systems, and is denoted as the *hash-and-sign* paradigm. Let $\mathcal{S}(G, S, V)$ be a signature scheme defined over $(\mathcal{M}, \Sigma)$ and let $H : \mathcal{M}' \to \mathcal{M}$ be a hash function, where the set $\mathcal{M}'$ is much larger than $\mathcal{M}$. Define a new signature scheme $\mathcal{S}'(G, S', V')$ over $(\mathcal{M}', \Sigma)$ as

$$S'(k_s, m) \leftarrow S(k_s, H(m)) \quad \text{and} \quad V'(k_p, m, \sigma) \leftarrow V(k_p, H(m), \sigma) \tag{2.35}$$

Digital Signature Algorithm uses the computational hardness of the DLP to create unforgeable signatures. It is constructed according to the hash-and-sign paradigm. The algorithm uses a cyclic group $\mathbb{G}$ modulo a large prime p. The order q of the generator g is a prime that divides $p-1$. The recommended bit sizes for (p, q) are $(3072, 256)$ [22]. The signer also chooses a private key $x \in \{1 \ldots q-1\}$. The value $y = g^x \bmod p$ constitutes the public key. The parameters (p, q, g) are publicly known. The signing and verification functions are depicted in Algorithm 5.

Algorithm 5 Digital Signature Algorithm.

```
1: procedure DSA SIGNING($x, p, q, g$)
2:     Choose a random value $k \in \{1 \ldots q-1\}$.
3:     Compute $r = (g^k \bmod p) \bmod q$. If $r = 0$, go back to step 2.
4:     Compute $s = k^{-1}(H(m) + x \cdot r) \bmod q$, if $s = 0$, go back to step 2.
5:     return signature $(r, s)$.

1: procedure DSA VERIFICATION($y, p, q, g$)
2:     Verify $0 < r < q$ and $0 < s < q$.
3:     Compute $w = s^{-1} \bmod q$.
4:     Compute $u_1 = H(m) \cdot w \bmod q$
5:     Compute $u_2 = r \cdot w \bmod q$
6:     Compute $v = (g^{u_1} y^{u_2} \bmod p) \bmod q$
7:     return accept if $v = r$ else reject.
```

The hash function, H, is commonly instantiated with SHA256. The security of a DSA signature depends heavily on the value k. This value must be random, secret, and unique. Violating these criteria might lead to a compromise of the private key x. RFC (Request for Comments) 6979 proposed an adapted version of DSA, called the deterministic digital signature algorithm that derives the value k in a deterministic way from the private key x. Deterministic DSA makes it easier to implement the algorithm, since it does no longer require access to a source of high-quality randomness [45].

2.3.4 Elliptic Curve Cryptography

Until now, all the public-key constructs we discussed base their security on hard mathematical problems over large groups of integers. Due to Moore's law and progress in algorithm design, cryptosystems that base their security on these mathematical problems have been forced to steadily grow their key sizes to provide a sufficient level of security [46, 47]. The best-known algorithm to solve the DLP in multiplicative groups over the integers, the general number field sieve, runs approximately in $\exp(O((\log p)^{1/3}))$ for an n-bit prime. In 2016, this made it possible to solve the DLP for a 768-bit prime. The large key sizes make cryptography based on modular arithmetic computationally very expensive. In the context of constrained IoT devices, it even becomes impossible to use these algorithms on certain hardware platforms.

Alternatively to modular arithmetic over the set of integers, we can define operations over an elliptic curve. The best known discrete-log algorithm in an elliptic curve group of size q runs in time $O(\sqrt{q})$. The key size can thus be significantly smaller while still providing sufficient security. Table 2.2 compares the key sizes between the modular arithmetic and elliptic curves for a given security level defined by the length of the symmetric key.

TABLE 2.2: Key size comparison.

Symmetric key size [bits]	Modulus size [bits]	Elliptic curve group size [bits]
80	1024	160
128	3072	256
256	15360	512

In cryptographic applications, elliptic curves are defined over finite fields, a.k.a. Galois fields. We write $E(\mathbb{F}_p)$ to denote that E is defined over $\mathbb{F}_p$.

Definition 2.3.5. Elliptic Curve Let $p > 3$ be a prime and let $a, b \in \mathbb{F}_p$ satisfy $4a^3 + 27b^2 \neq 0$. An elliptic curve E defined over $\mathbb{F}_p$ is given by an equation

$$y^2 = x^3 + ax + b \tag{2.36}$$

A point on the elliptic curve can be written as (x, y) if it satisfies the curve equation and both x and y are in $\mathbb{F}_p$. The curve includes an additional point $\mathcal{O}$ called the point at infinity.

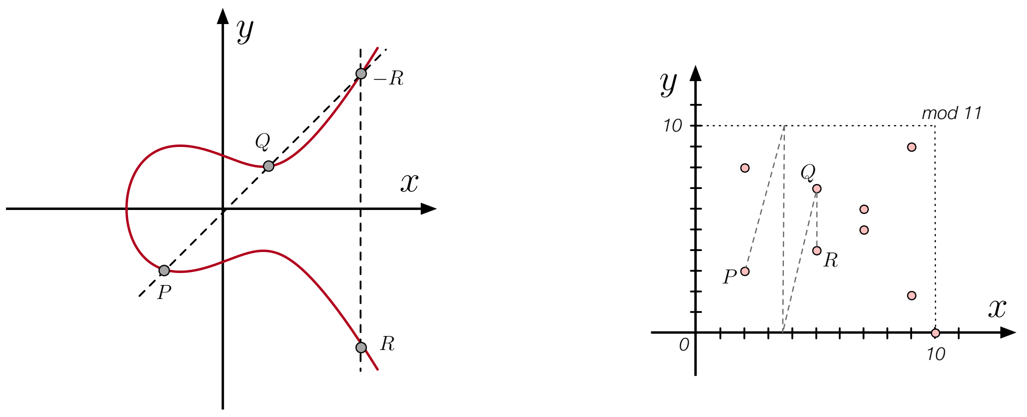
Schoof's algorithm [48] can efficiently calculate the number of points on a curve $E(\mathbb{F}_p)$, even for large prime numbers p. The points on the curve form a cyclic group. The group law, '$\boxplus$'. defined on the points of an elliptic curve, provides a way to perform point addition. The point $\mathcal{O}$ is the identity element of the group. We can thus write $P \boxplus \mathcal{O} = P$. If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ are two points in the curve, to calculate their sum $P \boxplus Q = R$, with $R = (x_3, y_3)$, we use the following three rules:

- If $x_1 \neq x_2$ we use the cord method, see Figure 2.16. Let $s_c = \frac{y_1 - y_2}{x_1 - x_2}$ be the slope of the cord through the points P and Q. The coordinates of the sum can be calculated as

$$x_3 = s_c^2 - x_1 - x_2 \quad \text{and} \quad y_3 = s_c(x_1 - x_3) - y_1. \tag{2.37}$$

- If $P = Q$ and $y_1 = y_2 \neq 0$ we use the tangent method. Let $s_t = \frac{3x_1^2 + a}{2y_1}$ be the slope of the tangent at P. Define

$$x_3 = s_t^2 - 2x_1 \quad \text{and} \quad y_3 = s_t(x_1 - x_3) - y_1. \tag{2.38}$$



(a) Point addition $P \boxplus Q = R$ on the curve $y^2 = x^3 - x + 1$.

(b) All the points on the curve $y^2 = x^3 + 1$ over the field $\mathbb{F}_{11}$.

Figure 2.16: Representations of point additions over elliptic curves.

- Finally, if $x1 = x2$ and $y1 = -y2$ then $P \boxplus Q = \mathcal{O}$.

We can also define scalar multiplication with a point on the curve. We write $2P = P \boxplus P$, $3P = P \boxplus P \boxplus P$, and more generally $\alpha P = (\alpha - 1)P \boxplus P$.

2.3.5 Discrete Logarithms over Elliptic Curves

Now that we have defined how to do computations over an elliptic curve, we can present the DLP over an elliptic curve. Let P be a point on the curve $E(\mathbb{F}_p)$ of prime order q so that $qP = \mathcal{O}$. The DLP in elliptic curve groups is the problem of computing α given the points P and αP as input, for a random $\alpha \in \mathbb{Z}_q$. However, there are some exceptions, and parameters of the curve must be chosen carefully. For this reason, NIST standardized a list of curves for which the parameters are considered safe. One of the most widely used curves is `secp256r1`, a.k.a. `P256`. All implementations of TLS 1.3 are required to support this curve for Diffie-Hellman key exchanges and the ECDSA (Elliptic Curve Digital Signature Algorithm) signature scheme [27]. The curve `P256` is defined as

$$y^2 = x^3 - 3x + b \mod p. \tag{2.39}$$

The prime p is of the form $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. The special structure of p can be used to improve the performance of arithmetic operations modulo p. The value of b is

$$\text{b} = \text{0x5AC635D8AA3A93E7B3ZBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B}. \tag{2.40}$$

Assuming there are no shortcuts, computing discrete logarithm on this curve through brute force takes approximately $\sqrt{q}$ group operations, which is about 2^{128}.

2.4 Cryptography on Constrained Hardware

Cryptographic algorithms have always been perceived as inefficient and time-consuming, particularly on constrained devices. Consequently, many product designers have used this as an excuse to not provide encryption in their products, leading to exposure of sensitive user data. In this section, we perform a comparative study of the cost of state-of-the-art cryptography on an embedded device. We analyze memory consumption and throughput of the algorithms. We study the AEAD cipher suites available in the latest version of TLS and see how they stack up against Ascon, a finalist from the CAESAR competition. Finally, we measure the cost of public-key cryptography.

2.4.1 CAESAR

The CAESAR competition started in 2013 to establish a modern portfolio of AEAD ciphers. There were three general use-cases for the submissions. Authors could submit AEAD ciphers for high-performance applications, AEAD ciphers providing defense-in-depth, and AEAD ciphers optimized of resource-constrained environments (e.g., the IoT). In this work, we are only interested in the use case for constrained environments. The submissions were additionally ranked based on their resilience against SCAs (Side-Channel Attacks). While modern cryptographic algorithms have always concentrated on security against brute-force attacks that recover sensitive information, resilience against SCAs is a more recent desirable property. SCAs analyze leaking information while the cryptographic functions are executing. A powerful SCA is DPA (Differential Power Analysis). While performing a DPA-attack, the adversary can measure minute changes in power output of the device, resulting from manipulating the different bits of the key and plaintext. Following sufficient measurements, the attacker can statistically derive the correct secret key. The wireless, remote nature of IoT devices, while having little physical protection, makes them particularly vulnerable to these types of physical attacks. Therefore, SCA-resistant, efficient, and robust AEAD ciphers are paramount for the security of the IoT. In March 2018, the CAESAR committee announced the final round winners: ACORN [49] and Ascon [50].

2.4.1.1 ACORN & Ascon

ACORN is a bit-based sequential authenticated cipher, based on a stream cipher. It uses a 128-bit key, and the authentication tag length is less than or equal to 128 bits. ACORN uses a 293-bit internal state by concatenating six LFSRs. The cipher processes one bit of the plaintext or additional data per step. Each step, the internal state updates, and the cipher produces a new key bit to XOR with the plaintext. ACORN is very fast in both hardware and software as up to 32 steps can be processed in parallel.

The second finalist was Ascon. Ascon is a sponge-based cipher. Sponge-based constructions are relatively new and also form the basis of Keccak, the winner of NIST's SHA3 competition. Sponge constructions act on a variable-length input and produce an arbitrary fixed-length output. The sponge operates in two phases – *absorbing* and *squeezing*. In the absorbing phase, input bits from the additional data or plaintext are XORed into the internal state of the sponge. In the squeezing phase, each squeeze returns a fixed-length number of bits, which form an output block.

2.4.2 Performance Study

To run the performance tests, we used an STM32F401RE board [17]. The board uses a Cortex-M4F CPU at 84 MHz with 96 KiB of SRAM and 512 KiB of flash. Instruction prefetch, data cache, and instruction cache are enabled to obtain the best performance. The implementation of the cryptographic algorithms is provided by ARM's mbed TLS library [51], except for the Ascon algorithm, whose implementation can be found in the SUPERCOP project [52].

For each primitive, we analyze the runtime and memory pressure. All the code is compiled with the `-Os` optimization flag. Memory usage can be divided into the amount of flash memory used, to store the program code, and peak SRAM usage, which contains the stack, the heap, and the segments `.bss` and `.data`. If available, test vectors provided by NIST's [53] Cryptographic Algorithm Validation Program are used.

2.4.2.1 Symmetric Primitives

While TLS 1.2 still provided a plethora of different symmetric ciphers, many which had been found insecure over the years, TLS 1.3 only supports a small set of secure AEAD ciphers. The remaining five symmetric cipher suites are:

- `TLS_AES_128_GCM_SHA256` and `TLS_AES_256_GCM_SHA384`
- `TLS_AES_128_CCM_8_SHA256` and `TLS_AES_128_CCM_SHA256`
- `TLS_CHACHA20_POLY1305_SHA256`

The first two cipher suites are fast on potent architectures, such as the ones used in PCs and smartphones. The Intel x86 CPU architecture provides dedicated hardware support for both the AES and GHASH part of the GCM AEAD cipher. The ChaCha20+Poly1305 cipher was introduced to provide an alternative to AES-based ciphers and to accommodate for fast encryption on systems that lack AES hardware support. Finally, for reasons discussed above, the CCM variant was introduced for the more resource-constrained devices. The `CCM_8` mode provides shorter authentication tags to minimize the number of bytes on the wire. During the performance tests, each of the TLS cipher encrypted one block of plaintext (16B) with a 12 byte IV and 16 bytes of additional data. The Ascon algorithm also encrypted 16 bytes of plaintext with 16 of additional data but used an IV of 16 bytes.

TABLE 2.3: Encryption performance of the different state-of-the-art AEAD ciphers.

Primitive	**Key size [bits]**	**Time [ms] (× 1e4)**	**SRAM [B]** stack	.bss	.data	**Heap [B]**	**Flash [B]**
AES-GCM	128	1957	2868	2648	12	280	6440
	256	2105	2884	2648	12	280	6456
AES-CCM	128	1960	2448	2648	12	280	5636
ChaCha20+Poly1305	128 + 128	1985	976	0	4	0	2809
Ascon (v1.2)	128	701	648	0	0	0	13308

If we compare the results, we notice that the AES-based cipher all have a similar throughput. The CCM variant consumes slightly less RAM and flash memory compared to GCM. The stream cipher does not perform better than the block ciphers on our test hardware, but it needs significantly less memory (both RAM and flash), compared to the block ciphers.

The new lightweight AEAD cipher, Ascon, is almost three times faster than the traditional ciphers. It also has a strongly reduced memory consumption. It does consume more than twice the amount of flash memory used by the AES-based algorithms. We did not test the ACORN primitive since its SUPERCOP implementation is not optimized for 32-bit CPUs. The results are, therefore, significantly worse compared to the other ciphers.

Some algorithms, such as AES, allow for a speed-up in encryption and decryption if some intermediate results can be pre-computed and stored in memory. ARM's mbed TLS uses the T-table optimization technique, which pre-computes up to 8192 B of data. The tables can be stored in flash memory (to save on RAM usage) or stored in RAM for faster access time. Table 2.4 shows the different performance results for the different levels of pre-computation for the AES algorithm. We must realize that the improvements obtained will differ between different devices. Much depends on the CPU data path and the flash access timings.

TABLE 2.4: Speed-Memory trade-off for AES.

Primitive	**Key size [bits]**	**ROM Tables**	**Less RAM Tables**	**Time [ms] (× 1e4)**	**SRAM [B]** stack	.bss	.data	**Heap [B]**	**Flash [B]**
AES-ECB	128			128	2596	8748	4	0	2700
	128	✓		168	864	0	4	0	10880
	128		✓	115	2572	2604	4	0	2604
	128	✓	✓	157	864	0	4	0	4752

2.4.2.2 Asymmetric Primitives

We test two asymmetric primitives: ECDH (Elliptic Curve Diffie-Hellman) and ECDSA. Both primitives are necessary during an authenticated key establishment. The former because it calculates the shared secret by multiplying the peer's public key with its private key. The latter is widely used to provide authentication during the key exchange. In Chapter 7, we provide an in-depth study of the TLS and DTLS (Datagram

Transport Layer Security) handshake. The performance of both primitives plays a significant role in the overall latency of the handshake.

Table 2.5 shows the performance results obtained for the ECDH primitive. In the test, we parse the public key from the peer and derive the shared secret by multiplying it with our private key. We tested two curves. The first curve is the widely deployed NIST curve, P-256. The second curve, Curve25519, was defined by Daniel J. Bernstein [54]. The curve provides the same security level as P-256 (128 bits) but achieves a better performance. Besides, Curve22519 lends itself to implementation in constant-time, making it resistant to a wide range of side-channel attacks [55].

Recently, Curve25519 gained much interest as an alternative for the NIST curves. The discovery of the NSA (National Security Agency) backdoor in the `Dual_EC_DRBG` algorithm has cast severe doubts on the security of the NIST curves [56, 57]. There is no explanation for the seeds chosen to generate the curves, and some cryptographers suspect there might be an NSA backdoor hidden in the parameter choices. In contrast, the process used to pick Curve25519 is fully documented and rigid enough so that independent verification can and has been done [58].

TABLE 2.5: ECDH performance (ECP window = 6).

Primitive	**Curve**	**Time [ms]**	**SRAM [B]**			**Heap [B]**	**Flash [B]**
			stack	.bss	.data		
ECDH	SECP256R1	437	1856	16	52	4308	14336
	Curve25519	375	1824	16	52	1412	16844

Table 2.6 and Table 2.7 depict the performance results for distinct digital signature algorithms. All the algorithms calculated a signature on the same short string. For ECDSA, we tested two NIST curves. To put their performance results in perspective, we can compare them to RSA-based signatures. The P-256 curve provides the same level of security as a 2048 bit RSA key. The signature speed is roughly three times faster for ECDSA, while heap and stack consumption is approximately two times lower.

Interestingly, RSA signature verification is extremely fast. There is a high asymmetry in cost associated with RSA signing and RSA verification. The elliptic curve digital signature schemes also show an inequality in speed and memory usage between signing and verification. However, for the ECDSA primitive, signature verification is approximately 25% more expensive than the signing operation.

TABLE 2.6: ECDSA performance (ECP window = 6).

Curve	**Curve**	**MD**	**Operation**		**Time [ms]**	**SRAM [B]**			**Heap [B]**	**Flash [B]**
			sign	verify		stack	.bss	.data		
ECDSA	SECP256R1	SHA-256	✓		459	1592	44	16	4072	18144
	SECP256R1	SHA-256		✓	608	1368	52	16	4512	14960
	SECP384R1	SHA-256	✓		748	1696	44	16	10012	18672
	SECP384R1	SHA-256		✓	937	1440	52	16	10212	15592

TABLE 2.7: RSA signature performance.

Primitive	**Key size** [bits]	**Padding Mode**	**Operation**		**Time [ms]**	**SRAM [B]**			**Heap [B]**	**Flash [B]**
			sign	verify		stack	.bss	.data		
RSA	2048	PKCSv1.5	✓		1447	3462	52	12	10032	19968
	2048	PKCSv1.5		✓	30	3056	52	12	4936	17280

Discussion

In the past decades, cryptography has become an indispensable tool for the construction of secure communication protocols. More than ever, we rely on cryptography to protect our data and systems from unauthorized access. With the emergence of the constrained IoT, the necessity to build fast and energy-efficient algorithms has only increased. Cryptographic algorithms are split into two categories: symmetric primitives and asymmetric primitives.

The class of symmetric algorithms is the most intuitive. Communication parties share a secret key and use it for both encryption and decryption. Besides confidentiality, symmetric cryptography also aims to protect the integrity of the messages. Integrity protection ensures that malicious modification is detected, and the receiving party does not undertake any actions based on falsified information. In general, the constrained IoT can execute symmetric primitives without much difficulty. However, in scenarios with stringent timing constraints, e.g., in time-slotted network protocols, or when energy is scarce, faster and less laborious algorithms are required. For this reason, competitions on constrained symmetric cryptography are underway. Only last year, the CAESAR competition concluded, and currently, a NIST competition is in its second stage.

The category of asymmetric algorithms, a.k.a. public-key cryptography presents solutions for several problems that symmetric constructions cannot solve. It provides public-key encryption, digital signatures, and key-exchange mechanisms. The asymmetric key, consists of a pair, denoted as the public key and the private key. While anyone can obtain the public key, the private key is kept secret. Asymmetric algorithms are more challenging to execute for the constrained IoT. They require much energy and often several seconds to complete. Most constrained devices, therefore, focus on the elliptic curve variant. Elliptic curve algorithms are capable of providing the same security level with much smaller key sizes, making them more suitable for low-power devices.

There is a good chance that in the near future, many of the algorithms we currently use to protect our data will be broken. In recent years, there has been a substantial amount of research on quantum computers. These machines exploit quantum mechanical phenomena to solve mathematical problems that are difficult or intractable for conventional computers [59]. In 1994, Peter Shor published an algorithm that would break the public-key algorithms based on the factorization assumption and the DLP if a powerful quantum computer existed [60]. Since it is believed that a large-scale quantum computer could be built in the upcoming decades, NIST has issued a cryptographic competition for post-quantum cryptography. Just as the traditional Internet, the IoT will have to adapt and deploy post-quantum cryptography. Already researchers are studying the use of post-quantum, hash-based signatures, and lattice-based primitives on constrained devices [61, 62].

Instead of merely using the cryptographic algorithms as a black box throughout this thesis, we presented an exhaustive introduction to modern cryptography in this chapter. We explained the reasoning behind the design of the constructions and their pitfalls. We started our discussion with the presentation of the assumptions upon which rest the security of many cryptographic primitives. We defined what it means for a cipher to be secure. Next, we presented the different symmetric primitives. We highlighted the differences between stream ciphers and block ciphers and showed different methods to construct MAC systems. Finally, we combined encryption and MAC algorithms in the AEAD construct. AEAD ciphers are the only symmetric construction that are secure in the strongest attacker model, i.e., CCA.

In the second part we switched to public-key cryptography. We considered, public-key encryption, key exchange methods and signature schemes. We detailed the mathematical problems that lay at the core of these constructions. Finally, we concluded this chapter with some performance tests of the discussed algorithms on constrained hardware. We compared the well-known AEAD ciphers, used in TLS 1.3 with Ascon, one of the winners of the CAESAR cryptographic competition. Concerning public-key cryptography, we tested the performance of different elliptic curve algorithms and showed their performance gain over RSA.

Chapter 3

A Secure IoT Networking Stack

Contents

Introduction

Traditionally, the complexity of computer networking has been dealt with through abstraction. A network stack consists of layers. Each layer uses the services of the underlying layer and provides an interface to the layer above. On the left side of Figure 3.1, we depict the communication and security protocols that typically make up the network stack used on the Internet (web), and the right side shows the ones considered in this thesis. To understand the differences between both stacks it is essential to keep in mind the hardware characteristics of the devices for which they were developed.

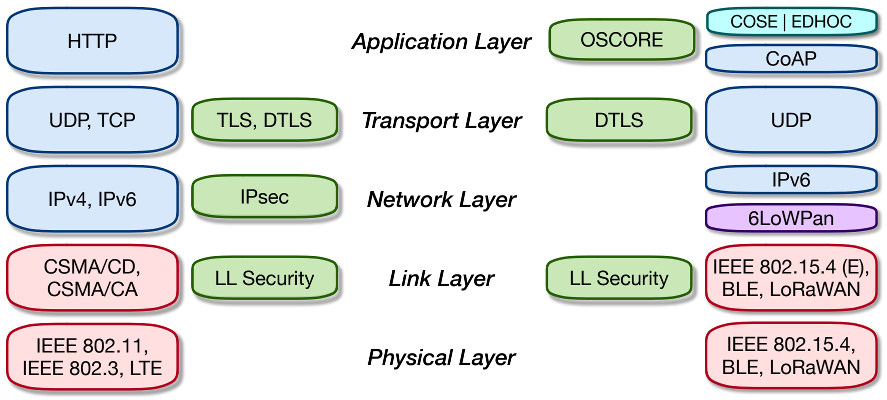


Figure 3.1: Comparison of the TCP/IP stack with the standardized IoT stack.

As discussed in Chapter 1 IoT, devices are often marked by significant limitations on energy and computational resources. These constraints have a direct impact on the operation of the lower layers and the security mechanism employed throughout the stack. Small packet sizes, radio duty cycling, and low transmission power characterize the physical and link layer of the IoT stack. Security protocols should be conservative with the use of computationally complex cryptography to prevent battery depletion. Finally, the properties of the traditional Internet protocols are often incompatible with the demands of the IoT applications. Applications requiring support for multicast messaging and caching cannot simply use the prevalent TCP (Transport Layer Protocol) and TLS (Transport Layer Security) protocols.

In response, several IETF working groups are currently standardizing new or adapted protocols that address the needs of the IoT. The protocols must be kept simple to reduce the code footprint and RAM usage, but should be interoperable with the existing Internet technologies. CoAP (Constrained Application Protocol) has been specifically designed for easy integration with HTTP (Hypertext Transfer Protocol), allowing traditional Internet hosts to access the IoT directly. Security protocols can benefit from the rising popularity of elliptic curve cryptography and modern AEAD encryption schemes. The COSE (CBOR Object Signing and Encryption) compact message format allows for efficient encoding of the data with built-in support for encryption, integrity protection, and signatures. OSCORE (Object Security for Constrained RESTful Environments) and EDHOC (Ephemeral Diffie-Hellman over COSE) aim to complement or replace TLS and DTLS (Datagram Transport Layer Security) in the IoT space. They both leverage COSE to optimize the message length. The push towards energy-efficiency in IoT protocols has also blurred the boundaries of the different layers in the network stack. The 6LoWPAN (IPv6 Low-power Wireless Personal Area Networks) specification provides an adaptation layer between the network layer and the link layer. It describes how to transmit IPv6 packets over IEEE 802.15.4 networks [63]. This is not a straightforward process. For instance, due to the IPv6 default minimum MTU (Maximum Transmission Unit) size of 1280 bytes, an unfragmented IPv6 packet would be too large to fit in an IEEE 802.15.4 frame. It shows the need for a fragmentation mechanism between the IP (Internet Protocol) and the link layer. Additionally, the IPv6 header is 40 bytes long. Without compression techniques, the header would waste the scarce bandwidth [64].

In this chapter, we give an overview of the different protocols of the standardized IoT stack. The first part of this chapter is devoted to a discussion on authentication and access control in the IoT. We describe how OAuth (Open Authorization) tackled a similar issue on the world wide web, and we present the adapted version for the IoT, currently under development by the IETF. Next, we provide an overview of the standardized IoT networking stack. We use a top-down approach to present the trade-offs between different protocol choices, their security features, and we discuss the associated network security protocols.

3.1 An Internet Threat Model

Security analyses of systems commonly begin with a model of the attacker. Dolev and Yao [65] formulated the standard attack model against messages exchanged over a network. The Dolev-Yao model makes the following assumptions about an attacker:

- **Eavesdrop:** An adversary can listen to any message exchanged through the network.
- **Forge:** An adversary can create and inject entirely new messages into the data-stream or change messages in flight; these messages are called forgeries.
- **Replay:** A special type of forgery, called a replay, is distinguished. To replay a message, the adversary resends legitimate messages that were sent earlier.
- **Delay and Rush:** An adversary can delay the delivery of some messages or accelerate the delivery of others.
- **Reorder:** An adversary can alter the order in which messages are delivered.
- **Delete:** An adversary can destroy in-transit messages, either selectively or all the messages in a data-stream.

The Dolev-Yao model is considered to be a powerful model, giving much power to the attacker. However, there are some aspects to take into account. The cryptographic operations used in the networking security protocols are considered unbreakable. Besides, the model does not handle the compromise of the individual nodes in the network. These types of attacks require a different model, presented and discussed in Chapter 4.

3.2 Identity Management and Access Control

Many IoT devices operate without any human interaction. Due to the vast quantities of devices, manual management would be a very time-consuming and tedious task. Nonetheless, during the lifetime of these devices, they will often have to make non-trivial authorization and authentication decisions, most notably, which parties are allowed to access the resources generated by the devices. This is the problem that authorization and authentication architectures try to solve.

3.2.1 Token-based Access Control

The authentication and authorization challenges that the IoT is currently facing have been encountered before in other domains, i.e., the world wide web. For the world wide web, the issue can be formulated as follows: how can a third-party web application dynamically request (scoped) access to sensitive user information stored by another web service. For example, a webpage personalizes its content for each user. To do so, it wants to obtain the full name, age, and gender of a user by retrieving this information from the user's Facebook profile. A naive approach would be to provide the web application with the user's password to access the information stored by Facebook. One can directly notice that this would provide the web application with unrestricted access to all the user's information. Additionally, it would be difficult for the user to revoke the web application's access rights; the user would have to change its password. The OAuth framework solves this problem.

3.2.1.1 The Open Authorization Framework 2.0

In the context of OAuth 2.0 [66], the third party is known as the *client*, the user is the *resource owner*, and the web service storing the user information is the *resource server*. The user information is called the *protected resource*. To delegate scoped access rights to different clients, OAuth uses tokens. A trusted party, called the *authorization server*[1], creates the tokens. The issued access tokens encode the scope of the client, e.g., the protected resources that can be accessed, and a time duration specifying the token's period of validity. The

[1]In practice, the authorization server and resource server are often managed by the same entity.

authorization server can use JWT (JSON Web Tokens) [67] to encode claims in a JSON (JavaScript Object Notation) object that is then signed. The tokens are transported inside a URL (Uniform Resource Locator). To protect the confidentiality of interactions between the different entities and to prevent token theft, all the communication must use HTTPS (Hypertext Transfer Protocol Secure).

OAuth is a flexible framework that adapts to several usage scenarios. Depending on the client type (e.g., web server app, native app or browser-based script) and the level of trust between the resource owner and the client, different authorization mechanisms and protocol flows are available: *authorization code flow*, *implicit flow*, *resource owner password credentials flow*, and *client credentials flow*. Each flow uses a specific resource owner grant type that it issues to the client. The general principles of each flow stay the same:

1. A client obtains an authorization grant, after being authorized by the resource owner.
2. The client exchanges the authorization grant for an access token at the authorization server.
3. The client uses the token to access the protected resources on the resource server.

In Figure 3.2, we depict the authorization code flow. This authorization flow is widely used on the web to provide web applications with access to protected resources, e.g., Google or Facebook user information. Before a client can interact with an authorization server, the client registers at the authorization server. The details of the registration process are out of the scope of the OAuth protocol. In practice, this process is usually a manual task. During the registration process, the authorization server assigns credentials to the client: a public OAuth client ID and (optionally) a client secret. The client may later use the client secret (if issued) to authenticate to the authorization server while exchanging a grant for a token [68].

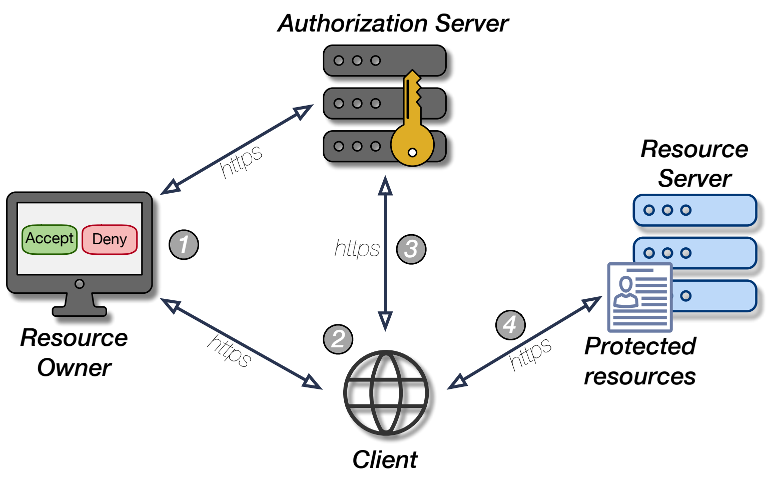


Figure 3.2: OAuth authorization grant code flow for a web server application.

1. **Client Authorization:** To start the flow, a resource owner visits the client (i.e., web application). When a client wants to access protected resources on behalf of the resource owner it asks the resource owner for authorization. The client redirects the owner to the authorization server where the owner needs to authenticate. This authentication step verifies if the resource owner is allowed to grant access to the requested protected resources.
2. **Obtaining an Authorization Grant:** After a successful authentication, the resource owner can inspect and possibly limit the scope and duration of the client request. Once validated the owner is redirected to the client. The redirection URL carries the authorization grant issued by the authorization server for the client.
3. **Retrieving the Access Token:** The client can now recontact the authorization server and recover an access token in exchange for the authorization grant.
4. **Accessing Protected Resources:** With the token in hand, the client can access the protected resources, stored on the resource server.

OAuth security is extensively described in RFC 6819 [69]. As a flexible and extensible framework, the OAuth security depends on many factors. The most important takeaway is that OAuth is a mechanism for authorization delegation. The protocol does not support authentication, although clients have misused it as a way to authenticate resource owners to their application. In this scenario, the client sees the authorization server and resource server as an identity API (Application Programming Interface). The client requests authorization from the resource owner to access a user-specific identifier on the resource server. If the client succeeds in obtaining an access token for this identifier, the resource owner is considered to be logged in successfully to the client's application [68]. Because OAuth was not designed with authentication in mind, the approach is highly dangerous. To fill the void of authentication through external parties (a.k.a. identity providers), OpenID Connect was created. It is an open standard published in early 2014 that defines a thin interoperable layer on top of OAuth to perform user authentication.

Because of the high complexity of the system and the many interactions between independent components, access token theft remains a threat. As a solution, the IETF worked on a draft for a PoP (Proof-of-Possession) architecture. We discuss the PoP principle further in detail in the next section. Alternatively, a new extension for the TLS protocol was proposed called, *the token binding protocol* [70]. Token binding allows the authorization server to bind the exchanged tokens to the underlying TLS protocol cryptographically, preventing attackers from replaying stolen tokens over new connections.

3.2.1.2 Authorization and Authentication in Constrained Environments

ACE (Authentication and Authorization for Constrained Environments) adapts and builds on the OAuth framework to provide authentication and authorization for the IoT of constrained devices. Instead of (web) applications requesting access to user data stored on powerful servers, ACE sketches an architecture where both the client and the resource server are possibly constrained [71]. It uses by default CoAP [72] and CBOR (Concise Binary Object Representation) [73] instead of HTTP and JSON for communication.

Similar to OAuth, ACE defines different authorization flows for specific scenarios. Each flow has a corresponding grant type, but there are two preferred types, namely the authorization code grant and the client credentials grant. The authorization code grant is a good fit for use with apps running on smartphones and tablets that request access to IoT devices. This can be a common scenario in the smart home environment, where users and applications need to go through an authentication and authorization phase (at least during the initial setup phase). ACE uses the client credential grant when the communication is purely M2M (Machine-to-Machine). It is a simplified authorization flow designed for use cases where the client itself is constrained. In this case, the resource owner has pre-arranged access rights for the client with the authorization server. The client has a set of client credentials, which it can use to obtain an access token from the authorization server directly.

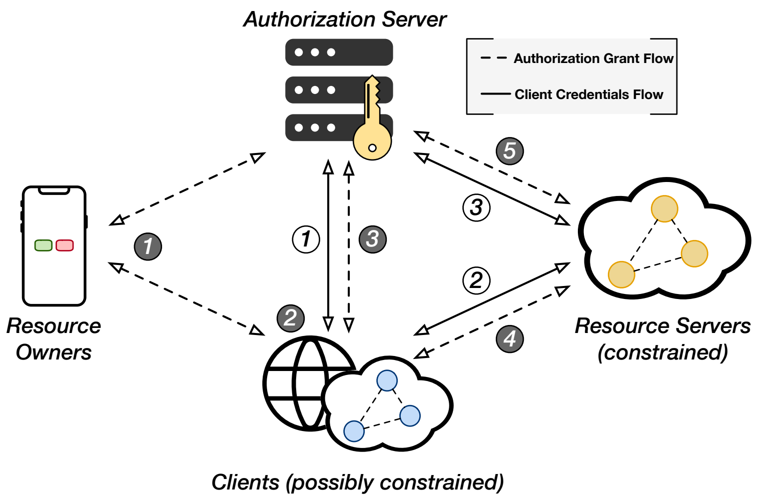


Figure 3.3: ACE authorization flows.

Figure 3.3 shows the authorization code flow and the client credential flow. The authorization code flow for ACE corresponds almost entirely to its OAuth sibling. ❶ A user (resource owner) opens an application (client). The application wants to access some protected resources, on behalf of the user, hosted on the constrained resource servers. The application redirects the resource user to the authorization server, where the resource owner authenticates and checks the scope of the client's request. ❷ After validation, the client receives an authorization grant. ❸ The client can then exchange the grant for an access token. ❹ Finally, the token is presented to the resource server to obtain the protected resources. During a client credential flow, ①, the client directly requests a token at the authorization server. If the requested scope matches the predefined access rules, an access token is issued. ② The client can then present the token to the resource server.

Because the resource servers can be highly constrained, the ACE framework provides an extra optional step, called *token introspection*. In this step, the resource server will delegate the access token validation to the authorization server (see ❺ and ③). Access token validation can require the use of asymmetric cryptography, e.g., signature validation, which could pose a high strain on the device's limited resources if it needs to serve many different clients.

ACE security defines a set of profiles that present the encodings and protocols the client, resource server, and authorization server must support to exchange data securely. Examples are the OSCORE [74] or DTLS [75] profiles. The profiles define methods to provide encryption, integrity, and replay protection of the data. Additionally, the profiles must declare how the entities can mutually authenticate each other. The authorization server is responsible for provisioning the keying material so that client and resource servers can authenticate each other.

ACE implements, by default, PoP access tokens. The authorization server generates a PoP token when it binds a client's cryptographic key to the access token. When used to access protected resources on the resource server, the client must prove possession of the secret that is bound to the PoP. Only the legitimate client of the token can prove possession of the secret.

3.2.2 Blockchain and the IoT

3.2.2.1 Blockchain Internals

In recent years, the blockchain technology has made a lasting impact on the research community and industry. It gained widespread fame as the underlying technology for cryptocurrencies, but it offers a wide variety of opportunities in multiple research domains. The blockchain technology distinguishes itself from its competitors due to the Byzantine threat model in which it operates. Distributed systems that function in a Byzantine threat model keep on functioning in the presence of compromised participants. The BFT (Byzantine Fault Tolerance) describes the maximum amount of malicious actors in the system that work towards the same goal until the system fails.

In general, a blockchain can be considered as a persistent log whose records are stored in time-stamped blocks. Each block contains transactions between parties. A cryptographic hash identifies a block, and it references the hash of the preceding block. Anything that is stored in the blockchain is public. The blockchain is maintained by nodes, each having a copy of the entire chain. The initial block is often called the *genesis block*. To provide security in the network and consistency among all the different copies of the blockchain, the participants run a consensus protocol. Not all the participants in the blockchain network need to run the consensus protocol. The nodes that do are called *miners*. There exist various types of consensus protocols: the most famous are called PoW (Proof-of-Work) and PoS (Proof-of-Stake).

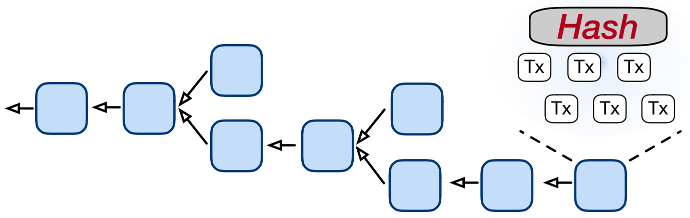


Figure 3.4: The blockchain, the consensus protocol provides security and resolves forks in the chain.

The participants in a blockchain network connect in a P2P (Peer-to-Peer) fashion. When transactions occur between participants, the transaction information propagates throughout the network. Each miner constructs its block, which it fills with transactions it overhears. In a PoW consensus protocol, the miners try to solve a cryptographic puzzle. The average time to solve the puzzle is known as the block time. It is a network-dependent parameter. For example, in the Bitcoin [76] network, the block time is 10 min, while in the Ethereum [77] network, it is only 15 sec. When one of the miners succeeds in finding a solution for the puzzle, it broadcasts its solution to all the other participants. If they can successfully validate the solution, everybody adds the block to their local copy of the chain. Multiple miners can find a solution at the same time. In this case, the blockchain will fork. Some parts of the network will continue working with the chain with solution A, while the other parts continue with solution B. Since the miners are configured to always work on the longest chain, once one of the branches overtakes the other, all the miners will switch to the longest branch.

The PoW consensus protocol suffers from several drawbacks. First, the protocol consumes large amounts of energy, e.g., the Bitcoin network currently uses more energy than a small nation. Secondly, because of the difficulty of solving the cryptographic puzzle, miners are organized in *mining pools*. These pools lead to a more centralized system, undermining the idea behind the blockchain technology and its security. A promising more resource-efficient alternative is PoS. In the PoS protocol, the cryptographic puzzle is replaced with a voting system that selects for each block a new block *minter*. The consensus protocol selects a minter based on the amount of wealth the minter has invested in the consensus protocol, i.e., the *stake*. Distinct implementations of the PoS protocol provide different punishments to discourage malicious behavior of the minter. In general, minters lose their stake if they maliciously manipulate the minted blocks.

3.2.2.2 Managing the IoT through Blockchain

Blockchains are automatically associated with cryptocurrencies, but with the rise of the Ethereum network, more applications became available. Ethereum provides the ability to execute smart contracts. Smart contracts are like regular computer programs, but they execute on in the EVM (Ethereum Virtual Machine). All Ethereum nodes execute the smart contract in their local instantiation of the EVM. The next appended block then solidifies the blockchain state changes triggered by the smart contract. The consensus protocol ensures that everyone agrees on the result of the contract execution.

In the context of IoT, programmable blockchains like Ethereum could provide an elegant solution to long-standing problems concerning identity management and resource transactions. A recurring challenge in the IoT is to track and verify the ownership and identity of IoT devices. The ownership of an IoT device can change several times during the lifetime of a device from the manufacturer, supplier, retailer, and consumer. The ownership must change or be revoked when an IoT device gets resold, decommissioned, or compromised. By creating a smart contract that handles the device ownership, the whole process could be done transparently and securely. The literature also contains proposals where the blockchain technology is used for logging and manage trading of resources generated by IoT devices [78–80].

3.3 The IoT Application Layer and End-to-End Security

3.3.1 The Constrained Application Protocol

The application layer is the top layer of the networking stack. It exposes an API to developers to easily exchange data between applications over a network. Application layer protocols are concerned with transferring information needed by the actual application, e.g., web pages or temperature readings from a sensor. CoAP [72] is a constrained application layer protocol designed by the IETF CORE (Constrained RESTful Environments) working group. CoAP messages can easily be mapped to HTTP for integration with the traditional web. CoAP uses a RESTful API in a typical client-server architecture.

CoAP is designed with the restrictions over the lower layers in mind. Although there are strong similarities with the HTTP protocol, significant changes were made to facilitate its use in highly constrained environments. CoAP can be used in combination with UDP (User Datagram Protocol), and optionally provides application layer reliable unicast and best-effort multicast. The protocol has a low parsing overhead and it supports asynchronous message exchanges. CoAP provides simple proxy and caching capabilities through a “freshness” mechanism. Caching of application data can reduce the overall traffic in constrained networks. The latter is exceptionally beneficial for the battery lifetime of the low-power devices and the overall latency in the network. Resource discovery is an optional feature of the protocol. The CoAP message format is divided into two layers, a message layer in charge of reliability, and sequencing and a request/response layer in charge of mapping requests to responses. Figure 3.5 shows the different message fields for a CoAP message.

The message layer uses a 16-bit message identifier to provide reliability and detect duplicates. A 2-bit type field indicates the type of the message: confirmable, non-confirmable, acknowledgment, or reset. The first two types indicate the need for replies, whether or not piggybacked on an acknowledgment message. If a confirmable message cannot be processed, the receiver must answer with a reset message.

The request-response layer sets the method code or response code in the 8-bit code field. The primary method codes supported by CoAP are `GET`, `POST`, `PUT`, and `DELETE`. Analogous to the HTTP response codes, CoAP uses the specific conventions. Codes 2.xx represents successfully received and processed. Client error codes use the 4.xx format and internal server errors return 5.xx. Optional (or default) request and response information, such as the URI (Uniform Resource Identifier) and payload content-type, are carried as CoAP options. There are two types of CoAP options: critical and elective. The CoAP endpoint can ignore elective options if they are not understood, while critical options must be answered with a 4.02 (Bad Option) response if they cannot be processed correctly. A variable-length token field is used to correlate responses and requests independently from the underlying messages [64].

3.3.2 OSCORE: Object Security for Constrained RESTful Environments

The defacto way to provide confidentiality, integrity, and authenticity for application data in transit is to build secure channels on the transport layer of the network stack, see Section 3.4. This approach to security is connection-focused and works well for typical Internet applications such as e-commerce, e-banking, or IP telephony [81, 82]. In these examples, we can identify the endpoints and the connections between them. Difficulties emerge once the notion of a connection disappears. This often occurs in systems and protocols with intermittent connectivity between the communicating parties, e.g., data is served from cache. For example, DNS (Domain Name System) provides data integrity with the application-level extension DNSSEC (Domain Name System Security Extensions), and not with a connection-oriented protocol, such as DTLS or TLS[2] [86, 87]. Electronic mail, passing multiple application level gateways, and without a clear connection between endpoints, is secured with S/MIME (Secure/Multipurpose Internet Mail Extensions) [88] or PGP (Pretty Good Privacy) [89]. IoT applications emerge as another example. The IoT application traffic is typically asynchronous, many of duty-cycled networks will employ caches at network entry points to alleviate pressure on the constrained devices, and group communication between the constrained devices will happen frequently. Caches and proxies would require the termination of the DTLS and the TLS connection. Therefore, these middleboxes not only have access to the data required for performing their intended functionality but are also able to eavesdrop on or manipulate any part of the message payload and metadata in transit between the endpoints [7]. To provide an end-to-end security model that better fits the needs of the IoT, the CORE working group defined a new security protocol that operates at the application layer, OSCORE [7], based on the work by Vučinić et al. [82]. OSCORE protects the CoAP requests/response layer end-to-end across intermediary nodes.

OSCORE endpoints use a pre-established shared security context that contains information on the master secret, the master salt, the context ID, the IV, and the cryptographic algorithms to use (for key derivation from the master secret and authenticated encryption). Each endpoint also has a sender context and receiver context. The first contains a sender ID, sender key and sequence number, while the latter has a recipient ID, recipient key and replay window. CoAP messages use a critical option to indicate that OSCORE is used to protect the contents of the packet. Not all CoAP fields can be encrypted to ensure the proper working of the CoAP caches and proxies. OSCORE divides the CoAP fields in classes. Class E fields are encrypted and integrity protected, fields of class I are only integrity protected, and class U consists of unprotected fields. Similarly, the different CoAP options are split in Class E, I, and U options. OSCORE protects the CoAP class

[2] As of 2019, several companies, i.e. Google and Cloudflare among others, have started serving DoS (DNS-over-TLS) to protect user privacy. There are also experiments with DoH (DNS-over-HTTPS) [83–85].

E and *I* fields and options by wrapping them in a COSE object. In Figure 3.5, the protected CoAP fields are colored green, the unprotected red.

COSE [90] is a binary message format for a concise representation of small messages. COSE messages can be encrypted, MAC'ed, and signed. The basic format defines three fields: protected header, unprotected header, and content (see Figure 3.5). The headers are flexible and can carry different types of information. For example, they can carry information about the content protection mechanism, such as the algorithm identifier, key identifier, IVs, and other contextual information. The COSE content itself is either the plaintext or the ciphertext. Depending on the COSE object type, an additional field might be added. There are 6 COSE message types: `COSE_Encrypt0`, `COSE_Encrypt`, `COSE_Mac0`, `COSE_Mac`, `COSE_Sign1` and `COSE_Sign`. The type names followed by a `0` or `1` indicate that the COSE object is either encrypted for a single recipient, MAC'ed without any additional recipient information, or signed by a single signer. In those scenarios, the additional field contains, either a single signature (for `COSE_Sign1`) or a single MAC (for `COSE_Mac0`). Note that the ciphertext from the `COSE_Encrypt0` message is carried in the content field; thus, the additional field is empty. The other message types are used when multiple recipients are addressed, or multiple signatures are attached. The additional field now contains a recipient structure or signatures structure. Both structures follow the same message format: headers, protected and unprotected, followed by either a recipient-specific key wrap of the used encryption or MAC key or a signature for each signer. The recursive message structure allows for a small code footprint on constrained devices.

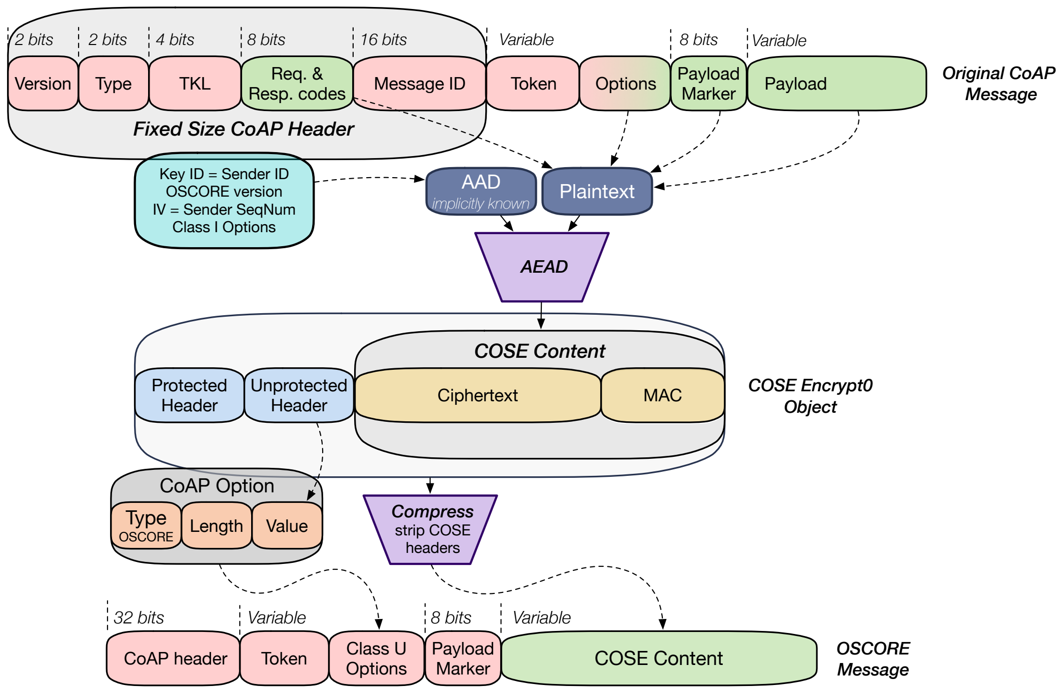


Figure 3.5: OSCORE transformation

OSCORE uses the `COSE_Encrypt0` message type to protect the *E*, and *I* class CoAP fields and options, see Figure 3.5. It uses an AEAD algorithm for encryption and integrity protection. By default, it uses AES-CCM. The plaintext input of the AEAD algorithm consists of a concatenation of the CoAP response code, the class *E* options [7], and the CoAP payload (if there is a payload present in the original CoAP message), prefixed with the payload marker. The AAD (Additional Authenticated Data) input to the AEAD cipher, consists of the OSCORE version, the AEAD algorithm identifier, KID (Key Identifier), the IV, and the CoAP class *I* options. The COSE protected header is empty and the unprotected header contains the IV, the KID parameter, and optionally a KID context. The final OSCORE message now consists of the original CoAP header, a CoAP token (if any), and the unprotected options (there is at least the OSCORE option). The request/response code in the CoAP header of the OSCORE message has been replaced with either a CoAP `POST` or `CHANGED` code since the original value has been encrypted and stored inside the COSE object. The information in the COSE unprotected header will be carried in the value field of the OSCORE option, see Figure 3.5.

3.3.3 EDHOC: Ephemeral Diffie-Hellman over COSE

During the discussion on OSCORE, we stated that the protocol uses a shared security context, but we did not explain how OSCORE established this context. At the time of writing, the ACE working group is designing the EDHOC lightweight key exchange mechanism [91]. The protocol will provide perfect forward secrecy, through ephemeral Diffie-Hellman with elliptic curve keys, and identity protection. Authentication will happen through pre-shared keys (symmetric, raw public keys, or certificates) established out-of-band. They can be provided by a trusted third party like the authorization server in the ACE framework. EDHOC uses COSE for cryptography, CBOR for encoding, and CoAP for transport, leveraging existing libraries and limiting the additional code footprint.

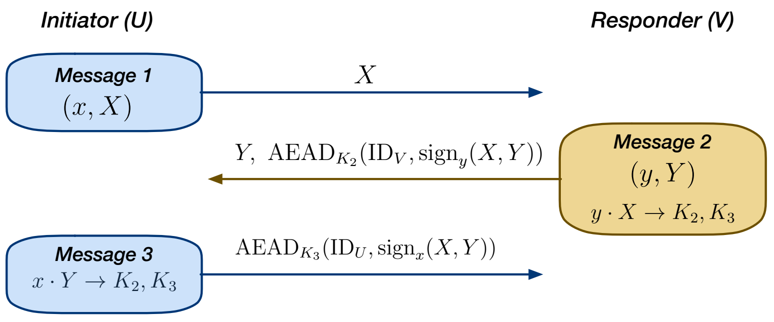

Initiator (U)
Responder (V)
Message 1
(x, X)
X
$Y,\ \mathrm{AEAD}_{K_2}(\mathrm{ID}_V, \mathrm{sign}_y(X,Y))$
Message 2
(y, Y)
$y \cdot X \rightarrow K_2, K_3$
Message 3
$x \cdot Y \rightarrow K_2, K_3$
$\mathrm{AEAD}_{K_3}(\mathrm{ID}_U, \mathrm{sign}_x(X,Y))$

Figure 3.6: The SIGMA-I protocol forms the template for EDHOC key exchange.

EDHOC, just like TLS 1.3, is a SIGMA-I type protocol [92], allowing each party to check the other's identity without revealing it to a passive attacker [93]. The protocol uses three message to establish a shared secret. The initiator (U) starts the protocol by deriving an ephemeral key pair (x, X) and sending the ephemeral public key, X, to the responder (V). The responder generates its ephemeral pair (y, Y) and derives two symmetric keys K_2 and K_3 from the Diffie-Hellman shared secret, $y \cdot X$. The key K_2 is used to encrypt the public identity credential ID_V together with a signature over the two ephemeral public keys (X, Y) responder (V). Upon receiving the second message, the initiator can also derive K_2 and K_3, from $x \cdot Y$ and check the identity of the responder. It then forms the third message containing the encrypted signature of (X, Y) and its identity credential ID_U. The signatures over the ephemeral public keys, ensure that both parties agree on the keys if they are freshly generated.

EDHOC uses the SIGMA-I protocol as a template and adds a few parameters. EDHOC adds connection identifiers C_U and C_V and, cryptographic suites to negotiate an elliptic curve, key derivation algorithm and AEAD cipher. Optionally the exchange messages can also carry unprotected application data. Protected application data can be transferred, starting from message 3. EDHOC makes use of `COSE_Key` objects, two different COSE message types, `COSE_Encrypt0`, `COSE_Sign1`, and COSE context information objects `COSE_KDF_Context`. The parameters $\mathrm{ID}_{\mathrm{cred}_U}$ and $\mathrm{ID}_{\mathrm{cred}_V}$, allow to both parties to obtain cred_V and cred_U. cred_V and cred_U are then used to verify the other parties identity.

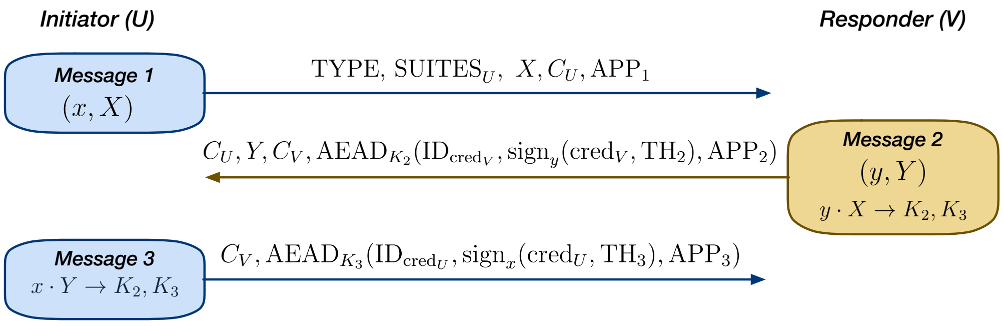

Initiator (U)
Responder (V)
Message 1
(x, X)
$\mathrm{TYPE},\ \mathrm{SUITES}_U,\ X, C_U, \mathrm{APP}_1$
$C_U, Y, C_V, \mathrm{AEAD}_{K_2}(\mathrm{ID}_{\mathrm{cred}_V}, \mathrm{sign}_y(\mathrm{cred}_V, \mathrm{TH}_2), \mathrm{APP}_2)$
Message 2
(y, Y)
$y \cdot X \rightarrow K_2, K_3$
Message 3
$x \cdot Y \rightarrow K_2, K_3$
$C_V, \mathrm{AEAD}_{K_3}(\mathrm{ID}_{\mathrm{cred}_U}, \mathrm{sign}_x(\mathrm{cred}_U, \mathrm{TH}_3), \mathrm{APP}_3)$

Figure 3.7: EDHOC messages mapped on the SIGMA-I protocol.

3.4 A Transport Layer Protocol Showdown

3.4.1 Tradeoffs between UDP and TCP

Currently, the main transport layer protocols in IP-based IoT scenarios are TCP [94] and UDP [95]. TCP provides reliability, traffic control and congestion control. However, reliable transport protocols require additional overhead information and acknowledgment packets. The standard TCP header is 20 bytes long, and TCP options can potentially add another 40 bytes. Additional drawbacks in the context of the IoT were summarized by Carles et al. [96]. TCP lacks flexibility for loss-tolerant applications and it is unsuitable for multicast (precluding it from group-oriented applications). TCP has worse performance than UDP-based solutions for non-critical monitoring with relatively frequent sensor reading updates. The alternative, UDP, is a connectionless datagram-oriented protocol. The UDP header is only 8 bytes long, and it provides a minimum of protocol mechanisms and overhead. On the downside, the delivery of packets is not guaranteed, and duplicates are not detected.

In consequence of the TCP overhead, many initial IP-based IoT deployments resorted to using UDP in combination with application-layer reliability. For example, the support for confirmable messages in CoAP. The IETF wrote several RFCs describing header compression techniques in combination with 6LoWPAN for UDP when being carried in IPv6 packets. This can potentially shrink the UDP header from 8 bytes to, on average 4, bytes. A similar proposal for 6LoWPAN TCP header compression was started but never completed [97].

Nonetheless, the need for smooth integration of CoAP with enterprise infrastructure has recently triggered the development of a CoAP over TCP specification [96]. Additionally, messaging protocols such as MQTT (Message Queuing Telemetry Transport) and AMQP (Advanced Message Queuing Protocol), both assuming TCP underneath, have achieved IoT market presence. These novel industry and standardization tendencies suggest that TCP may gain extensive support in IoT scenarios soon. With appropriate configuration, TCP can behave similarly to unicast end-to-end reliability mechanisms well-accepted for the IoT, while integrating much better with middleboxes than UDP. The specific parameters and options that can boost TCP performance in constrained scenarios are summarized in [98]. Other work on TCP optimizations present improvements based on the use of caching that reduces the number of control packets [99].

3.4.2 TLS: Transport Layer Security

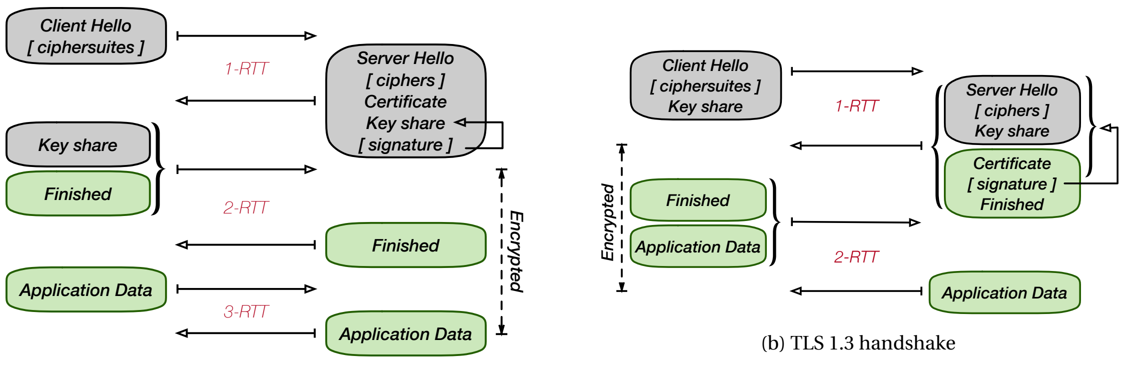
TLS is the prevalent protocol that secures today's Internet traffic. TLS is a so-called hybrid cryptosystem. It combines both symmetric key cryptography (for bulk encryption of the transport layer packets) and public-key cryptography (to establish keys for the symmetric algorithms) in one system. TLS makes heavy use of PKI (Public-Key Infrastructure) to build trust between the communicating parties. TLS requires a reliable transport protocol to function correctly, e.g., TCP. The TLS protocol consists of 2 layers. The bottom layer is fixed and is called the *record layer*. Depending on the state of the TLS protocol, the record layer encapsulates messages from the *handshake protocol*, the *alert protocol*, the *change cipher spec protocol* or the *application data protocol*.

To bootstrap a TLS connection, both peers (client and server) run the TLS handshake protocol. They negotiate the different encryption and signing algorithms by sending each other *hello messages*. The client presents its supported algorithms, in order of preference, and the server picks the first match. Next, PKI certificates and key shares are exchanged. The key shares can be generated by running either the Diffie-Hellman key exchange mechanism or the RSA key exchange. When a peer has derived the shared secret (for example, by combining its private key with the received key share), the peer uses the change cipher spec protocol to indicate that all the subsequent messages will be encrypted with the newly established keys. The final messages of the TLS handshake are called the *Finished messages*. It is the first message that is protected with the derived keys and allows both peers to check if everything went well. If something goes wrong, the alert protocol is used to notify the peer.

Over the last few years, TLS has had its fair share of issues. On the one hand, TLS has been proven a challenging protocol to implement. Many vulnerabilities, including Heartbleed [100], BERserk [101], and `goto fail;` [102] are not fundamental to the protocol and mostly resulted from a lack of testing. On the other hand, the recent proliferation of formal verification techniques for network security protocols have unveiled many design flaws in the protocol. The weaknesses range from purely theoretical (SLOTH [103] and CurveSwap [104]), to feasible for highly resourced attackers (WeakDH [105], LogJam [105], FREAK [106], SWEET32 [107]), to practical and dangerous (POODLE [28], ROBOT [29]).

3.4.2.1 TLS Reborn: TLS 1.3

In 2013, the IETF started working on a new version, called TLS 1.3 [108]. TLS 1.3 entered RFC status in March 2018. The most visible improvement is the speedup of the handshake protocol. It now takes one full RTT (Round Trip Time) less to complete, see Figure 3.8. While TLS 1.2 took two full RTTs until application data could be exchanged, TLS 1.3, only requires one RTT. The client tries to guess the key exchange algorithm the server is going to pick, allowing the client to send its key share during the first RTT. In case the client picks an unsupported algorithm, the server requests a new key share.



(a) TLS 1.2 handshake

(b) TLS 1.3 handshake

Figure 3.8: The differences between the TLS 1.2 and TLS 1.3 handshake protocols.

Another striking change introduced by TLS 1.3 is a complete purge of all the insecure legacy features and algorithms such as static RSA and static Diffie-Hellman key establishments, RC4, 3DES, MD5, and SHA1, AES-CBC, RSA-PCKS1.5 padding, DSA[3], compression, and renegotiation features. The remaining key exchange algorithms are (EC)DHE, PSK (Pre-Shared Key)-only and PSK with (EC)DHE, and the symmetric cipher suites are all AEAD algorithms. TLS 1.3 also cryptographically signs the negotiated cipher suites to prevent downgrade attacks.

Furthermore, all the handshake messages following the *ServerHello* message are now encrypted, with the freshly established key. Extensions to the protocol can also be encrypted. The key derivation functions that transform the master secret in a useable encryption key have been redesigned. The base specification now includes elliptic curve algorithms.

The session resumption negotiation, where an old master secret is used to establish an new secure session, was also improved to provide forward secrecy in TLS 1.3, by running an additional ECDHE (Elliptic Curve Diffie-Hellman Ephemeral) key exchange during the session resumption. Session resumption, combined with the client's ability to guess the server's key exchange algorithm paved the way for the 0-RTT handshake. A client can thus immediately attach application data to the client hello message. However, there are two caveats while using 0-RTT: forward secrecy is not enabled for the initial application data coming from the client, and the initial application data can be replayed.

Overall, TLS 1.3 has improved the latency, encrypts larger parts of the handshake protocol, has become more resilient against cross-protocol attacks and, removed insecure features.

3.4.3 DTLS: Datagram Transport Layer Security

Due to the wide deployment of unreliable UDP-based IoT applications, the traditional TLS protocol is unusable. DTLS is a modified version of TLS that functions properly over datagram transport [86]. Similarly to TLS, DTLS encapsulates its messages within DTLS records. The record layer supports the same four subprotocols: handshake, alert, change cipher spec, and application data protocol. To be able to handle datagram

[3]ECDSA is still supported as signature algorithm.

losses, DTLS uses ciphers with no residual state. Every datagram is protected independently from the next one, DTLS does not support any stream ciphers. In TLS, the anti-replay and message reordering protection are provided by a MAC that includes a sequence number, but the sequence numbers are implicit in the records [109]. DTLS adds explicit record sequence numbers and employs a replay window since it can not rely on message ordering from the transport layer. Furthermore, DTLS uses a retransmission timer mechanism during the handshake. UDP datagrams containing DTLS messages can grow very large during the handshake. To prevent fragmentation at the IP layer, which is no longer supported by IPv6, DTLS implements its fragmentation mechanism. We provide a more in-depth comparison of the DTLS and TLS in Chapter 7.

It is interesting to note, that apart from the obvious advantage of using an already standardized protocol, no argument has been given on actual applicability of DTLS for the IoT [2]. DTLS suffers from some of the same drawbacks as TLS, when applied to IoT applications. It is incompatible with multicast traffic, uses a verbose encoding scheme and the large datagram sizes, e.g., exchange of certificates during the handshake, make it hard to deploy the protocol in constrained networks.

At the time of writing, the DTLS 1.3 specification is still under review by the IETF. The new DTLS version will inherit many of the changes from TLS 1.3. The specification will use the new handshake pattern, support only AEAD ciphers, use the new, improved session resumption mechanism, and the new key derivation function. The DTLS 1.3 record layer has also been redesigned to optimize the header sizes, and the sequence numbers will be encrypted [110].

3.5 The Network Layer and its Challenges

The standardized IoT stack foresees the use of IPv6 to attach embedded systems to the broader Internet. IPv6 improves on IPv4 (Internet Protocol version 4) in several ways. The most noticeable change is the address space. IPv6 endpoints use addresses of 16 bytes. This solves the well-known issue of the limited IPv4 address space (2^{32} distinct addresses) but poses a challenge for the constrained IoT devices that use small link layer frames. The larger address result in an increased IP header size. Furthermore, IPv6 compatible devices must be capable of handling IP packets of 1280 bytes. When IPv6 needs to cross over IEEE 802.15.4, which uses frame sizes of 127 bytes, the need for an adaptation layer becomes apparent.

3.5.1 6LoWPAN: A Tale of Compression and Fragmentation

In the absence of the link layer security overhead, the IEEE 802.15.4 frames can transport up to 102 bytes of payload [111]. The 6LoWPAN adaptation layer optimizes the usage of this limited payload space through header compression. It also defines mechanisms for the support of operations required in IPv6, in particular, neighbor discovery [112] and address auto-configuration. Before the network layer hands an IPv6 datagram to the link layer, 6LoWPAN leverages the shared state across all devices in a local network (such as network prefix) to compress the upper-layer headers. In the case of IEEE 802.15.4, this typically results in an available application-level payload size of approximately 80 bytes. For a detailed overview of 6LoWPAN compression techniques, the reader should refer to RFC 4944 [63] and the subsequent updates. When an IPv6 packet exceeds the available link layer payload size, the 6LoWPAN fragmentation mechanism treats the (compressed) IPv6 packet as a single data field and iteratively segments this field into fragments according to the maximum frame size at the data link layer [113].

To route the different 6LoWPAN frames, 6LoWPAN supports two different schemes: *mesh-under* and *route-over* [114]. In the mesh-under scheme, the routing decisions, based on the information provided by the routing protocol, are taken by the 6LoWPAN adaptation layer. The node uses the 6LoWPAN mesh header to transport the original source and destination addresses, in EUI (Extended Unique Identifiers) 64-bit address or the 16-bit short address format. The link layer addresses that are included in the IEEE 802.15.4 header indicate the immediate next hop, reachable from the current node. Mesh-under routing has the advantage that it does not require per-hop reassembly of the full IPv6 datagram in case the original datagram was fragmented. When using the route-over mechanism, IPv6 datagrams need to be reassembled and decompressed at each hop to read the address information stored in the IPv6 header. In this scenario, routing decisions are taken on the network layer based on the IPv6 header information.

The per-hop reassembly policy of the route-over schemes results in a high end-to-end latency. Additionally, all the intermediary motes must be capable of allocating buffers of 1280 bytes to store the assembled IPv6 datagram temporarily. In response to these issues, the IETF proposed a fast fragment forwarding scheme [115]. The core idea is to inflate the first received fragment, containing the IPv6 header, deriving the link-layer destination based on the IPv6 address and storing the info in a VRB (Virtual Reassembly Buffer) as a key-value pair (fragment tag, link-layer address). Every fragment originating from the same IPv6 datagram carries the same fragment tag value. Thus, for subsequent packets, the node can quickly look up the link-layer address in the VRB and directly forward the fragment.

3.5.1.1 Security in 6LoWPAN

A security analysis done by Hummen et al. [113] shows that the current design of the 6LoWPAN fragmentation mechanism is vulnerable to low-cost DoS (Denial-of-Service) attacks. An attacker can easily block fragmented IPv6 datagrams of its choice by injecting a single spoofed 6LoWPAN fragment. An attacker can also execute a buffer reservation attack by forging a 6LoWPAN fragment. The attacker pretends to occupy the entire reassembly buffer space by sending one fragment. The inability of the nodes to detect malicious packets leads to buffer depletion. Valid fragments cannot be stored because malicious fragments occupy the space.

As a countermeasure, the authors propose a content-chaining scheme. To this end, the legitimate sender adds an authentication token, based on the principles of hash chains, to each fragment during the 6LoWPAN fragmentation procedure. The recipient can cryptographically verify the link between fragments at the time of reception and discard maliciously duplicated fragments early on the forwarding path. A second defense mechanism is based on a split buffer approach. It forces the attacker to compete with the legitimate fragments for buffer space. The attacker must now invest more resources in the attack. Before it was a DoS based on a single malicious fragment, now it resembles a flooding attack. Hence, the attacker does not benefit significantly from sending fragmented packets over unfragmented packets and must have sufficient resources to mount a flooding-based DoS attack against the target node.

3.5.2 Routing Security in the IoT

The ROLL (Routing Over Low-power and Lossy Networks) working group of the IETF designed routing solutions for the IoT. RPL (Routing Protocol for Low power and Lossy Networks) is currently the most common approach to routing in 6LoWPAN environments. It provides a framework adaptable to the requirements of particular classes of applications. In a typical constrained network, nodes connect through multi-hop paths to a small set of sink devices. RPL builds a DODAG (Destination- Oriented Directed Acyclic Graph) identified by a DODAG ID for each network sink. It accounts for link costs, node attributes, node status information, and its respective objective function. The network topology depends on a rank metric, which encodes the distance of each node to its sink, as specified by the objective function. The rank should monotonically decrease along the DODAG and towards the root node.

The header of the RPL control messages defines a security field that can provide integrity and replay protection as well as optional confidentiality and delay protection. Cryptographic MACs and signatures provide authentication over the entire ICMPv6 RPL control message. Encryption provides confidentiality of the RPL ICMPv6 message.

3.5.3 IPSec for the IoT

Several research efforts have considered IPSec as a potential solution to provide end-to-end security for the IoT. The authors mostly investigated the feasibility of porting IPSec to the IoT. They evaluated the processing overhead and energy requirements of different cryptographic suites used by IPSec, but also the memory footprints and system response time. Even though it was initially considered too heavy for constrained environments, these results led to the common conclusion that a lightweight version of IPSec is a feasible option [2].

By operating at the network layer, it can be used with any transport protocols, including potential future ones [116]. Furthermore, it ensures the confidentiality and integrity of the transport layer headers (as well as the integrity of IP headers), which can not be done with a higher-level solution like TLS. The latter, while being an advantage in some scenarios, could also hurt the deployment of smart objects. Integrity at the network layer would obstruct any protocol mappings. More precisely, the cryptographic authentication of the IP payload prevents an HTTP-CoAP mapping at the network gateway.

3.6 Security for the IoT Physical and Link Layer

The typical protocol stack for the IoT employs IEEE 802.15.4 to support low-energy communications at the physical and link layers. IEEE 802.15.4 supports communications at 250 kbitps in a short range of approximately 10 m. Several amendments exist to the standard. We are particularly interested in the IEEE 802.15.4E amendment. It defines modifications to the link layer to support time-synchronized multi-hop communications.

3.6.1 Time Synchronized Channel Hopping Protocol

TSCH (Time Slotted Channel Hopping), defined in the IEEE 802.15.4E addendum, uses up to 16 different frequency channels to provide a robust and possibly latency-bounded way of communication in the multi-hop networks. TSCH finds its roots in the TSMP (Time Synchronized Mesh Protocol) [117]. It uses a sparse time-slotted schedule, see Figure 3.9, combined with frequency channel hopping. The schedule synchronizes the nodes' accesses to the wireless transmission medium and prevents collisions, while channel hopping provides frequency diversity in the 2.4 GHz ISM band.

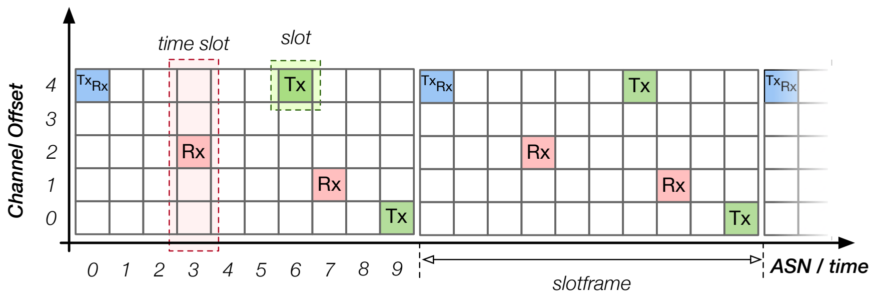


Figure 3.9: A typical TSCH schedule with multiple dedicated slots.

A TSCH schedule consists of a repeating structure, called a slotframe. A slotframe, in turn, consists of a group of timeslots. A timeslot is subdivided by channels into slots. The individual slots describe when a node should wake up to communicate with its neighbors and when it should sleep to save battery energy, see Figure 3.9. There are three types of active slots: transmission slots (Tx), reception slots (Rx), and shared slots (TxRx). The position of the slots in the schedule allows the calculation of the radio frequency channel used in the pairwise communication between nodes. Each timeslot has a unique ASN (Absolute Slot Number). The ASN is a simple counter initialized to zero at the start of the network formation. The counter is incremented for every timeslot. The value of the ASN is tracked by all the devices synchronized to the network. Each slot is uniquely identified by an ASN and a channel offset value. When a node encounters an active slot in its schedule, it turns on its radio and tunes it to the right frequency $\mathcal{F}$, using Equation 3.1.

$$\mathcal{F} = F\left[\left(\text{ASN} + \text{channelOffset}\right) \bmod n_{ch}\right] \tag{3.1}$$

Every TSCH node in the same network shares the same hopping sequence, a list of all the physical channels used by the nodes. In total, there are 16 different frequency channels, but the hopping sequence can be longer as channels are allowed to repeat in the sequence. When the length of the hopping sequence and the length of the slotframe (expressed in timeslots) are relatively prime, Equation 3.1 iterates over all the channels in a pseudorandom manner, resulting in frequency diversity and helps mitigate the effects of interference and multipath fading. In case specific frequencies experience a lot of interference, channels can be blacklisted.

A single slot in the schedule must be long enough to send a maximum length IEEE 802.15.4 frame (127 bytes) and receive a short acknowledgment frame. While the exact duration of a slot is implementation-specific, 10 ms, 15 ms, or 20 ms are commonly used. When a node wakes up for a *Tx* slot in its schedule, it checks the transmission buffer for a frame to send. If the buffer is empty, the node goes back to sleep. If there is a frame in the buffer, the node activates the radio, sends the frame, and possibly waits for the acknowledgment. For an *Rx* slot, the node turns on its radio to receive a frame, sends back an acknowledgment if required, and goes back to sleep. When the node does not receive anything within a specified time interval, it goes back to sleep. During the *TxRx* slot, the node first checks the buffer for a frame to transmit. If there is one, it proceeds as in the *Tx* slot; otherwise, it acts as an *Rx* slot. No frame reception in *Rx* mode means that either the sender had nothing to send or the frame sent by a neighboring node was lost.

Network advertisement is done by periodically sending out an EB (Enhanced Beacon) containing a payload with IE (Information Elements). IEs contain all the information needed by a node to join the network. The IEs allow the joining node to construct an initial local schedule and negotiate with the advertising node dedicated slots in which only one pair of nodes can communicate. After building the initial schedule, the node starts emitting its own EB to advertise the network further.

3.6.2 Security for the IEEE 802.15.4 Link Layer

The 2011 IEEE 802.15.4 standard defines security mechanisms at the link layer. There are several security modes available with different security guarantees. Figure 3.10 illustrates the layout of an IEEE 802.15.4 link layer frame. The *auxiliary security header field*, present when the *security-enabled bit* is set to 1 in the *frame control field*, specifies information required for security processing, including how the frame is protected, denoted by the security level, and which keying material is used.

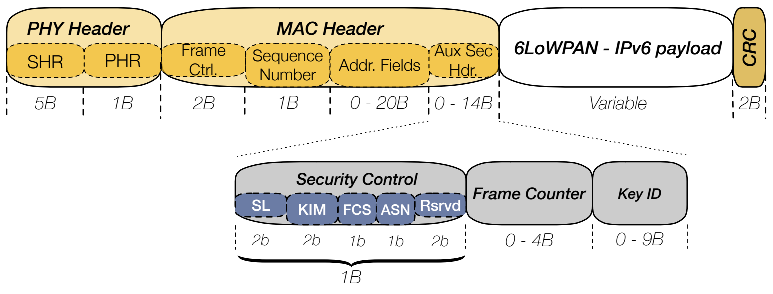


Figure 3.10: IEEE 802.15.4 link layer frame. The *auxiliary security header* describes how the frame is protected.

The standard uses the AEAD cipher, AES-CCM, to provide authentication and, optionally, confidentiality. The keys are 128 bits long. The nonce necessary for the CCM-mode can be derived in two different ways. If IEEE 802.15.4 operates in TSCH mode, the nonce is a concatenation of the 8-byte source address and ASN value, encoded in 5 bytes, else, the nonce is formed by concatenating the source address, the frame counter value and nonce security level. In both cases, the nonce is 13 bytes long. The remaining two bytes are used as counter. They increment for each 16-byte block that CCM-mode protects, see Chapter 2.

Table 3.1 shows the different security levels. IEEE 802.15.4 allows for various levels of data authentication, to minimize the security overhead and for optional data confidentiality. In the previous version of the standard, security level 4 was a level that only provided data confidentiality but without data authenticity. This level was deprecated in the 2015 standard.

The KIM (Key Identifier Mode) bits, inside the *security control field*, inform the receiver how the key used to protect the frame can be derived. The *frame counter suppression* and ASN field are set according to the mode of operation of IEEE 802.15.4, TSCH mode or non-TSCH mode.

The IEEE 802.15.4 standard also provides access control functionalities, enabling a sensing device to use the source and destination addresses of the frame to search for information on the security mode and security-related information required to process security for the message. The 802.15.4 radio chips of the device stores an ACL (Access Control List) with a maximum of 255 entries, each containing the information required for the processing of security for communications with a particular destination device.

TABLE 3.1: Security levels defined by IEEE 802.15.4.

Sec. Level	Mode	Encrypted	Integrity	MIC Length [B]
0	NONE			0
1	MIC-32		✓	4
2	MIC-64		✓	8
3	MIC-128		✓	16
4	RESERVED			
5	ENC-MIC-32	✓	✓	4
6	ENC-MIC-64	✓	✓	8
7	ENC-MIC-128	✓	✓	16

3.6.2.1 Secure Join

The 6tisch IETF working group defined a *secure join* procedure that can be used by new devices, called *pledges* to join a secured TSCH network [118]. The term *secure join* refers to network access authentication, authorization, and parameter distribution. The CoJP (Constrained Join Protocol) handles parameter distribution needed for a pledge to become a joined node. CoJP assumes the presence of a JRC (Join Registrar/Coordinator), a central entity. The pledge and the JRC share a PSK. The PSK is used to configure OSCORE to provide a secure channel to the CoJP. How the PSK is installed is not defined in the 6tisch draft, although a provisioning phase by a key exchange protocol is suggested. The secure join follows:

1. The pledge synchronizes to the network by listening for EB frames. The node that sent the EB acts as a JP (Join Proxy) for the pledge, it will relay traffic between the pledge and the JRC
2. The pledge configures its link-local IPv6 address and advertises it to the JP using IPv6 neighbor discovery protocol. This step may be omitted if the link-local address has been derived from a known unique interface identifier, such as an EUI address.
3. The pledge sends a join request to the JP to securely identify itself to the network. The join request is forwarded to the JRC.
4. In case of successful processing of the request, the pledge receives a join response from the JRC. The response contains key material used by the network for encryption and message integrity protection.

Discussion

It is now clear how the limitations of the underlying hardware have affected the design of the IoT networking stack. Starting at the physical and link layer, the necessity to consume little energy imposed strict duty cycles and small frame sizes on the communication protocols. These design choices bubbled upwards throughout the stack, affecting the choices of the higher-layer protocols. The deployment of the IPv6 protocol makes every IoT device uniquely addressable on the Internet but requires the use of an adaptation layer to provide fragmentation and header compression. At the transport layer, UDP was chosen over TCP because of its simplicity, resulting in a small code footprint, and a limited header size. Finally, CoAP has a similar functionality as HTTP, but it packs some additional features to accommodate the unique characteristics of the IoT. CoAP has also inherited some of the mechanisms of TCP for those IoT applications that require reliability.

Security for the IoT network traffic can, in a similar fashion to the traditional Internet stack, be implemented at several layers. The most significant difference with the traditional TCP/IP stack can be found at the application layer. The IoT networking stack implements object security. The object security paradigm is connectionless and supports multicast security and caching of encrypted packets. DTLS, which has been chosen as the designated network security protocol for the IoT, does not support these features. Compared to TLS, which is more prevalent in the traditional Internet, DTLS does not need a reliable transport layer. It handles datagram losses by protecting each DTLS datagram independently. In practice, this means that the cipher suites cannot have any residual state in between records [86]. However, DTLS requires more header information than TLS, and it uses the same verbose packet format and encoding scheme. For these reasons, its applicability in the constrained IoT is highly debatable.

Just like any other Internet-connected system, IoT networks can become the victim of DoS attacks. This threat places high demands on resiliency and survivability upon the network. Since IoT networks have been designed to operate under harsh conditions, we would expect them to provide a minimal level of resilience against the DoS threat. For example, adaptive routing protocols could route traffic around nodes that have been brought offline due to a DoS attack. However, multiple vulnerabilities still exist. The 6LoWPAN layer, responsible for reassembling large IPv6 packets, can easily fall victim to a simple but effective DoS attack, as described in Section 3.5.1. Researchers have proposed some countermeasures, but none have made it into the RFCs. The 6LoWPAN DoS attack can be partially mitigated by activating link layer authentication. The network will reject malicious 6LoWPAN frames if they do not possess a valid MAC. Other types of DoS attacks include excessive querying or jamming. In a TSCH network, battery depletion due to excessive querying can be mitigated by imposing a strict duty cycle, fixing the total number of allocated slots in the schedule. However, the flood of additional packets could induce high latencies for legitimate network traffic. A jamming attack in TSCH can be very destructive for the operation of the network since it breaks the time synchronization of the network. We will shed more light on this type of attacks and its defense in Chapter 6.

A separate set of security considerations applies to IoT devices bootstrapping into a network (e.g., for initial key establishment). This generally involves application-level exchanges or out-of-band techniques for the initial key establishment and may rely on application-specific trust models [119]. The Secure Join procedure for 6tisch-enabled networks describes such a mechanism. Nodes use a PSK to enroll in a network. The PSK authenticates the application layer OSCORE packets with the authorization server.

In this chapter, we considered the IoT networking stack and its defenses in the context of the Dolev–Yao threat model. We saw that the state-of-the-art security protocols could provide extensive protections to the network traffic on all layers of the stack. The model, however, limits the attacker to network only attacks, considering perfect cryptography and secrecy of the keys. IoT devices are more than other IT systems vulnerable to code injection and physical attacks. These attacks can compromise the security of the entire device and extract any secret inside. While powerful systems use multiple layers of protection mechanisms, a constrained IoT device cannot spare the resources to provide defense in depth. A single vulnerability can suffice to hack the device. Since the attackers will almost always choose the path of least resistance, device security is not an aspect we can ignore. In the next chapter, we study several software-based and hardware-based techniques to improve the device security state.

Chapter 4

System Security for Constrained Devices

Contents

Introduction

In the last chapter of our overview of the secure building blocks for the constrained IoT, we have a closer look at device security. While the previous chapter was devoted to protecting data on the network by employing different cryptographic protocols, we considered the source and destination devices to be uncompromised and *trusted*. However, this trust is often misplaced [120]. Any computing system can be compromised by adversaries, from complex server systems to the tiniest microcontrollers [121]. Once infected with malicious software, the system will no longer behave as expected.

The goal of *trusted computing* is to develop mechanisms that provide guarantees about the behavior of the software running on a device. More specifically, a device can be trusted if it always behaves as expected for the intended purpose [120]. A trusted computing architecture breaks the system into relatively small discrete modules. The modules have well-defined characteristics. This design approach then permits decision making based upon the expectations of the behavior of the modules. The most crucial discrete modules of the architecture are also known as the TCB (Trusted Computing Base). It is the set of hardware and software components critical to the security of a system. To keep the TCB as small as possible, the most trusted computing technologies build trust chains. The chains are anchored in a trusted component, which we refer to as the RoT (Root-of-Trust). A RoT is inherently indivisible, and it provides a functionality upon which all subsequent trust decisions are based. There exist several types of RoTs:

RTM (Root-of-Trust for Measurement) is a trusted implementation of a cryptographic hash or checksum function, which provides us with an integrity measurement of the system. RTMs can either by static or dynamic. A secure boot procedure is an example of an SRTM (Static Root-of-Trust for Measurement). After a successful, secure boot, the system enters a known, trustworthy state. When the RTM can be activated dynamically, for example, just before we want to execute some sensitive code, we call this a DRTM (Dynamic Root-of-Trust for Measurement).

RTS (Root-of-Trust for Storage) is a trusted implementation of a protected memory area to store cryptographic keys.

RTR (Root-of-Trust for Reporting) is similarly to the RTS a protected area in the device which contains a unique platform identity.

The concept of trusted computing is not new. It has been present in computer science security literature for quite some time. The ideas have since trickled down in COTS systems such as PCs, mobile phones, and have even started influencing the design of the more constrained IoT devices [122]. Depending on the nature of the trusted components, we can distinguish software, hybrid, or hardware-based trusted computing architectures. In hardware architectures, the user's trust is anchored in the immutability of the hardware, hybrid architectures mix trusted hardware, and trusted software and software-based designs only contain trusted software components. Hardware and hybrid designs can protect the system critical functions from advanced attackers that have compromised the OS (Operating System). Software-based designs typically cannot curb attackers once the security is breached and must resort to detection and reporting mechanisms to inform the surrounding systems of the compromise.

In this chapter, we look at the applicability of trusted computing in the constrained IoT. We discuss several software, hybrid, and hardware designs and discuss the features they provide.

4.1 Threat Model

We can define two different threat models for constrained IoT devices, based on which components make up the TCB: a treat model for software-only trusted computing and a model for hybrid and hardware-based trusted computing architectures. In both scenarios, the attacker has additional capabilities on top of the abilities described in the Dolev-Yao model [65], allowing the attacker to compromise all components of the system that do not belong to the TCB. In both attacker models, the architectures do not provide any *availability* guarantees; thus, they are all vulnerable to DoS (Denial-of-Service) attacks. Physical attacks, invasive attacks, and side-channel attacks are out-of-scope of the threat models. In the sections that discuss the software, hybrid, and hardware designs, we provide more details on the respective attacker models.

4.2 Trusted Computing Security Properties

Isolation and attestation are the most straightforward features trusted computing aims to provide. These properties can subsequently be used to build additional features such as sealing and code confidentiality.

Software Isolation is the first line of defense against many different types of attacks. Isolation puts software components and their associated data in their own protected memory space. No software outside this memory space can interfere with the runtime state of the protected software. Additionally, the protected software component has only a small set of enforced, predefined entry points. External code cannot make arbitrary calls into the code memory of the protected software, preventing malicious code of bypassing access control checks, and use techniques like ROP (Return-Oriented Programming) attacks to reveal sensitive data inside the protected domain. For high- to mid-end systems, two important classes of solutions exist that enforce isolation [123]:

1. By using virtual memory, each software module gets its own virtual address space. The operating system or hypervisor implements and guards communication channels between them (for instance, shared memory sections or inter-process communication channels).
2. By using a memory-safe virtual machine (for instance, a Java VM (Virtual Machine)), software modules are deployed in memory-safe bytecode, and the security architecture in the VM guards the interactions between them.

The listed approaches have some clear drawbacks when considered in the context of low-end systems. First, the cost (in terms of required resources such as chip surface, power, or performance) is not suited for low-end systems. Secondly, these solutions all require the presence of a sizable trusted software layer (either the Operating System OS, hypervisor, or the VM implementation). Finally, both solutions require extensive support from hardware. When discussing the hardware and hybrid approaches to trusted computing, we present several "lightweight" approaches to hardware-enforced isolation.

Attestation provides a proof of the integrity of the internal memory, e.g., RAM and flash, and the state of a device. We distinguish between *local* and *remote attestation*. A device performs local attestation when it checks the integrity of its proper components. If a third-party performs the integrity check, it is called remote attestation.

Attestation techniques use a *challenge-response* paradigm. During the protocol, the *verifier* checks the integrity of either the full memory or a specific critical function. The verifier initiates the protocol by sending the *prover* a *challenge*. If the prover can provide the correct answer to the challenge, the verifier considers the prover successfully verified. In remote attestation, the verifier is often some base station that interacts with a device in the network. For local attestation, a combination of trusted software and hardware components implement the verifier functionality. The challenge-response attestation routine uses a checksum function, denoted as $\mathcal{H}$, to compute a checksum value over the prover's memory contents. The protocol also uses a nonce, $\mathcal{N}$, to ensure the freshness of the response. The checksum acts as an RTM. Throughout the chapter, we will see that the attestation routine can be leveraged to build both SRTMs and DRTMs.

Sealing is a mechanism that encrypts code or data in such a way that it can only be decrypted under certain circumstances. The latter is often achieved by binding the encryption key to the identity of the device, the state of specific software modules or their configuration. The binding can be enforced by deriving the key from the result of a local attestation routine.

Code Confidentiality ensures that unauthorized parties cannot obtain sensitive code or data. To securely enforce this property the system must provide memory isolation and encryption. Isolation ensures that no other software components can spy on the code at runtime, while the encryption protects the code and data at rest. A mechanisms based on sealing can enforce that only a specific component can unlock the code and data.

4.3 Software-based Trusted Computing

Software-only techniques target platforms that lack any form of hardware support for trusted computing. The absence of security-critical components that can provide isolation, e.g., an MPU, actively impedes the deployment of efficient defenses against attacks. The next best thing is the detection of malicious activity on the system, for example, through attestation. Many constrained devices do not use a full-fledged OS with multiple applications, but instead, use a simple bare-metal application, making the attestation of the system more feasible.

Without isolation and code confidentiality, the attestation checksum function executed on the prover can be modified by the attacker in an attempt to forge a valid response. Hence, the RTM can be tampered with and is not inherently secure. Next, we present two implementations of software-only remote attestation functions and discuss how they try to prevent the attacker from breaking the RTM.

4.3.1 Attacker Model for Software-based Trusted Computing

Previous work [124–130] describes an attacker with the ability to infect a victim with malicious code. The code is persistent and can hide in flash memory or RAM. The attacker has full control over the device and can modify program and data memory, except ROM memory. Any cryptographic keys stored in unprotected memory are also lost. The attacker's goal is to evade detection by the attestation protocol. At attestation time, the malicious code is still running and can actively try to evade detection by moving around bits of memory. The malicious code is, however, not capable of colluding with other malicious devices during attestation time. The threat model also does not encompass an attacker, which might have copied the device's contents to a more powerful device. The model thus assumes that the attestation code executes on the correct device.

4.3.2 SWATT: SoftWare-based ATTestation for Embedded Devices

SWATT uses a checksum function that pseudorandomly accesses the memory contents of a device while calculation its attestation value. It is important to understand that an attacker cannot predict in which order the attestation function will access the different memory lines. Any malicious code active during the attestation routine must reroute memory accesses that would reveal a word of malicious code. The access must be redirected to the original memory word that would have been stored at the particular location. Malicious code can, therefore, not simply overwrite the original memory contents but must store the original memory words somewhere else in memory. The outputs of the pseudorandom function not only depends on the initial nonce value, provided by the verifier but also on the current value of the attestation function. The latter makes the attestation routine non-parallelizable. SWATT puts strict timing constraints on the response time of the prover. The verifier detects possible reroutings of memory addresses by the additional latency in the response of the prover. The authors claim that if the malicious code had to reroute even a single memory access, they would be capable of detecting the delay in the prover's response.

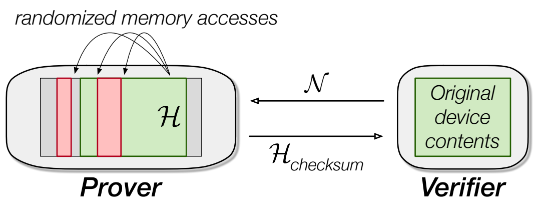


Figure 4.1: SWATT remote attestation function.

A similar SWATT-like proposal [129], which does not base its security on timing constraints, requires that the unused memory space of the device is filled with randomness. The randomness filling can happen pre-deployment or on-the-fly. For the latter, the verifier provides a seed that the prover uses in combination with a PRG to fill its free memory with random values. Having the memory occupied with known randomness (the verifier knowns the seed that generated the randomness and can thus generate the same memory contents), prevents the attacker from installing malicious code on the system. The attacker can no longer store a copy of the original memory contents elsewhere on the device, thus rerouting becomes impossible.

4.3.2.1 Malicious Software Detection Probability

To make sure no malicious code can stay hidden, SWATT must include all memory locations in the checksum, even though the address are generated randomly. The authors use the Coupon Collector's Problem. The verification procedure does $\mathcal{O}(n \ln(n))$ accesses to the memory, where n is the memory size. The result of the Coupon Collector's Problem states that if X is the number of memory accesses required to access each memory location at least once then the probability that not all memory locations are checked after $c \cdot (n \ln(n))$ access is

$$\mathcal{P}\Big[X > c \cdot n \ln(n)\Big] \leq n^{-c+1}. \tag{4.1}$$

To reduce the total number of memory accesses during the integrity check while minimizing the security risks, Perito et al. [131] propose to use the principle of PDP (Provable Data Possession). Again the attestation protocol uses randomly generated memory addresses, but instead of trying to include the entire memory in the checksum $\mathcal{H}$, only x different memory addresses are used. The probability that the malicious code stays hidden on the device can written down as follows

$$\mathcal{P}\Big[\mathcal{M}_{\text{found}}\Big] \geq 1 - \Big(1 - \frac{m}{d}\Big)^{x}, \tag{4.2}$$

where m is the number of memory words where malicious code resides and d is the total number of memory words. Figure 4.2 shows the probability of detecting malicious code on the devices after t different memory accesses. To generate the figure, we picked the memory characteristics of a typical sensor node [14]. The device contains 131,072 memory words. The malicious code residing on the device varies between 100 and 1600 bytes in size. Number of memory accesses performed by the checksum routine range from 256 to 8192, this is only 0.1% – 6% of the total number of read operations to scan the entire memory.

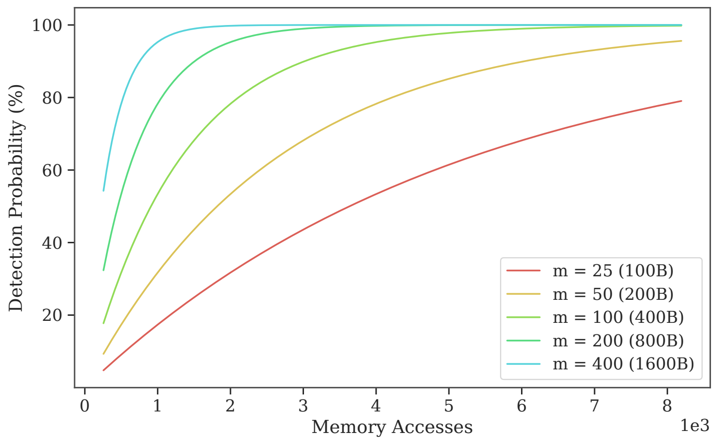


Figure 4.2: Probability of detecting malicious code on the device.

4.3.3 ICE: Indisputable Code Execution

ICE (Indisputable Code Execution) challenge-response schemes [126, 130] verify if some critical code (called the *target code*) on the prover has executed in its entirety and without interference of malicious code. To this end, the routine builds an *untampered execution environment* for the target code. It enforces the integrity of the execution environment through *self-checksumming* code. The authors define self-checksumming code as a sequence of instructions that compute a checksum over themselves while executing in a way that the checksum would be wrong or the computation would be slower if the sequence of instructions were modified [126].

ICE builds the untampered execution environment in three steps. Firstly, the integrity of the target code must be checked. Secondly, the CPU is configured to provide *atomic execution*, preventing malicious code from intervening during the execution of the target code. In practice, this is achieved by disabling all the interrupts. Finally, the target code is invoked. All these steps must happen atomically to prevent TOCTTOU (Time-of-Check-to-Time-Of-Use) type attacks. In this type of attack, the adversary regains control just after the integrity check, but before the code is executed, allowing the attacker to provide the verifier with a valid checksum and still run maliciously modified code. The ICE routine effectively provides a DRTM for the execution of the target code.

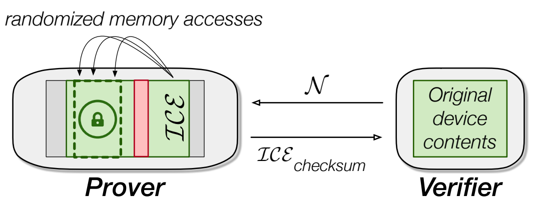


Figure 4.3: SWATT remote attestation function.

Seshadri et al. [126, 130] claim that ICE provides the above properties on COTS devices without the need for any hardware support. In previous work, the ICE-routine accepted a start address and end address for the memory region, which needed to be verified. Applying ICE to remote attestation, the ICE-function builds an untampered execution environment for the memory attestation function.

Algorithm 6 Self-checksumming code

```
1: procedure IntCheck($N$)
2:     for $0 < i < N$ do
3:         $x \leftarrow x + x^2 \vee 5 \mod 2^{16}$
4:         $\texttt{addr} = ((\texttt{addr} \oplus x) \wedge \texttt{MASK} + \texttt{code}_{\texttt{begin}}$
5:         $C_j \leftarrow \texttt{PC} \oplus \texttt{mem(addr)} + l \oplus C_{j-1} + x + \texttt{addr} + C_{j-2} \oplus \texttt{SR}$
6:         $C_j \leftarrow \texttt{ROTLEFT}(C_j)$
7:         $j \leftarrow (j+1) \mod 10$
```

Algorithm 6 presents the pseudocode of the ICE routine. As input, it takes value N, which denotes the number of iterations (and memory reads) the function performs. Firstly, the code uses a simple, fast PRG to generate random memory addresses. The function then uses a *strongly-ordered checksum*. It consists of an alternate sequence of additions and XOR operations between the memory contents, address values, PC (Program Counter), and the SR (Status Register) to compute the checksum. This sequence of operations has the property that the final value of checksum will be different with high probability if the attacker altered the sequence of operations in any way. A strongly-ordered checksum function prevents the attacker from computing the checksum out-of-order or in parallel. It also prevents the attacker from removing operations from the checksum function or replacing them with other operations in an attempt to speed up checksum computation.

4.3.4 Secure Boot for Off-The-Shelf Constrained Hardware

While on-the-fly memory integrity verification, as presented in the works above, is a desirable property for remote attestation, the lack of hardware support for an immutable attestation function makes the checksum functions complicated and computationally expensive. Some of the schemes also rely on strong timing constraints, which is not ideal since small latencies induced by the network could trigger false positives. The work presented by Schulz et al. [132] proposes an attractive alternative approach. Instead of requiring the capability to attest the memory at arbitrary moments in time, their approach only supports software integrity verification at system boot. The architecture relies on a minimal set of hardware features to implement a secure boot procedure. In return, the attestation architecture succeeds in providing a hardware-enforced SRTM, RTR, and an RTS, which allows for a stronger attacker model. More precisely, their approach can protect against collusion attacks.

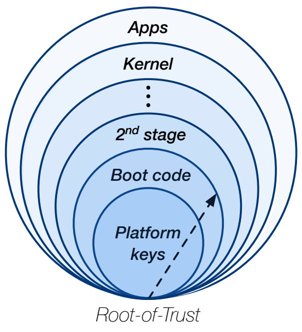


Figure 4.4: Secure boot with a Root-of-Trust for Measurement.

Although the dependency on hardware limits the deployability of their solution (e.g., not possible on an LPC1200 board [133]), the requirements are minimal and already available on the majority of COTS devices. The scheme offers a middle-ground between the pure software techniques and more exotic and expensive hardware architectures, see Section 4.4. The secure boot scheme builds a chain of trust, see Figure 4.4, by accumulating different integrity checksum values, m_x of the different boot stages in a single value M_x, where x is the index of the current boot stage being evaluated.

$$M_x \leftarrow \mathrm{PRF}_{AK}(M_{x-1}, m_x) \tag{4.3}$$

For each checksum value, it uses a PRF function combined with a new attestation key AK_x. Every AK_x is derived from the master key, AK, through the use of a KDF (Key Derivation Function). A single HMAC function can instantiate both the PRF and the KDF.

$$AK_x \leftarrow \mathrm{KDF}_{AK_{x-1}}(m_x), \quad \text{with } AK_0 \leftarrow AK \tag{4.4}$$

To securely implement the secure boot attestation scheme, the hardware must provide the following functionalities:

1. **RoT Integrity:** The TCB consists of the initial boot code, which sets up the device in a secure configuration, and the checksum function together with the AK (both form the SRTM). The RTM calculates the authenticated checksum over the second stage boot loader. The boot code must be able to protect itself, the checksum and the AK to prevent any from malicious alteration of the RTM. When leaving the POR (Power-On-Reset) phase, the boot code locks itself and the RTM, i.e., it marks the corresponding flash pages *execute-only*. Before passing control to the untrusted code, the flash page that contains the AK is read/write locked (activation of RTS & RTR). These settings must be immutable and must stay in place until the next POR.

2. **Intermediate Checksum Protection:** When calculating the m_x measurements and deriving the attestation keys AK_x, the respective boot stage must be able to operate in a secure memory that cannot be accessed by later boot stages or other unauthorized platform components. In practice, this requirement breaks down to operating in memory that is shielded against simple hardware attacks, such as the SoC on-DIE SRAM, and clearing sensitive intermediate values from memory and caches before handing control to the respective next stage.

3. **Debug Protection:** Once programmed and provisioned, the device should reject unauthorized access via external interfaces such as UART consoles, JTAG (Joint Test Action Group), or SWD (Serial Wire Debug) debug interfaces. Since the attacker model does not encompass physical attacks, this constraint should be straightforward.

By linking the integrity check with a unique device key, the trusted computing scheme can defend against collusion attacks. Under the assumption, that TCB is secure, a verifier that triggers the secure boot, and later obtains the final measurements is now sure the attestation routine was effectively run on the intended device.

4.4 Hybrid- and Hardware-based Trusted Computing

When trust is anchored in hardware, the architecture can provide more security features and stronger guarantees than the ones we saw with the software-based schemes. Below, we describe different proposals and discuss the different features they support.

4.4.1 Adversary Model for Hardware and Hybrid Trusted Computing

Trusted computing systems that have their trust anchored in hardware can defend against much more powerful attacks. The attacker is assumed to control the entire system, except for the components that are part of the TCB. Because the hardware provides isolation, it automatically supports secure storage (RTS), secure reporting (RTR), and immutable integrity checksum functions (RTM). The hardware support helps in building both dynamic on-the-fly memory integrity checks and static secure boot verification. The system can defend against collusion attacks.

4.4.2 Arm TrustZone-M

The Arm TrustZone architecture is an implementation of a TEE (Trusted Execution Environment). A TEE is an isolated execution environment in which applications can securely execute irrespective of the rest of the system [134]. GlobalPlatform has formed a committee to standardize the APIs for different implementations of the TEE, see Figure 4.5. Software inside the TEE runs alongside software executing in the REE (Rich Execution Environment).

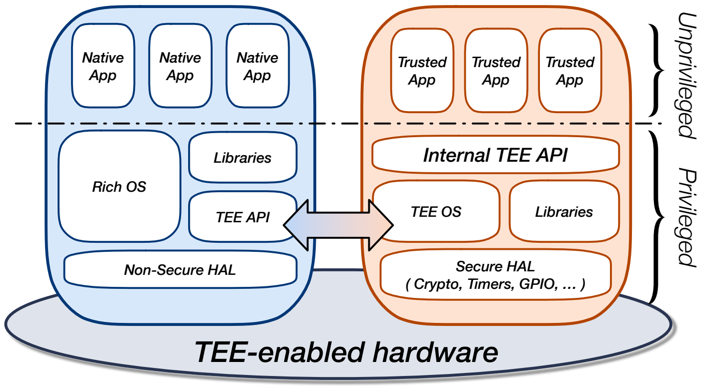


Figure 4.5: Generic architecture for a TEE-enabled system.

The original TrustZone Security Extensions targeted the powerful Cortex-A processors, but recently, with the introduction of the ARMv8 architecture for the Cortex-M line, Arm extended TrustZone to low-power CPUs. The implementation of TrustZone for the A- and M-line processors is quite different due to the distinct design requirements, but the overall functionality remains the same. For the rest of this section, when we refer to TrustZone, we mean the TrustZone ARMv8M architecture for low-power devices.

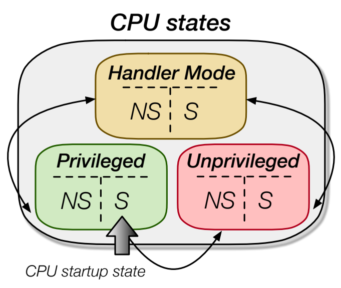
TrustZone divides the entire system in two worlds: the *secure world* (S) and the *non-secure world* (NS). The main idea is to put security-critical functions in the secure world and the remaining firmware in the non-secure world. All the system resources are divided between the two worlds, e.g., CPU cycles, memory, peripherals, etc. In each world, we can additionally have a separation between privileged and non-privileged software, as described in Chapter 1. This separation is orthogonal to the secure, non-secure worlds, see Figure 4.6a.

When a TrustZone-enabled CPU boots, it starts in the secure world. It configures the system[1] and loads a dynamic memory configuration. This configuration divides the entire device memory map (flash, SRAM, ROM, and peripherals) between the secure and non-secure world, see Figure 4.7b. Code stored on flash pages that belongs to the secure world is denoted as secure code, while its counterpart is called non-secure code. Secure code can freely access the non-secure resources, but the opposite is not true. Non-secure

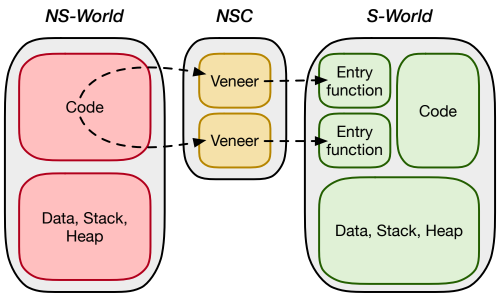
[1]On embedded systems the startup code is often responsible for initializing the C-runtime environment, setting up the clock system and configuring peripheral devices.

software can only call secure functions that are compiled with the right compiler attributes. These functions are secure world entry points, and by marking them as such in the source code, the compiler uses special instructions to handle the world transition. More precisely, the compiler uses the `SG`, `BXNS`, and `BLXNS` instructions. The `SG`, or secure gate, instruction is used to transition from the non-secure to the secure world. The `BXNS` instruction branches or returns to the non-secure world, and the `BLXNS` instruction is used by secure software to call non-secure functions. Some CPU registers are also banked. For example, there are four different stack pointers – two for the secure world and two for the non-secure world.

The non-secure code does not branch directly into the secure world but uses a *veneer*. A veneer is typically used to extend the range of branch instructions, by becoming the intermediate target. In this scenario, the Arm linker uses "secure gateway veneers" to handle the non-secure to secure state transitions. The `SG` instruction is executed in the veneer to initiate the state change. The veneers reside in a special memory, marked as *non-secure callable* (NSC). From the secure gateway veneers, the code branches further into the secure world. When the processor returns from the secure world to the non-secure world, it must clear all the non-banked CPU registers to prevent any information leakage.

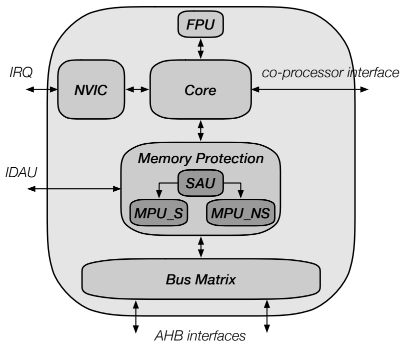


(a) Secure and Non-Secure CPU states.

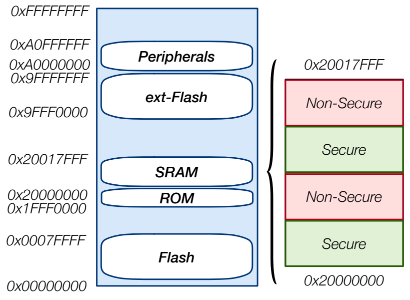


(b) Veneers in NSC memory form a bridge between the NS and S worlds.

The entire memory configuration is loaded and enforced by a special hardware component, called the SAU (Security Attribution Unit). The SAU can configure up to 8 memory areas. Optionally, chip designers can use an IDAU (Implementation-Defined Attribution Unit), which supports up to 256 statically configured regions. Similarly to the MPU, the SAU and IDAU sit between the CPU core and the system's memory and check every access, depicted in Figure 4.7a. Both hardware components check their respective configurations, and the highest security attribute is used to deny or grant access.



(a) Simplified block diagram of a TrustZone-enabled CPU.



(b) Memory map.

Figure 4.7: Hardware specificities for TrustZone-enabled devices.

Arm also added a new control signal, known as the NS-bit, to the bus infrastructure. This allows the TrustZone technology to extend the isolation between both worlds to the peripheral devices. The NS-bit communicates the current security state of a master component to a slave component. When peripherals in the non-secure world try to access secure world peripherals, the bus transaction fails.

The Arm TrustZone Extensions are implemented in two ARMv8 Cortex-M designs: the Cortex-M23 and the Cortex-M33. It is important to note that the TrustZone architecture only provides the hardware isolation features. To build any RoT-like feature, attestation routine or sealing mechanism, the user must write its trusted software that runs from the secure world. For example, the OP-TEE project implements a secure world OS for the Cortex-A TrustZone. It is designed to run from within the TEE with a Linux kernel on the REE side. Similar projects are being developed for the Cortex-M TrustZone-enabled devices.

4.4.3 SMART: Secure and Minimal Architecture for a Dynamic RoT

Eldefrawy et al. [135] describe an approach to establish a DRTM. The design goals of SMART are similar to the previously discussed ICE proposals. Both contributions aim to provide an untampered execution environment where they execute an integrity-checked piece of software. Any attempt to intervene by malicious code can be thwarted or detected. The difference lies in the approach. Where ICE tried to obtain the above properties with a software-only technique, SMART uses a minimal set of architectural changes to the hardware to enforce the DRTM. The hardware support also allows them to authenticate the prover device strongly. The SMART routine accepts a memory range $[a, b]$ and a memory address x as input parameters. The attestation function checks the integrity of the memory range and then transfers executing to x. With $x = a$ the verified code can be executed.

SMART relies on four building blocks: attestation ROM, secure key storage, MCU access controls, and reset and memory erasure. The ROM stores the attestation code and ensures it is immutable. The attestation code starts by disabling the interrupts and calculates an HMAC over the memory range. Afterward, it jumps to the address provided by x. The secure key storage guards the symmetric key used during the HMAC function. The access controls inside the CPU ensure that only the attestation code can access the secure key. This is enforced by tracking the value of the PC. The access controls also specify a single entry point for the ROM code. Software can only jump to the first instruction stored in ROM. In addition, the ROM code can only be exited from the last instruction. When the access control mechanism detects an invalid memory access, the processor resets immediately. The attestation code is carefully written to ensure it cleans up any sensitive data after it has finished. However, when the processor resets during its execution, this cleanup might be skipped. Therefore, all RAM is erased by the hardware when the processor boots or after a reset.

4.4.4 Sancus 2.0

In the previously discussed architectures, the TCB consisted of software and hardware components. In the TrustZone architecture, the TCB contains all the hardware techniques to enforce the TEE and the actual privileged code running inside the secure world. The SMART design relied on hardware modifications to the CPU and the attestation code stored in ROM.

The Sancus 2.0 architecture [123], an updated version of the original Sancus architecture proposed in 2013, differs from the others in that it provides a hardware-only TCB. Secure storage, inaccessible to software, is provided for volatile cryptographic keys and security-critical data, while static keys and algorithms are etched in silicon to omit all the software components from the TCB. The authors describe a setting where an infrastructure provider manages deployed Sancus nodes (implementation based on the MSP430). The infrastructure provider allows a variety of third-party software providers to deploy software, denoted as software modules, on the nodes. All the nodes and the infrastructure provider share a fixed symmetric key K_N, denoted as the node master key. When third-party software providers want to load a trusted application onto a node, they go through the infrastructure provider. Each software provider has a unique public identifier, SP, which is used by the infrastructure provider to derive a key $\mathrm{KDF}(K_N, SP) = K_{N,SP}$. Each software provider gets its unique secret key, $K_{N,SP}$, which allows secure deployment of software modules. The software provider uses this key to derive a software-module-specific key $\mathrm{KDF}(K_{N,SP}, SM) = K_{N,SP,SM}$. The value SM is the software module identity which can be calculated by hashing the contents of the `.text` section, the start and end addresses of the `.text` section, and the start and end addresses of the runtime data (containing the software module stack and `.bss` variables) of the software module.

$$SM = \mathcal{H}(\texttt{.text}, C_{start}, C_{end}, D_{start}, D_{end}) \tag{4.5}$$

This means that each software module is loaded on a node in a very specific address range in memory. However, the loading process itself is not protected. The address ranges for the software module binary segments are also known as the software module layout.

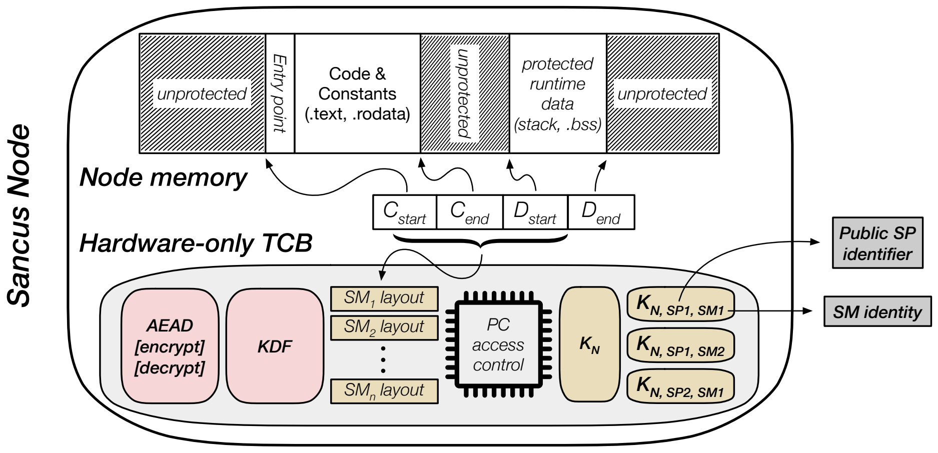


Figure 4.8: Hardware layout of a Sancus node.

4.4.4.1 Loading & Protecting Software Modules

When the untrusted OS, deployed on the Sancus nodes, receives a third-party software module, it will load it in memory. Any malicious code active on the system could now intervene and change the code in the `.text` segment and change the start and end addresses the `.text` segment and the runtime data segment. Once the loading phase finishes, the OS calls a Sancus-specific CPU instruction `protect`; the instruction does the following:

- The layout is checked to not overlap with existing modules. A new module is registered by storing the layout information in the protected storage of the processor (see Figure 4.8).
- Memory access control is enabled (see below).
- It creates the module specific key: $K_{N,SP,SM}$.

If malicious code would have changed either the contents of the `.text` segment or the address ranges, the derived key, $K_{N,SP,SM}$, would not match with the key owned by the software provider and any attempt to attest the module or set-up a secure communication channel with the module would fail.

Access to the software module memory is protected by control logic inside the CPU. The logic enforces that the .data section of a module is only accessible when instructions from the corresponding module are being executed. Code execution can only start by jumping to a well-defined entry point. The entire memory range containing the instructions is marked "execute-only". By enforcing the code entry point, the Sancus node prevents attackers from mounting ROP-like attacks. Besides memory access control, the processor also ensures that modules cannot be interrupted while being executed to prevent the register contents from leaking outside the module. The `unprotect` instruction disables the memory access controls and clears the module's code- and data. Memory access violations, intercepted by the CPU's hardware trigger a CPU and memory reset.

4.4.4.2 Remote Attestation & Secure Communication

The CPU also provides the `encrypt` and `decrypt` instructions that use a hardware-implemented AEAD cipher. These instructions can be used to provide confidentiality, integrity, and authenticity guarantees for the data exchanged between modules and their providers. The authenticated ciphertext can be sent using the untrusted OS over an untrusted network. To perform a remote attestation, the software provider sends a nonce to the node. The node returns the MAC of this nonce using the key $K_{N,SP,SM}$, and the `encrypt` instruction. Only if the module was correctly installed, the node's hardware would have generated the correct key. Therefore, if the software provider can verify the returned MAC, it knows the module is correctly running without malicious interference.

4.4.4.3 Local Attestation & Secure Linking

The Sancus architecture also supports secure linking between two software modules. If a software module SM_1 wants to link against another, already deployed module SM_2, it must include the modules ID in its `.text` segment. When SM_1 wants to call a functionality of SM_2, it first uses the `attest` instruction. This instruction verifies that SM_2 is effectively loaded in memory; it computes the identity of the module and compares the result with the ID embedded in SM_1. If the instruction succeeds, SM_2 is securely installed, and SM_1 can call the SM_2 function securely. The callee can also attest the caller by using the instruction `attest-caller`. The hardware only supports one single entry point per module. A Sancus-enabled compiler can circumvent this limitation by implementing multiple logical entry points by means of a jump table. The compiler assigns every logical entry point a unique ID. When calling a logical entry point, its ID is placed in a register before jumping to the physical entry point of the module. From the physical entry point, the code then jumps to the correct function based on the stored ID. When returning from an external function call, the module entry point is also used by providing an ID for the logical “return entry point”.

The secure linking technique is necessary to provide resource sharing in the Sancus architecture. Imagine that a memory-mapped temperature sensor is attached to the node. A special software module, SM_S, must be installed that provides an API to interact with the temperature sensor. All modules that want to access the sensor must then secure link with the SM_S module.

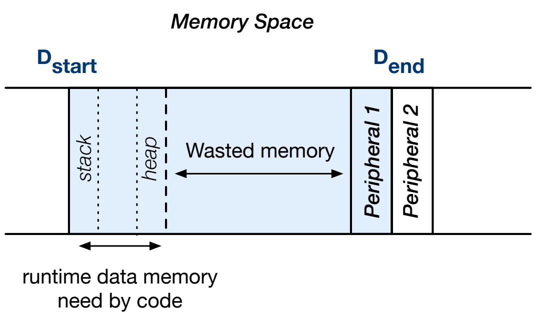
4.4.5 TrustLite

The TrustLite architecture was developed by the Intel Collaborative Research Institute for Secure Computing. It was implemented as an extension to the Intel Siskiyou Peak research platform. The authors use the terminology *trustlet* to describe protected software on the platform. The architecture provides operating-system-independent isolation, confidentiality, and attestation of trustlets. Additionally, there is trusted inter-process communication for the trustlets, secure peripherals, and support for handling memory access violations and hardware interrupts. The latter functionality is not available in the SMART or Sancus architecture.

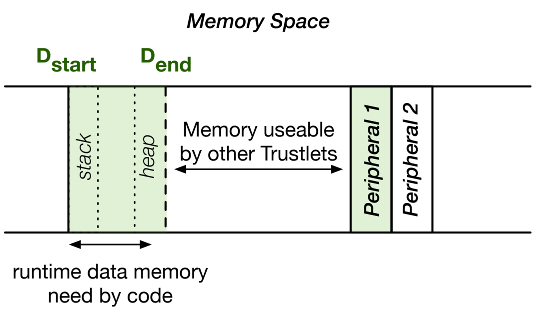
When a TrustLite device boots, the first software that executes is the *Secure Loader*. The Secure Loader sets up the different trustlets and loads their data regions into memory. The Loader also configures the system MPU to enforce isolation between the memory regions (both code and data) associated with the different trustlets. The MPU has enhanced capabilities and is called *execution-aware*. It acts like the control logic described in Sancus and SMART. Internally, the execution-aware MPU has a limited amount of registers containing code address ranges, data segments, and memory-mapped IO zones reserved for each trustlet. A truslet data segment can only be accessed if the PC is in the corresponding code segment. The configured code and data regions are also recorded in the *Trustlet Table*, a write-protected zone in memory, such that they can be verified by other trustlets or during an attestation routine. When a memory protection violation occurs, a CPU exception is raised and handled as in regular MPU designs. In particular, the MPU protection fault invalidates the executing instruction, and thus also any associated (speculatively allowed) data reads and instruction fetches. The CPU exception engine flushes the pipeline and diverts execution to the designated exception handler. This functionality is not available on Sancus and SMART nodes. Any violations trigger an immediate CPU and memory reset.

Similar to the Sancus approach, shared resources and peripheral access are protected by assigning their memory ranges exclusively to a dedicated trustlet. This trustlet then presents an abstract interface to the other trustlets. However, the Trustlite MPU allows for more flexibility by supporting the association of multiple non-continuous data regions with a single code region. Recall that in Sancus, a layout register contains one start and one end address, forcing the protected data region to be continuous. If a Sancus software module wants access to a memory-mapped peripheral, it must make sure it includes it in its protected data segment, forcing the runtime data to be allocated in the address ranges close to the addresses giving access to the peripherals. Otherwise, large chunks of memory might be lost, as shown in Figure 4.9.

In contrast to SMART and Sancus, TrustLite supports general interrupts interfering with the trustlets. The architecture ensures that no sensitive information leaks when it interrupts an executing trustlet. To avoid having to add all the interrupt handlers to the TCB, the authors decided to modify the CPU exception engine. The engine stores the stack, the instruction pointer, and general-purpose registers in the protected data region of the interrupted trustlet. Next, control is transferred to the OS by restoring its stack pointer is.



(a) If the peripheral and data section are not located close to each other, memory is wasted.


Memory Space
D_{start}
D_{end}
stack
heap
Memory useable by other Trustlets
Peripheral 1
Peripheral 2
runtime data memory need by code

(b) MPU registers allow to associate two independent data regions with the same code region.

Figure 4.9: Handling memory-mapped peripherals and shared resources in Sancus and TrustLite.

This is followed by the execution of the ISR (Interrupt Service Routine). When returning from an interrupt, the PC jumps to the trustlet entry point and restores the trustlet stack pointer in software.

Each trustlet uses an entry vector to specify the addresses that can be called by other tasks or trustlets. The trustlet itself can execute its entire code section, but other tasks or trustlets can only execute the addresses listed in the entry vector. The entry vector needs to be carefully programmed to avoid information leakage or other exploits.

Signaling and sending short messages between trustlets is done by calling the entry of a specific trustlet and passing the arguments in CPU registers. Larger messages can be transferred by passing a pointer to a shared memory region, which needs to be inside an MPU region. Trusted communication between trustlets is performed by means of a simple handshake protocol. The handshake requires that the initiator verifies the platform state, and that each party attests the other trustlet state by checking the correctness of the relevant entries in the trustlet table and MPU registers. The initiator may additionally perform an integrity check of the responder program code to ensure that it was not maliciously modified. After mutual attestation, subsequent messages can be authenticated by means of a cryptographic session token.

4.4.6 TyTan: Tiny Trust Anchor for Tiny Devices

TyTan is the final architecture we discuss. It is a hybrid trusted computing architecture that likewise uses an execution-aware MPU. It provides task isolation, attestation, secure IPC (Inter-Process Communication) with mutual authentication, and has real-time guarantees. TyTan extends TrustLite by allowing for dynamic task relocation by the OS. The authors extended FreeRTOS with a binary loader to perform the dynamic task relocation. Loading at runtime requires the allocation of memory for the new task, loading the task into memory, and preparing its stack. The MPU provides a secure driver that allows to dynamically reconfigure the MPU internal registers to protect the relocated task. Next, the task is measured by an independent RTM task. Finally, the OS is notified so that it can add the task to its scheduler.

Several static, secure software components are parts of the TCB. The secure boot task is responsible for loading all other trusted software components. Each component is isolated from the rest of the system by the MPU. The MPU driver code is also securely loaded by the secure boot task. The remote attestation task uses the result of the RTM task and a MAC algorithm to provide integrity checks to a remote verifier. Secure IPC is done by means of the IPC proxy task. It is responsible for forwarding a message m from the sender S to a receiver R. For short messages, the sender invokes the proxy with the receiver identity id_R, obtained through the RTM task, and message m as parameters. The IPC proxy task then copies m into R's memory. Since the MPU ensures that only the proxy can write to R memory, this implicitly authenticates m and id_S. For large messages, the proxy sets up shared memory that is accessible only by the communicating tasks.

The Secure Storage task seals data by storing it encrypted in non-secure memory. It is encrypted with a task key that is derived from id_t. Tasks communicate with secure storage via the secure IPC. Finally, a trusted Interrupt Multiplexor task is used to securely save the context of an interrupted task to its stack and clear the CPU registers before control is passed to the interrupt handler. Different interrupt handlers can be specified in the interrupt descriptor table, protected by the MPU.

Discussion

Trusted computing is a loaded term in the field of computer science. Despite the advanced security guarantees it can provide, the use of trusted computing mechanisms in computing systems has raised some concerns. Trusted computing is strongly correlated with DRM (Digital Rights Management), which has received notable criticism. Trusted computing can be used to restrict the freedom of the users or violate their privacy. Secure boot and sealing can be misused to prevent users from installing other software, locking out the competition. The disclosure of platform unique identifiers, keys, or configuration parameters can lead to the identification of users who wish to remain anonymous. Moreover, the terminology used to describe the technology, "trusted", is confusing. It is important to realize that trusted computing does not entail security. In trusted computing systems, users are asked to trust a set of components blindly. If one of the trusted components is breached, the security of the system is no longer guaranteed. Users are given no guarantees that the trusted components will not violate their security policies. This stands in stark contrast to trustworthy computing, where users are given proof that the system components will not violate security policies. Nonetheless, we need trusted components to build secure systems.

We started this chapter with a description of the methods used to build trust in platforms that provide no or little hardware support. We could argue that these mechanisms have no TCB, and, under the assumption that the hardware behaves as intended, the user does not have to put its trust in any component[2]. These techniques provide thus a high level of *trustworthiness*. However, the lack of hardware support has definitely its drawbacks. Software-only attestation mechanisms are defined in a limited threat model, and many proposals have been proven insecure, including the ones presented in this thesis – the work of Castelluccia et al. [136] highlights these vulnerabilities and shows how to exploit them. The authors present a compression attack and a rootkit attack. The first compresses program code to make place for a malicious payload, while the latter uses program hooks and a ROP chain to erase and reinstall the malicious code dynamically. The authors also succeeded in providing a more optimal checksum verification function. This breaks the security of the schemes that rely on timing constraints. Finally, they prove that the ICE checksum function is insecure. One might start to question why we even include these works in this overview. Although the implementations presented in the works are flawed, the ideas behind the design are still valid. The use of software-based attestation in a general manner, independent of the underlying hardware is not possible [136, 137]. In spite of that, we believe that by adapting the ideas to specific hardware platforms, aided by recent advances in formal verification [138], they can form a valid approach for legacy systems with no hardware support.

The second part of this chapter is dedicated to hardware trusted computing solutions for the constrained IoT. In general, we can distinguish two approaches. The first one, used by Arm in their TrustZone technology, divides all the resources on the system into two protection domains: a secure world and a non-secure world. Switching between both worlds is highly-regulated and enforced through special CPU instructions. The code is instrumented with these context-switching instructions at compile time.

SMART, Sancus and TrustLite all use extended version of the traditional MPU, denoted in some works as an execution-aware MPU. The MPU not only divides the memory into different zones based on the CPU privilege level but also adds isolation based on the current value of the instruction pointer. The SMART architecture provides the most basic implementation and only enforces extra rules on the attestation key and attestation routine. The Sancus architecture uses information about the memory layout of a specific software module in combination with its MPU. The drawback is that memory zones associated with a code regions must be continuous. Finally, TrustLite provides more flexibility by allowing multiple data regions to be associated with a code region. The amount of trusted software components depends on the amount of available internal MPU registers. TrustLite also provides more extensive IPC capabilities then the other research proposals.

[2]This claim is not entirely true since the users must trust the verifier in the remote attestation protocol.

Part II

A bottom-up approach to securing the Internet of Things

Chapter 5

Scalable and Secure Physical Device Identification

Contents

Introduction

In the previous chapter, we discussed how to build trust in constrained devices. There were two main approaches: software-based and hardware-based. Hardware-based constructions have strong security guarantees and properties but require expensive modifications to the underlying hardware. Except for the TrustZone architecture, none of the designs that enable lightweight, trusted computing are currently in use on a large scale. The software approach is more flexible and has the advantage that it can installed on legacy devices. The main drawback is the weaker attacker model. Software-based techniques struggle to defend against collusion attacks. Without hardware support, the devices cannot securely store a unique key, denoted as the RTR, which purpose is to bind the attestation response to the identity of the device. Therefore, an attacker can unnoticed reroute the attestation challenge from the compromised prover to a second, potentially more powerful, device to forge the correct answer.

The problem statement has some similarities with the *Sybil attack* in networks. We can define the Sybil attack as a malicious device impersonating multiple network node identities. The impersonated nodes are known as *Sybil nodes*. A particular case of the Sybil attack is the collusion attack. A powerful malicious device impersonates the identity of one or more “non-compromised nodes” during the attestation routine. A first approach to counter a traditional Sybil attack in constrained networks is through the use of secret information stored on the legitimate nodes. However, in our software-based attestation attacker model, presented in Chapter 4, we considered all secret information compromised. A second more promising method uses resource testing [139]. Resource testing assumes that each physical entity is limited in some resources. The verifier tests whether advertised node identities correspond to different physical entities by verifying that each advertised identity has as much of the tested resource as a physical device.

To strengthen device identification further, we propose to use hardware fingerprinting techniques. The basic idea is to passively or actively extract unique physical patterns, also known as features, from the device’s hardware. The features were introduced during the device’s manufacturing process. Before deployment, the features of all devices in the network are measured and securely stored in a database. During the attestation protocol, the verifier extracts the features from the prover executing the attestation function. The verifier can check the identity of the device it is communicating with, by comparing the extracted features with those stored in the database. Such an approach would allow software-based attestation techniques to defend against collusion attacks. We introduce a new terminology to reason about the uniqueness of the device’s features. We use the term *inter-device* if we compare the extracted features of device A with the features of device B. Also, we use the term *intra-device* to describe a comparison between two or more sets of features extracted from the same device (potentially under different external influences). The hardware characteristics that we consider for device fingerprinting must at least satisfy the following properties:

- They must possess enough inter-device entropy, meaning that if an attacker analyzes the features of a large number of devices, he still cannot forge the fingerprint of his target. The characteristics should not be biased, making it easy for the attack to guess or brute force an identifier.
- The intra-device features should be stable, even in the presence of environmental changes. The values of the extracted features should not change significantly between two different extractions. Significant changes in the measured values would lead to a high false-positive or false-negative rate.

In this chapter, we present our first contribution. We study the applicability and security of physical device identification in the context of the constrained IoT. We first identified two families of fingerprinting approaches: signal-based fingerprinting and PUF (Physical Unclonable Function)-based identification. We specifically selected fingerprinting techniques that rely on ubiquitous hardware components to ensure ease of deployment on legacy devices. Additionally, we also consider the verifier device to be constrained. The latter puts certain restrictions on the precision with which we can measure the prover’s features. Because of the limitations on energy supply, features that can be measured, passively are preferred over active extraction techniques. We use low-power IEEE 802.15.4E sensor nodes to perform experiments with signal-based features. PUF-based features are extracted from several types of devices. For each considered hardware component, we collect the features under varying environmental circumstances, and we present the properties. Next, we provide a discussion on its use in constrained (remote attestation) protocols and the robustness of the approach.

5.1 Signal-based Fingerprinting

Signal-based fingerprinting is a great candidate for efficient device identification in low-power networks. Wireless communication offers a rich set of features that can be extracted and analyzed. The device's identities can be observed by passively sniffing the passing network traffic. Feature extraction can happen without the cooperation of the prover. Since the features are based on meta-data derived from the radio transmission, the technique also works if the network uses encryption to protect the transmission. Examples are timing information, radio startup transient behavior, or signal strength. In general, we can identify two subsets of signal-based fingerprinting techniques: location-dependent and location-independent mechanism.

In the category of location-independent features we can find characteristics such as clock skew measurements and various physical layer RF (Radio Frequency) parameters, e.g., the transient phase at the start of transmissions [140], frequency offsets, and phase offsets [141] and AGC (Automatic Gain Control) behavior [142]. Among location-dependent features, one can use RSS (Radio Signal Strength) or CSI (Channel State Information). The main challenge is that many of these radio characteristics require specialized hardware to be measured with sufficient accuracy or are utterly inaccessible from software on the low-power devices. Unique signal characteristics that are accessible through software on our IEEE 802.15.4E enabled test platform, are RSS, and clock skew information. Drivers can read RSS information directly from a radio register during or after packet reception. Since the IEEE 802.15.4E uses channel hopping to provide transmission robustness, we can measure the RSS information on 16 different channels. The clock skew of the prover device can be measured on the physical layer through hardware timestamps on the link-layer through explicit link-layer timing information or packet inter-arrival timings in the upper layers. The IEEE 802.15.4E protocol heavily depends on timing measurements for the link-layer operation. Thus, this information is easily measurable through software timestamps triggered by radio interrupts.

5.1.1 Clock Skew Patterns

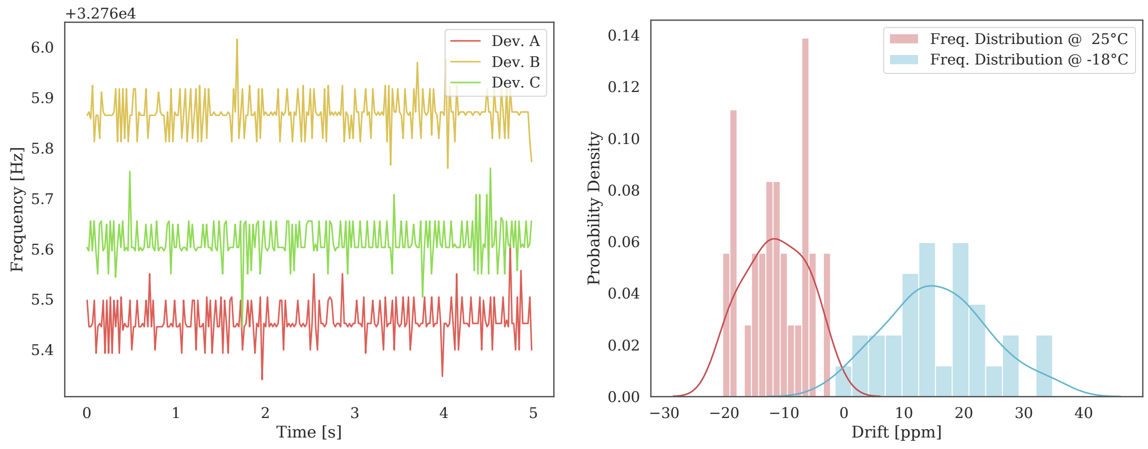
Electronic clock systems frequently use an external crystal oscillator to provide a reference signal. Alternatively, clock signals can be generated by RC-circuits, but these circuits suffer from poor frequency accuracy. The crystals have a predefined operating frequency. For example, 32.768 kHz is a typical frequency in low-power devices. The accuracy of the crystal depends on internal, and external influences. Production spread, small imperfections in the crystal and the surrounding electrical components are considered to be internal factors. External factors such as temperature changes, supply voltage, and aging also affect the clock frequency. The clock drift is commonly expressed in ppm (Parts-per-Million). Since clock drift partially depends on hardware characteristics, it could act as a physical device identifier.

5.1.1.1 Background: Clock Drift & Clock Skew Modeling

We denote the listed frequency of the oscillator component as the *target frequency*. We call the difference between the real oscillator frequency and the target frequency, the *absolute clock drift*, D_{abs}. Figure 5.1a shows the measured frequencies of several crystal oscillators measured on the CC2538 platform. The CC2538 platform uses a low-power oscillator with a target frequency of 32.768 kHz. We used a logic analyzer running at 250 Msamples per second to track the frequency of the oscillator. Small fluctuations in the clock frequency and the offset compared to the target frequency are partially caused by imperfections in the crystal and the surrounding components. The difference between the absolute drift of two systems is called the relative drift D_{rel}. The relative drift over time leads to a clock skew between two systems. The clock skew can be calculated by equation 5.1, where $\pm e_f$ [ppm] is the production spread and ϵ represents the frequency offset between both devices.

$$\delta_{\text{skew}} = \Delta t \left(\frac{1}{1 \pm e_f} + \Delta\epsilon \right) \tag{5.1}$$

The main external factor that influences the clock drift is temperature. Figure 5.1b shows the distribution of the oscillator frequency in a population of 30 devices. We notice two effects. First, there is a definite shift in average operating frequency when the ambient temperature changes. Second, when the oscillators



(a) Evolution of the oscillator frequency over time.

(b) Clock frequency distribution.

Figure 5.1: Clock frequency measurements for clock modeling.

function near their nominal operating conditions, the spread on the frequency offset is limited. The standard deviation, σ, is 5.12. However, if the temperature diverges from the optimal temperature, the impact on the oscillator frequency differs for each device ($\sigma = 8.66$). Datasheets of clock crystals often describe the relationship between the temperature and the crystal frequency, but our measurements show that this relationship is different for each device. Other causes, i.e., supply voltage and crystal aging, change more slowly [143]. We can describe the overall impact of the external factors by an additional term in the equation 5.2.

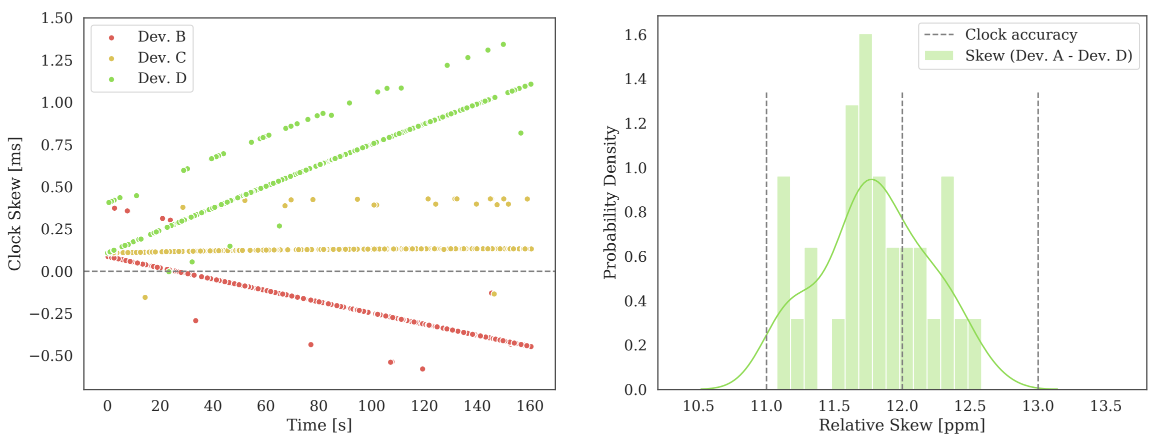
$$\delta_{\text{skew}} = \Delta t \left(\frac{1}{1 \pm e_f} + \Delta\epsilon + E(t) \right) \tag{5.2}$$

5.1.1.2 Remote Attestation and Clock Skew Identification

We can imagine a remote attestation protocol that combines one of the software checksum functions presented in the previous chapter with clock skew-based authentication. During the execution of the checksum function, the prover can be instructed to send dummy packets to the verifier. The exact moment the prover has to send the packet depends on the intermediate results of the checksum. The randomization of the transmission timings makes each run of the attestation function unique. Recall that the attestation protocol provides a unique nonce at the start of the protocol to prevent replay attacks. An attacker can only recover the exact timing information on which it has to send a dummy packet by correctly calculating the checksum. The verifier measures the relative clock skew by checking the arrival timings of the dummy packets. If the dummy packets do not arrive at the right moment, either the intermediate checksum value was wrong, or another device sent the dummy packet. In both cases, the verifier detects an anomaly.

5.1.1.3 Forging Skew Patterns

A remote attestation protocol that relies on clock skew patterns for device identification must reliably and accurately be capable of measuring the clock skew of its neighbors. Thus, a verifier cannot attest a prover that is more than one hop away, as the intermediate hop could introduce unpredictable delays. If the clock skew measurements do not have sufficient precision, the verifier must wait a long time for the skews to diverge sufficiently. Otherwise, the verifier is not capable of distinguishing two different devices. The clocks precision is a time for a single clock tick, in this case, 30.5 µs. Figure 5.2a shows the relative clock skews of three distinct devices (B, C, and D) to device A (represented by the gray dashed line). At the start of the attestation protocol, the devices need to synchronize their clocks. We notice that all three devices start with a little offset. The limited precision of the clocks causes the synchronization offset. The offset is around 10 µs, well-below the clock's precision of 30.5 µs. By using its low-power 32 kHz clock, device A can detect three different neighbors after approximatively 5 s.



(a) Evolution of the relative clock skew over time.

(b) Skew distribution for a single pair of devices.

Figure 5.2: Clock frequency measurements for clock modeling.

The total amount of devices that can be uniquely identified by the verifier during an attestation protocol depends on the total time the clocks drift freely, the precision of the verifier's measurements, the stability of the relative skew between the devices and the overall distribution of the clock drift of all the systems. If we assume that the majority of the devices have a clock drift in the interval ±30 ppm, the maximum relative clock skew is limited to ±60 ppm. After 1 min of free clock drift, a prover with a 32 kHz clock, can distinguish a maximum of 117 devices. However, the latter is only valid if the clock skew stability remains within an "accuracy slot" (see Figure 5.2b) and has a uniform distribution in the ±30 ppm drift interval. Figure 5.2b shows the distribution of 30 skew measurements between the same devices, 20 s after a synchronization point. We can see that some of the skew measurements spillover in neighboring "accuracy slots", possibly causing identification errors.

Figure 5.1b shows the distribution of the clock skews for 30 distinct CC2538 devices. We note that the drift distribution is relatively limited, meaning that many devices have similar clock skew patterns and will be indistinguishable from each other for the prover. Similar clock drift distributions are found in other research which investigated clock drift, e.g., Lanze et al. [144] studied clock drift in IEEE 802.11 networks. The reason for the limited variance in the drift distribution of the clocks is probably due to quality specification constraints by vendors. It is to be expected that for crystal oscillators which are used in communication protocols, a preselection is done by manufacturers to assure a minimum level of quality.

Clock skew-based identification mechanisms are further complicated by the fact that the relative clock skew between two device varies over time. A sudden change in temperature or a voltage drop due to depleting batteries may alter the clock skew in such a way that the identification mechanism would generate false positives or false negatives. Figure 5.1b shows that a strong temperature swing causes a non-negligible shift in clock frequency. Finally, an attacker might be able to learn the specific clock skew between the verifier and the prover by comparing the relative clock skews it shares with both devices. For example, an attacker device has a +5 ppm relative skew with the verifier and a +3 ppm skew with the prover. The attacker can derive that the relative skew between the prover and verifier is +2 ppm. It could use this information to forge the clock skew pattern of a legitimate node.

5.1.2 Received Signal Strength Features

The second signal-based identification feature uses the RSS values associated with each received packet. RSS values can provide an estimation of the link quality between two nodes [21, 145]. In addition, the measurements can be used to localize nodes in a wireless network [146, 147]. In the context of security protocols, RSS has been used to recognize Sybil attacks [148, 149] by detecting that two transmissions, supposedly coming from different identities, are coming from the same physical location. In our case, this localization information assures us that the attestation response is at least coming from the physical location where we expect the prover to be positioned. In static networks, with nodes that cannot be easily removed and replaced, this would force the attacker to intercept the attestation challenge, solve it remotely, send the solution to the compromised prover, which then transmits it to the verifier. Surrounding the attested node with trusted *helper* nodes can verify that the prover is not using radio transmission to communicate with other colluding devices in between the reception of the attestation challenge and the attestation response. An attacker would then require a physical connection to the malicious prover. Since the latter would constitute physical tampering, we did not include it in our attacker model, see Chapter 4.

5.1.2.1 Background: Multipath Fading & Channel Hopping

The verifier and prover devices typically operate in slowly changing environments, e.g., an office space where people and objects move around, affecting the measured RSS values. To better understand the impact of multipath fading on the RSS values, we use a simple scenario, shown in Figure 5.3, where two nodes, A and B, are communicating, similar to [150].

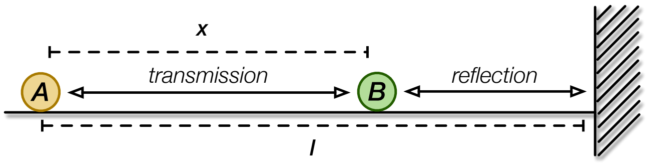


Figure 5.3: Simplified multi-path fading model.

We consider a LoS (Line-of-Sight) signal and one reflection. The signal at the receiving node can be described as

$$r = \cos\left(2\pi f t + \frac{2\pi x}{\lambda}\right) + \cos\left(2\pi f t + \frac{2\pi(2l - x)}{\lambda} + \phi\right), \tag{5.3}$$

where f is the frequency, t is time, λ the wavelength and ϕ the phase shift caused by the reflection. The received transmission power at the receiver is impacted by the phase relationship of the two converging signal. The amplitude of the signal is given by

$$A_r = cos\left(2\pi\left(\frac{l - x}{\lambda}\right)\right) \tag{5.4}$$

If the converging signals arrive in anti-phase, destructive interference occurs, also known as a *deep fade*, leading to very low RSS values and poor overall packet reception. By changing either the distance x or the carrier frequency f, it is possible to manipulate the RSS. Frequency hopping technologies, such as IEEE 802.15.4E or BLE (Bluetooth Low Energy), leverage these properties to mitigate the effects of multipath fading. The work in [150] showed that for IEEE 802.15.4E nodes operating in the 2.4 GHz band, in order to transition out of a deep fade, they can either move their location by half the wavelength (λ =5.5 cm) or change the operating frequency. As an example, Figure 5.4 shows the distribution of the different RSS measurements on separate frequency channels for a node pair operating in an office environment.

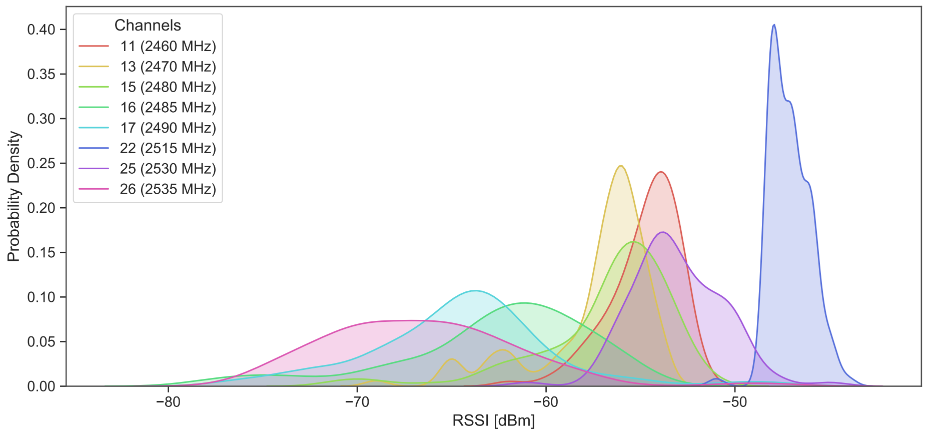


Figure 5.4: Impact of frequency shifts on multi-path fading and the resulting RSS.

Depending on the LoS distance between the communicating nodes, the necessary change in frequency ranges from a single to a few IEEE 802.15.4E channels (one channel is 5 MHz in width). The wireless communication theory also defines the concepts of *coherence length* and *coherence bandwidth*. They are used to quantify the maximum change in distance or frequency that will result in the properties of the channel being largely the same (highly correlated).

5.1.2.2 Adapting Sybil Detection Algorithms For Remote Attestation

As stated before, by calculating the origin location of the received transmissions, a verifier can thwart Sybil attacks. Two messages, supposedly coming from two distinct nodes in different locations, can be traced back to the same physical location by measuring the RSS at the multiple receivers. Four receivers are required [146], to calculate the Cartesian coordinates of a node. Their RSS values combined in a set of equations reveal the coordinates of the transmission's origin. Solving these equations each time can be quite cumbersome, and it requires, on top of the trusted verifier, three additional trusted nodes to take the RSS measurements. The works in [148, 149] present a technique that merely uses a single additional helper node to mitigate Sybil attacks. Instead of calculating the exact location, they detect changes in location if the messages truly originate from different devices. Both papers use the following approach. The signal strength of each received packet on the helper and verifier can be modeled as follows,

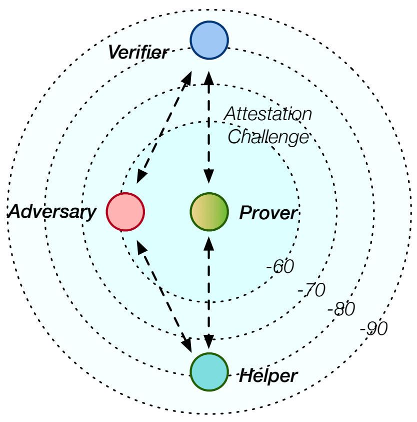
$$R_i = \frac{P_0 \cdot K}{d_i^{\alpha}} \tag{5.5}$$

where R_i represents the received signal strength, P_0 the original transmit power, K is a constant, d_i the Euclidean distance and α is the distance-power gradient. The basic Sybil attack detection algorithm functions as follows:

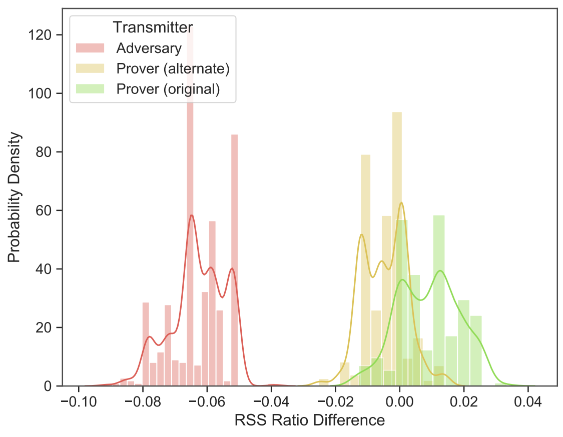
- At time t_1 the malicious node, sends out a transmission with a node identity tag ID$_1$. One verifier node and one helper node receive the transmission and log the RSS (R_v^1 and R_h^1).
- At time t_2 the malicious node, emits a second packet, now impersonating ID$_2$. Again both the verifier and the helper measure the RSS (R_v^2 and R_h^2).
- The helper transmits its data (R_h^1 and R_h^2) to the verifier node which calculates the following ratio's.

$$R_1 = \frac{R_v^1}{R_h^1} \quad \text{and} \quad R_2 = \frac{R_v^2}{R_h^2}. \tag{5.6}$$

A Sybil attack is detected if both ratios are equal ($R_1 - R_2 = 0$), or very close ($R_1 - R_2 < \epsilon$). The transmissions, supposedly coming from nodes with identifiers ID$_1$ and ID$_2$, were emitted from the same physical location. Ratios that are significantly different unveil two physically different locations.



(a) Localization of the attestation response.



(b) Detection of prover location forgery.

Figure 5.5: Adapting the Sybil detection algorithm to recognize prover location spoofing.

For our purposes, the reasoning is the inverse. The setup is shown in Figure 5.5a. Imagine the verifier has a set of trusted RSS measurements of the prover device. It can calculate the trusted ratio, R_t. The verifier sends an attestation request to the prover device. When an answer arrives, the verifier and the helper measure the RSS values. The verifier computes the ratio R_r and compares it to its trusted, stored RSS ratio, R_t, for this particular prover. If the ratios are close or equal, $R_t \overset{?}{=} R_r$, the attestation response came from the location where the prover is supposed to be. Otherwise, the response came from a different location (potentially from a more powerful malicious device that forged the attestation response). Figure 5.5b shows the RSS ratio differences for several nodes and locations. The "prover (original)" data shows the RSS ratio difference between transmissions emitted by the prover node at time t_1 and time t_2, as described in the algorithm above. As expected, the difference of the RSS ratios is centered around 0, meaning that both transmissions, at t_1 and t_2, came from the same physical location. We can clearly distinguish between the "prover (original)" distribution and the distribution describing the adversary. Since the difference of the ratios is not centered around 0, the adversary is in a different physical location than the prover. To prove that distributions are majorly influenced by the location of the nodes and not manufacturing dependent differences between the radio transceiver's circuits, we repeated the experiment and switched out the prover node with another device, "prover (alternate)". The resulting distribution is also centered around 0 and overlaps strongly with the distribution obtained from the "prover (original)" device.

However, by only using two receivers, there is a risk. The concepts of coherence bandwidth and coherence length tell us there are other locations where the channel characteristics are closely correlated to the channels currently used between the prover and the verifier and the prover and the helper. If we place the adversary's device in such a location and change its output power accordingly, we can achieve the same absolute RSS values at both receivers and also the same RSS ratios. The transmission coming from the adversary in a different physical location would not be distinguishable from the prover's transmission.

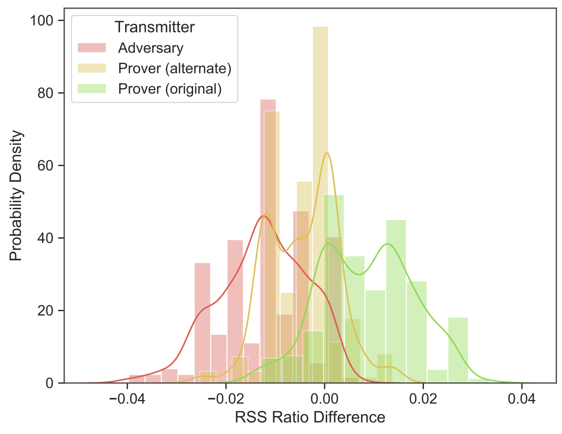


Figure 5.6: The adversary's transmission can no longer be distinguished from the prover's.

Figure 5.6 shows the resulting histograms when the adversary is in the coherence position. Calculating the exact position is difficult due to the many environmental factors influencing the state of the channel. We experimentally derived the position of the coherence position for the adversary. We can observe that the distributions now strongly overlap, and it has become complicated to distinguish a transmission coming from the adversary device from a transmission coming from the prover. The adversary would be able to intercept any attestation request, forge the response, and send it to the verifier without being detected. In the Sybil attack detection algorithm from [148, 149], this position would trigger false positives, since the algorithm only reveals one physical location for two node identities.

One solution would be to add additional receivers, increasing the difficulty for the adversary to experimentally find the coherence position that fits all three receivers at the same time. The protocol would then require three trusted device instead of two. Instead, we propose to leverage the multiple frequency channels that are available to the channel hopping protocols. The coherence position is specific for a given frequency. It can never match for the frequency channels at the same time. We tested this hypothesis and the results are depicted in Figure 5.7. For the channels 11, 14 and 17, the RSS ratios are all highly similar, resulting in indistinguishable distributions (as seen in Figure 5.6), however, on all the other channels we can

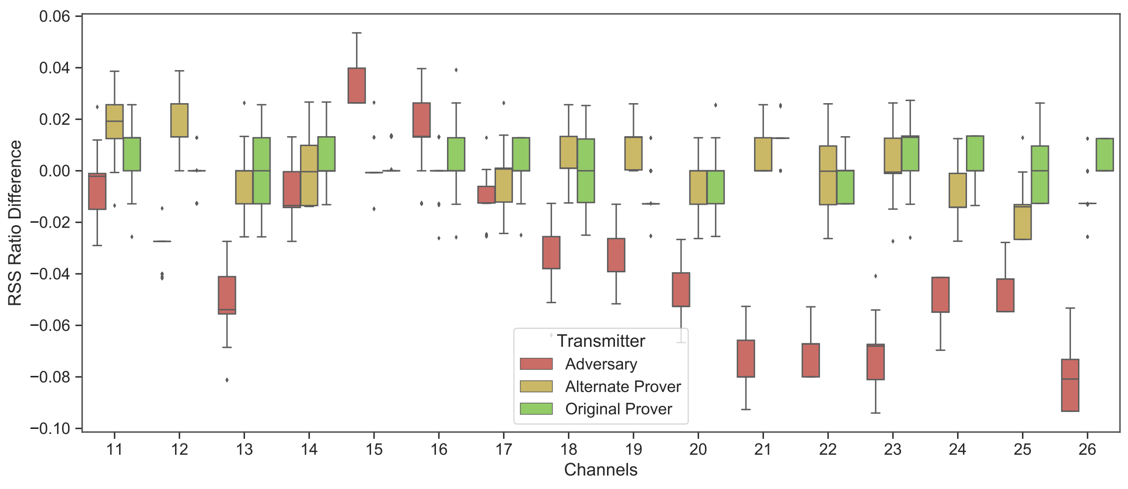


Figure 5.7: Multi-channel position verification.

clearly distinguish between the adversary's position and the prover's position. As a control experiment, we again switch out the original prover device, with an alternate device, placing it in the same position with the same antenna orientation. The RSS distributions are all centered around 0, validating the alternate prover's position as valid.

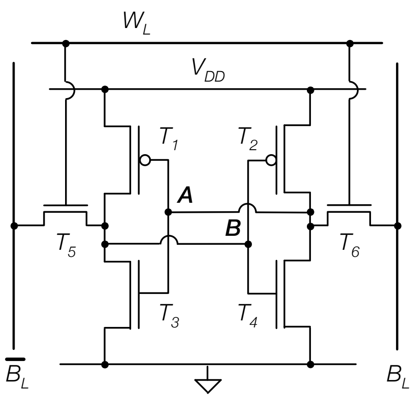
5.2 Physical Unclonable Functions

A PUF is a primitive that derives some unique information from complex physical characteristics of an IC [151]. For example, it is possible to generate some unique device-specific data from the delay characteristics of wires and transistors. PUFs depend on the random variations that occur during the IC manufacturing process, making the PUF behavior extremely difficult to predict. We can distinguish between strong and weak PUFs. A strong PUF generates a unique, unpredictable response based on a challenge. It uses the unique characteristics of the underlying physical system to perform the challenge-response mapping. An attacker with access to a challenge input and response output should not be capable of modeling the behavior of the PUF. A weak PUF typically does not accept a challenge (or has a fixed one). It generates a unique identity based on the characteristics of the internal IC. In general, a weak PUF should hide the PUF's output from the attacker, as one or a few responses are sufficient to model the PUF.

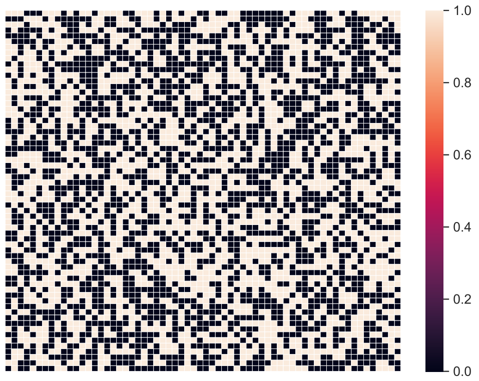
Both PUF-types can help to increase device security of constrained hardware by generating secret information that is only accessible when the chip is powered. PUFs limit the overall attack surface by forcing the adversary to mount an attack while the IC is running and using the secret. A significantly more complicated task than recovering cryptographic keys from non-volatile memory. PUFs can be implemented with various physical systems, but most common are silicon-based PUFs. Some examples are the ring oscillator PUFs and arbiter PUFs [151]. Although strong PUFs provide better security properties than weak PUFs, strong PUFs are not readily available on constrained hardware. Instead, we use a weak PUFs that can be implemented on most constrained hardware. It utilizes the internal SRAM of the MCU powering the IoT device.

5.2.1 SRAM PUFs

The SRAM PUF builds upon the observation that the power-up values of SRAM cells reveal a physical fingerprint unique to the IC. A typical SRAM cell, capable of storing a single bit of information, consists of six CMOS (Complementary Metal–Oxide–Semiconductor)-type MOSFETs (Metal-Oxide-Semiconductor Field-Effect Transistors). The cell is made up of two cross-coupled inverters and two access transistors. The two CMOS inverters drive two state nodes, labeled A and B in Figure 5.8a. Both state nodes contribute to the state of the cell. The cell has two stable states, 0 and 1, and the standby state when the cell is unpowered. In the standby state, both state nodes are discharged. When a device boots, all the cells in the IC get powered-on. The cells transition from their unstable standby state to one of the two stable states, either 0, when A = 0 and B = 1, or 1, when A = 1 and B = 0. The tendency of the cells to stabilize on a specific value depends on the relative strength of the MOSFETs that make up the cell. The MOSFET's strength, in turn, depends on process variation mismatch during the production phase of the IC and environmental noise to which the IC is subject during the powering-on process.



(a) An SRAM cell consisting of 6 transistors.



(b) Startup values for a 64 × 64 block of SRAM cells.

Figure 5.8: The race conditions that occur in an SRAM cell during power up lead to an unpredictable startup value.

We denote the cell tendency to transition to 1 or the 0 as the skew of the cell. We can identify three types of cells: 0-skewed cells, 1-skewed cells, and neutral cells. The skewed cells will always transition to the same state, regardless of noise conditions. However, neutral-skewed cells do not have a strong tendency toward either state, and during each power-up, they can transition to either 0 or 1. The unstable behavior is not necessarily due to perfectly matched inverters. Probably there is an unknown combination of volatile variables that cause the cell to flip to one state or the other under nominal conditions [152]. This latter is essential as it indicates that neutral-skewed cells may not remain neutral across all operating conditions.

5.2.1.1 Background: Statistical Properties of SRAM Startup Values

Before we look at the identification capabilities of the SRAM startup values, we study some of its statistical properties. We use two different populations of devices. We have 30 CC2538 SoCs [14] manufactured by Texas Instruments and four STM32F401RE [17] by STMicroelectronics. Both companies are IDMs (Integrated Device Manufacturers); the companies design and produce their proper silicon wafers and chips[1]. A significant difference between both device's SRAM startup values could, in addition to identifying a single device, also identify the chip manufacturer.

If we carefully observe the startup values shown in Figure 5.8b, we notice that memory shows with alternating patterns of consecutive 1 bits and consecutive 0 bits. Such a pattern can be caused by SRAM cells not being completely mutually independent. To model the correlation between the cells, we calculate the autocorrelation of the fingerprints. In Figure 5.9, we observe the autocorrelation of the SRAM cells for a single CC2538B device. We also calculated the autocorrelation coefficients for the other CC2538B devices in our possession, and we obtained the same pattern across all devices. The transparently colored red rectangle shows the 95% confidence interval. Anything outside the rectangle is with a high probability a statistical pattern. We can see that there is some correlation between a cell and its direct neighbors, the cells 256-bit positions further, and the cells on the relative positions 32 and 64, respectively.

Furthermore, the correlation coefficient pattern shown in the top half of the figure is periodic. The same pattern reappears every 256 bits. We suspect that the SRAM is organized in lines of 32 bits. In the lower half of the figure, we plot the autocorrelation on the byte-level. A byte distinctly shows a strong correlation with bytes in the relative positions 4 and 8 with respect to its address. This observation reinforces our hypothesis about the layout of the SRAM.

[1]IDM companies differ from fabless companies, such as Qualcomm and Broadcomm, which only produce the designs and then outsource the production to an IDM or a Pure Play Foundry, e.g. TMSC, GlobalFoundries.

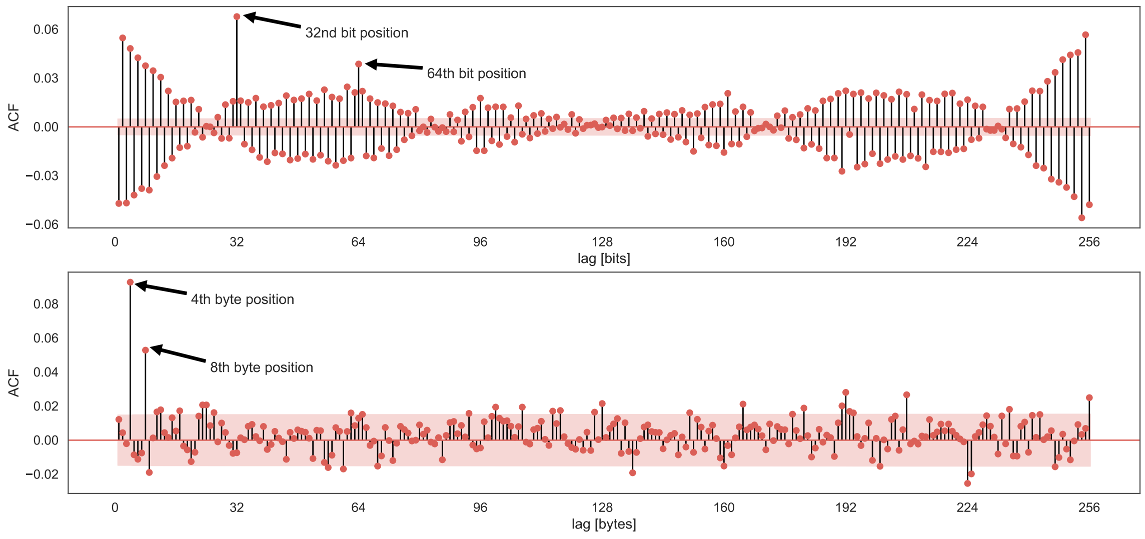
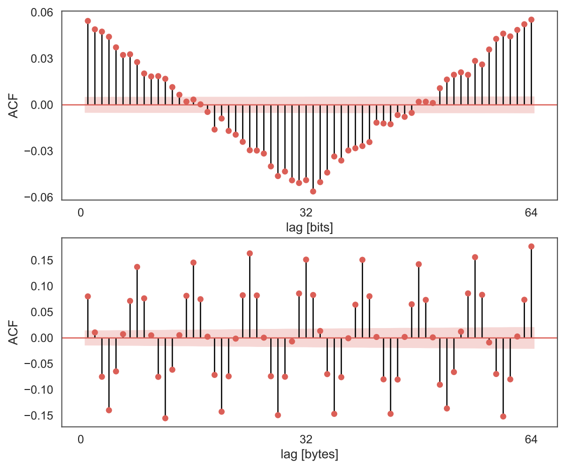
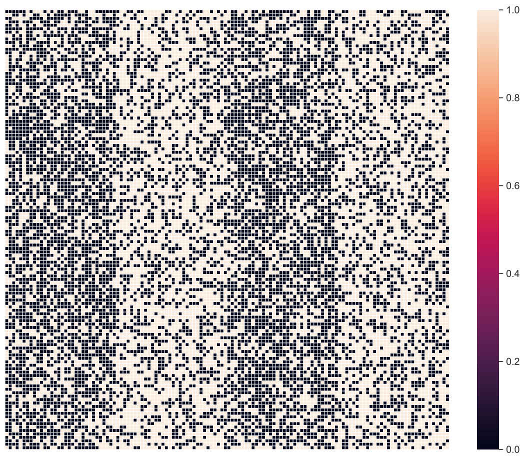


Figure 5.9: Autocorrelation for a CC2538 device on bit- and byte-level.



(a) Autocorrelation for a STM32F4 device.

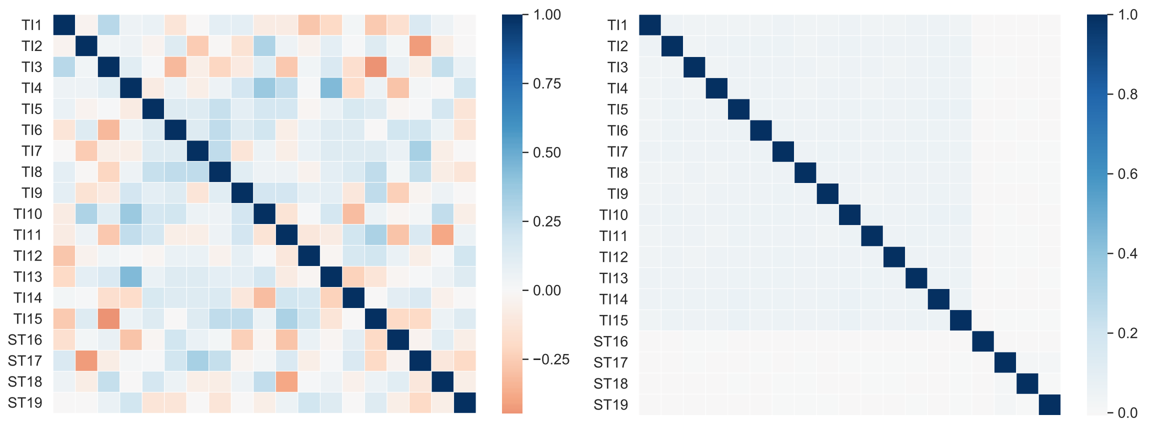


(b) Startup values for a 128 × 128 block of a STM32F4.

Figure 5.10: Recurring patterns in SRAM startup values of the STM32F4.

In Figure 5.10a, we show the autocorrelation for a single STM32F401RE. Again, we verified the autocorrelation for the remaining STM32F4 devices and obtained a similar pattern, except for one other device which did not show any clear correlation between the subsequent bits. For the three remaining devices, the periodicity of the correlation coefficients is 64-bits long. The pattern is so distinct that it can be observed by plotting the SRAM startup values of the device, shown in Figure 5.10b. The figure shows the startup values in a 128 × 128 grid. We can clearly distinguish four bands, which are alternately darker and lighter colored. The darker zones contain significantly more 0 bits, while the lighter zones mostly consist of 1 bits.

Another aspect that is of interest is how the devices correlate to each other. Figure 5.11 shows the correlation matrix for the CC2538B and STM32F4 devices. On the left, we plot how the initial 32-bit values of each SRAM chip, across different devices, are correlated. Overall we do not notice any strong correlation between the different devices. On the right, we depict the correlation between the entire memory chips of the different devices. Although the overall correlation is almost none-existing, we remark that the CC2538 devices from Texas Instruments are slightly more mutually correlated compared to the devices from STMicroelectronics. We did not find any clear explanation for this bias.



(a) Cross-device 32-bit correlation.

(b) Cross-device full memory correlation.

Figure 5.11: Cross-correlation between different devices.

5.2.1.2 SRAM Startup Values as a Fuzzy Identifier

We define a device's SRAM fingerprint as the combination of all the values an SRAM IC acquires immediately after the device is powers-on. The majority of the cells in the chip always obtain the same value for subsequent power-cycles. The remainder, neutral-skewed cells, are unpredictable. The neutral cells thus add randomness to the device's fingerprint. To investigate if a block of SRAM cells can act as a unique identifier for a device, we conducted the following experiments:

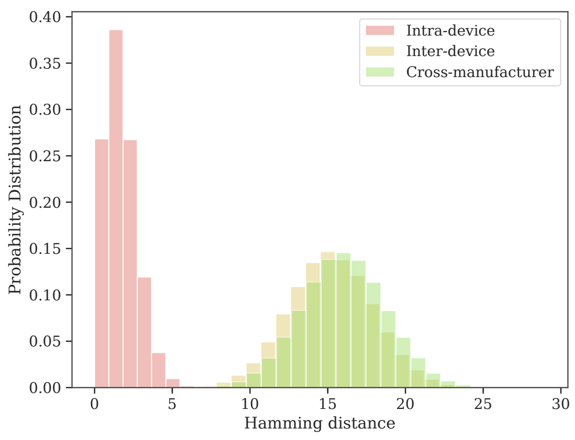
- **Intra-Device Hamming distance:** We power-cycle the same device multiple times and extract each time the first 128 bits. We calculate the Hamming distance between the subsequent SRAM dumps to analyze the stability of the device's fingerprint.
- **Inter-Device Hamming distance:** We calculate the Hamming distance between the SRAM chips of identical devices produced by the same manufacturer.
- **Cross-Manufacturer Hamming distance:** We calculate the Hamming distance between the SRAM chips of different devices produced by different manufacturers.

We start by acquiring 30 different SRAM fingerprints for the same device A, $[F_{\mathrm{A}_0}, \dots F_{\mathrm{A}_{30}}]$. We treat a single fingerprint, F_{A_i}, as an array of individual SRAM cells. For a device with 32 KiB of memory, we have an array of length 262,144. We compute the probability, for each cell, that it will to hold the value 1.

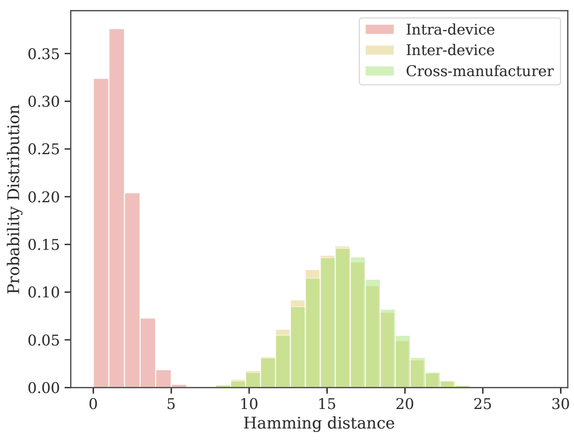
$$\overline{F_A} = \frac{\sum_{i=0}^{n} F_{\mathrm{A}_i}}{n} \tag{5.7}$$

Next, if the probability of a cell to transition to a 1 value is greater than 50%, we mark the cell as a 1-bit cell; otherwise, it is a 0-bit cell. This way, we generate an average fingerprint, denoted as $\hat{F_A}$, that represents the most probable startup state and thus the most probable identifier for the device. By averaging over multiple power-cycles, we reduce the impact of noise, making $\hat{F_A}$ more representative of the device. Once we have established $\hat{F_A}$, we gather 50 new SRAM fingerprints of the same device. For each newly extracted fingerprint, we calculate the Hamming distance between $\hat{F_A}$ and the new fingerprint per 32-bit line.

Figure 5.12 shows to Hamming distance plots. On the left of Figure 5.12, we depicted the Hamming distance distribution for all the consecutive 32-bit SRAM lines of a CC2538B device compared to itself, compared to the entire population of CC2538B devices and the four STM32F4 devices. For a single SRAM line, approximatively one cell flipped value between consecutive power-cycles. The majority of the cells are thus fixed and can be used to identify the device. Inter-device Hamming distance is on average, 15 bits. We can strongly differentiate between fingerprints coming from the same device and fingerprints extracted from two identical, but distinctive devices. An average Hamming distance of 15 for a 32-bit line is somewhat surprising. We would have expected the Hamming distance to be 16 bits. The 1-bit bias for the CC2538B devices is reflected in the slight cross-correlation, which we depicted in Figure 5.11b. This bias has also been found on other devices, see [152]. On right side, we performed the same analysis but now with respect to an



(a) HD with respect to the CC2538B.



(b) HD with respect to the STM32F4.

Figure 5.12: Hamming distances for blocks of 32-bits of SRAM.

STM32F401RE device. We can see that the inter-device and cross-manufacturer Hamming distance is fully overlapping. We already noted that there is no cross-correlation between two distinct STM32F4 devices.

Finally, we investigate the impact of the temperature on the SRAM startup values. We cool down the device to $-18\,^{\circ}\mathrm{C}$ and recalculate the intra-device Hamming distance. Instinctively we expect the SRAM to lose some of its entropy. Figure 5.13 shows the ambient temperature's impact on the Hamming distance, and thus the stability of the neutral-skewed under temperature changes.

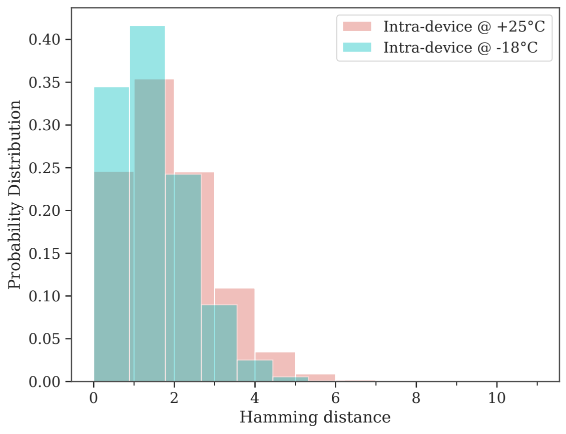


Figure 5.13: Temperature impact on the intra-device Hamming distance.

We note that some of the neutral-skewed cell seem to have transitioned to a more stable behavior if we strongly reduce the temperature of the device. The respective means of the Hamming distance distribution at $25\,^{\circ}\mathrm{C}$ and $-18\,^{\circ}\mathrm{C}$ are $1,36$ and $1,15$.

5.2.1.3 SRAM Identification for Remote Attestation

SRAM-based identification clearly improves on many of the weaknesses that plagued clock skew fingerprinting. Identifiers derived from the SRAM startup values are unique. Cold temperatures seem to stabilize some of the neutral-skewed cells but this does not affect the uniqueness of the startup values. The main drawback is that an attacker who compromises a device can simply learn the SRAM PUF response by power-cycling the device. The PUF is unprotected, and merely one response suffices to spoof identification requests from a verifier. To protect the SRAM from being easily read-out by malicious code, a trusted boot code, that wipes the SRAM startup values before any other software component becomes active, is necessary. The trusted boot code, in turn, must be protected by hardware. This approach is similar to the secure boot mechanism we discussed in Chapter 4.

Conclusions

In this chapter, we considered several techniques to derive physical fingerprints from the device's hardware. The methods we analyzed were designed and tested on low-power IEEE802.15.4E devices. We discussed two main categories: signal-based identifiers and PUF-based identifiers. None of the fingerprint approaches we saw, can provide a constrained IoT device, without hardware support, with a reliable and secure physical identifier. However, in scenarios where devices are compromised, the different techniques can provide some level of identification where otherwise none is available. Furthermore, multiple features can be combined to form a more reliable device fingerprint. Different features can accommodate different granularities in device identification giving rise to tradeoffs in false positive and false negative rates. For instance, location-dependent features such as RSS can be used in conjunction with a clock skew tracking algorithm. Devices that implement some form of software-based secure boot, see Section 4.3.4, can use SRAM to hide their unique identifier from an attacker while being powered-off.

In our discussion of the SRAM, we only considered the identification properties of the SRAM startup values. The literature contains several other applications, such as secure random number generation and secure key storage for constrained devices, which leverage the unique behavior of the SRAM chips. Secure key storage uses error-correcting codes. A popular choice are the BCH (Bose, Ray-Chaudhuri, Hocquenghem) codes to counter the instability of the neutral-skewed cells. Constructions that build true random number generators use the neutral-skewed cells as a strong source of randomness on the constrained devices. However, these constructions assume that the individual SRAM bits are independent random variables. The autocorrelation patterns we exposed clearly show that this is not true. Additional work is required to analyze how this impacts the entropy in the startup values.

Another aspect that should be considered in future works are the privacy concerns related to physical identifiers. Identifiers are important to the proper working of many protocols and services, not limited to security. However, the uniquely revealing nature of identifiers, combined with careless broadcasting over the wireless medium, raises serious privacy concerns. These concerns are even amplified in the case of physical fingerprints since a device cannot change the characteristics from which the identifier is derived. This makes it possible to track devices across location and time, possibly without the consent of the end-user. When developing protocols that make use of physical fingerprinting techniques, we should carefully consider the impact on the privacy of the end-user and strive to make sure that only authorized parties are able to track the device's identity.

Chapter 6

Vulnerabilities in Time Synchronized Link Layer Protocols

Contents

Introduction

Link-layer time synchronization is a recurring feature in low-power networking. In both TSCH and BLE, it provides an efficient way to transmit data between nodes. It is frequently combined with a schedule that tells the transmitting and listening nodes exactly when they should send and expect incoming data. The latter also helps avoid frame collisions in the network. Link layer protocols that are not time synchronized require the listening nodes to be always active, thus wasting much energy. Keeping the network synchronized is not a trivial task. To this goal, the protocol uses a global, virtual network clock, to which the individual nodes' clocks are aligned. In the previous chapter, we saw that the clock systems are susceptible to drift. When the drift grows unchecked, the node will no longer be able to communicate. Both TSCH and BLE have mechanisms to counter clock drift and keep a tight network synchronization.

The operation of the link layer and the overall energy-efficiency of the IoT device depend heavily on the network synchronization, which makes it an attractive target for adversaries. An adversary could effortlessly perform a DoS attack by blocking or interfering with critical synchronization frames in a TSCH or BLE network. In Section 6.1, we present the first part of our second contribution. We discuss in detail the synchronization mechanisms used in TSCH and why the DoS attack could be devastating to the network. A direct defense against DoS attacks targeting the synchronization mechanisms is near to impossible unless we can physically secure the access to the entire network. Alternatively, we present a novel algorithm allowing for the fast recovery of all the desynchronized nodes after the attack took place. The algorithm works distributed and acts in three phases. Before an actual attack takes place, nodes learn the relative clock drift to their *time source neighbor*, a parent node responsible for propagating timing information. During the jamming attack, the nodes desynchronize from the network but keep track of the global network clock through the use of an offline low-power timer and clock drift predictions. After the attack subsides, the desynchronized nodes use an adaptive reception window. The window gets scheduled at specific times to maximize the chances of receiving a frame from the time source neighbor, which would allow resynchronization to the network. Altogether, the algorithm minimizes the downtime of the attacked nodes and reduces the required energy to rejoin the network with a factor of 1000 compared to the current state of the art.

In Section 6.2, we discuss the second part of this contribution. We switch roles from defender to attacker. We exploit the synchronization mechanisms used in BLE to build a covert channel between a compromised *victim* device and a *covert receiver*. Covert channels allow two colluding devices to exchange data secretly. An adversary can use the channel to leak sensitive information such as passwords or encryption keys without triggering security mechanisms. We describe an attack scenario where a BLE-enabled device is infected with a malicious application. The application secretly leaks sensitive data from the compromised device by piggybacking its data on the legitimate BLE traffic generated by another application. The covert channel uses the same principles as the side-channel attack presented in [153]. The main idea of the attack is to use heat emissions of the victim device's CPU, provoked by the malicious application, to influence the frequency of the crystal oscillator. The CPU's heat causes a shift in the frequency of the crystal which translates into a measurable clock skew. In contrast to the work presented in [153], the attacker does not interact directly with the victim. Our covert channel leverages the existing traffic of a legitimate BLE connection to an innocent third device, henceforth known as the helper device, to hide the communication channel to the covert receiver. The attacker measures the varying clock skew of the victim by passively sniffing the traffic between the victim and the helper. As long as the total clock skew stays within the limits of the link layer's synchronization constraints, the covert channel remains hidden and does not disrupt the legitimate connections. We show the feasibility of the attack on three different hardware platforms: a Raspberry Pi 3B, a Motorola X 2014, and an iPhone 5s, respectively running Arch Linux, Lineage OS 14.1 (equivalent to Android Nougat 7.1.2) and iOS 11.4.1. We built a simple application that sets up a persistent BLE connection to a helper device, i.e., a smart light bulb. At the same time a malicious application schedules threads of cryptographic calculations in the background to heat the crystal oscillator and modulate bits under the form of a changing clock skew. The malicious application does not require any special privileges. As covert receiver, we use an Ubertooth One [154], a simple off-the-shelf Bluetooth sniffer, to passively measure the varying clock skew between the victim and the smart light bulb. The overall throughput of the channel is low, between 10 to 60 bits per hour, but comparable to related work that uses thermal energy to build covert channels.

6.1 Denial of Service Recovery for the TSCH Link Layer

6.1.1 TSCH PAN Formation and Maintenance

In Chapter 4, we already gave a brief introduction to the workings of the TSCH protocol. Previously we explained that TSCH uses a schedule consisting of repeating slotframe structures. The slotframe is sparsely populated with active slots for transmission or reception of frames. Each active slot uses one of 16 distinctive radio frequencies, which are mapped on channels (channel 11 to 26). The exact frequency used by each slot can be derived with the equation 3.1, which selects pseudo-randomly a channel from the channel hopping list. During the inactive slots, the nodes sleep to conserve energy. For new nodes to join the network, they must capture an EB frame. EBs are special link-layer frames that advertise the network presence to surrounding nodes. They contain multiple IEs, listing the information necessary to join the network. The following IEs are mandatory to ensure the successful joining of a new node:

- **Synchronization IE**: contains the network's current ASN
- **Timeslot IE**: carries the timeslot template that specifies the internal structure of an active slot. It tells a node among other when the radio of the receiver and sender should be activated. Figure 6.1 presents a timeslot template.
- **Channel Hopping IE**: contains the channel hopping sequence used by the network.
- **TSCH Slotframe and Link IE**: holds one or more slotframes to construct the schedule. The joining nodes can use the active slots to communicate with the advertising device.

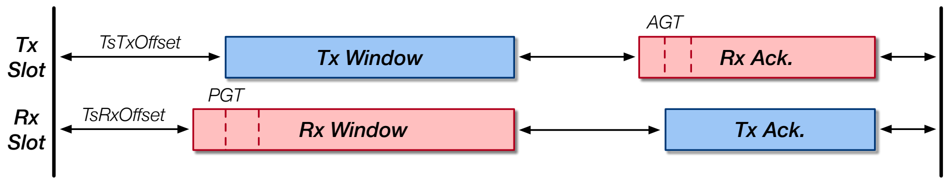


Figure 6.1: Timeslot template, the internal structure of a TSCH slot.

Not only the contents of the EB frame are essential, but also its exact arrival time. A new node uses the EB frame arrival time to align its newly constructed, local schedule with the schedule used by its neighbors. In Figure 6.1, we see that the expected frame arrival time is `TsTxOffset` seconds after the start of a timeslot. The initial bootstrap synchronization of a new node is denoted as advertisement-based synchronization [155].

After the initial join phase, nodes maintain time synchronization through regular communication with their *time source neighbor*. The time source neighbor is a parent node whose timing information is used to calibrate the clock. The entire TSCH network forms a time source tree, with the PAN (Personal Area Network) coordinator as the initial time source, see Figure 6.3a. The need for synchronization comes from the fact that independent clocks systems off the nodes drift, which unmistakably leads to clock skew, see Chapter 5. To combat the inevitable clock skew, TSCH defines a synchronization method based on exchanging KA (Keep-Alive) frames. A higher network management layer, such as 6top (6TiSCH Operation Sublayer), can create a KA frame when there has not been a clock calibration for a long time. When a node does not communicate with its time source neighbor for a time longer than the KA interval, it is considered desynchronized. KA frames contain no data; their only purpose is clock calibration.

The KA mechanism is merely a fallback system in case there is not enough application traffic in the network to keep the nodes synchronized. Otherwise, a node can use generic link-layer frames exchanged with its time source neighbor, to perform frame-based or acknowledgment-based synchronization. Both synchronization methods require the receiver to measure the clock skew (δ_{skew}) on frame reception. The clock skew is defined as the number of clock ticks between the expected and actual time of the frame arrival. Because of the clock skew, nodes have to use a guard time when turning on the radio, see Figure 6.2. The length of the PGT (Packet Guard Time) also directly affects the KA interval. If we consider a maximum relative clock skew, δ_{skew}, of 60 ppm and a PGT of 2600 µs [155], then the KA interval can be calculated as follows:

$$|\mathrm{KA}| = \frac{PGT}{2 \cdot \delta_{\mathrm{skew}}} \tag{6.1}$$

For the above values, the KA interval amounts to ± 21 s. Therefore, a receiver turns on its radio PGT/2 earlier than it expects the arrival of a frame (see Figure 6.2. If at least one frame is sent every KA interval, then $|\delta_{\mathrm{skew}}| < \mathrm{PGT}/2$ and the reception of the frame should be guaranteed. When the node does not receive a frame preamble after a full PGT, it will turn its radio off and go back to sleep. The same procedure is used on the transmitter side when it expects an acknowledgment.

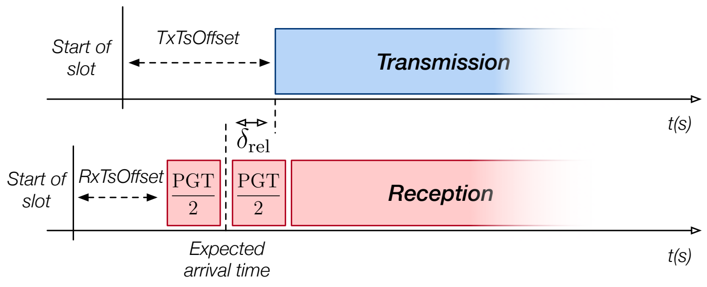


Figure 6.2: Frame-based synchronization.

The frame-based synchronization and acknowledgement-based synchronization schemes are defined as follows:

1. The receiver records a timestamp t_s when it detects the frame start symbol.
2. It measures the clock skew as follows:

$$\delta = \mathtt{RxTsOffset} + \frac{PGT}{2} - t_s, \tag{6.2}$$

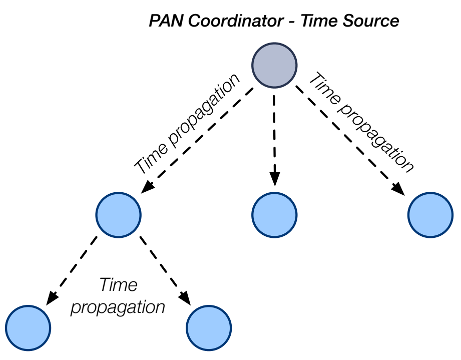
   where `RxTsOffset` represents the time between the beginning of the timeslot and the activation of the radio. For Tx and Rx slots these offsets are called TsTxOffset and TsRxOffset, respectively, see Figure 6.2.
3. In case of frame-based synchronization, the receiver will use the measured value of δ_{rel} to calibrate its clock. When using acknowledgement-based synchronization, the receiver will add δ_{rel} to the acknowledgement frame, which is sent back to the transmitter. In this scenario, the transmitter calibrates its clock.

6.1.2 DoS Vulnerability

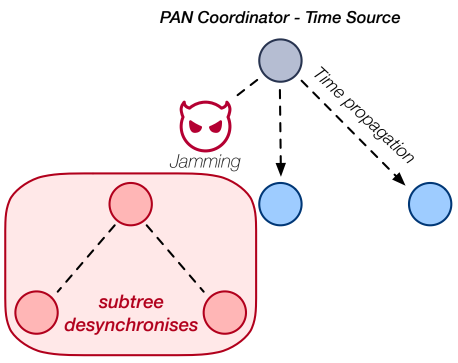
The previous section described how a node joins a network after it has captured an EB frame and how to maintain the synchronization. However, we omitted one crucial bit of information. How can a new node capture an EB without ever being synchronized to the network? A new node has no information on the schedule, nor is it synchronized in any way to the network. An EB can be transmitted on one of the 16 different frequency channels, significantly lowering the chances that a new node will overhear an EB when it randomly scans a channel. If we consider that a joining node is only in range of one neighbor, we ignore the fact that there can be transmission errors and frame collisions, and there is one EB sent every slotframe, the average joining time is

$$T_j = t_s * \frac{C+1}{2} \tag{6.3}$$

where C is the number of channels and t_s is the length of a slotframe expressed in seconds. The IEEE standard [156] does not provide a more efficient mechanism to capture EBs, making the initial join procedure extremely expensive. Some work has been done to accelerate the initial join phase in TSCH networks, but they rely heavily on specific network advertisement policies and depend on the density of the network the new node wants to join [157, 158].



(a) TSCH time source tree.



(b) Desynchronization of subtree.

Figure 6.3: DoS vulnerability in the TSCH time synchronization mechanism.

It is essential to realize that the expensive join procedure is not only performed when the node joins the network for the first time, but each time it desynchronizes from the network. Once a node is desynchronized, i.e., when the drift has grown larger than the PGT/2, there are no mechanisms in place to keep the clock skew in check. The only way to rejoin the network is to perform a new advertisement-based synchronization, requiring active scanning of the different channels. Desynchronization can frequently occur due to interference or depleted batteries. We saw that a mere 21 s without communication with the time source neighbor could suffice to desynchronize a node. Furthermore, if a node desynchronizes and it is not capable of resynchronization to network within a KA interval, its children also desynchronize, causing a chain reaction that can desynchronize an entire subtree of the network, see Figure 6.3b. Resynchronizing the entire subtree to the network can take up to $n \times T_j$ seconds, where n is the number of nodes in the subtree and requires a substantial amount of energy.

It is now clear that an attacker can perform a powerful DoS attack on the network without investing much energy. By jamming a single communication link in the network for the duration of a KA, the attacker can take down a large part of the network. If the attacker jams a communication link close to the root of the time source tree, the number of desynchronized nodes, and thus the impact on the network, grows. Not only does it takes a long time for all the nodes to resynchronize. The procedure is also expensive energy-wise.

6.1.3 Fast Recovery for Desynchronized Nodes

Preventing DoS attacks is difficult, and in a low-power network, we cannot invest much energy in countering these types of attacks. However, we can try to minimize the damage done to the network. We designed a fast, distributed resynchronization scheme that uses clock skew prediction, derived from the drift model presented in Chapter 5, to quickly rejoin the network after being desynchronized. It is backward compatible with the standard TSCH protocol. Nodes implementing the scheme do not rely on any actions taken by other nodes and can thus coexist with “standard” nodes. Based on the hardware characteristics presented back in Chapter 1, we assume that the radio is the primary energy-consumer. Our algorithm is tailored to minimize the time during which a node actively scans for frames. To decide when the node needs to reactivate its radio, we use prior information stored on the node from its initial synchronization to the network. The scheme computes the relative clock skew with its time source neighbor to predict possible arrival times of the incoming frames. To capture the frames, we use an adaptive listening window, which compensates for any unforeseen drift causes and inaccuracies in the prediction scheme. While the standard TSCH protocol specifies a fixed duration for the PGT, we adapt the PGT based on the computed clock skew, the minimal duration being default PGT (2600 µs).

6.1.3.1 Principles of the Predictive Rejoining Scheme

Phase 1. The first stage is active when the nodes are still synchronized. Every node periodically measures its clock skew with respect to its time source neighbor, δ_{skew}, by using the frame-based or acknowledgment-based synchronization method and calibrates its clock accordingly, see Figure 6.2. The rejoining scheme derives the average clock skew between itself and its time source neighbor. The average clock skew is continuously updated with freshly measured clock skew values as long as the node stays synchronized. A node also stores the ASN value of the slot where the last synchronization occurred. After each calibration, the node restarts a low-power timer that counts the elapsed slots until a new clock calibration takes place. The timer keeps running even when the node is in the low-power sleeping mode to save energy. A node becomes desynchronized when the time elapsed since the last clock calibration exceeds the KA interval. At this instant, we cannot guarantee that frames will be captured as the relative clock skew, δ_{skew}, might be larger than PGT/2. When the desynchronized node no longer exchanges frames with its time source neighbor, the clock drift causes the skew to grow further, see Figure 6.4. The ASN of the desynchronized node for a given slot will no longer correspond to the ASN used by its time source neighbor, and the node will fail to calculate the correct frequency channel.

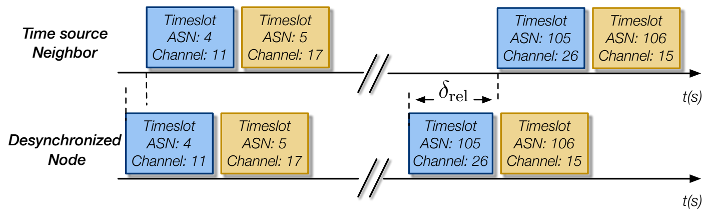

Time source Neighbor
Timeslot ASN: 4 Channel: 11
Timeslot ASN: 5 Channel: 17
Timeslot ASN: 105 Channel: 26
Timeslot ASN: 106 Channel: 15
t(s)
δ_{rel}
Desynchronized Node
Timeslot ASN: 4 Channel: 11
Timeslot ASN: 5 Channel: 17
Timeslot ASN: 105 Channel: 26
Timeslot ASN: 106 Channel: 15
t(s)

Figure 6.4: The drift keeps on growing once the child node is desynchronized.

Besides the invalid ASN, the information included in the EB frame (slotframe size, channel hopping sequence, and timeslot template) during the initial synchronization phase remains valid after desynchronization. We can make this assumption because changing these parameters would require a full update of the whole network.

Phase 2. The second phase starts when the node detects it is desynchronized from the network. The node can notice the desynchronization either directly after the KA interval overflows or after an extended period of time when the node wakes up from low power mode. The node now starts the procedure to rejoin the network. It will try to locate the transmissions from its time source neighbor in order to realign itself to the network. The node starts by adding the value of the low-power timer to the most recently stored ASN. The resulting value gives the node the first estimation of the ASN currently used by the network. Next, the node calculates the relative clock skew by multiplying the most recent drift estimation with the total time the node was desynchronized. If the total relative skew since the last clock calibration is larger than an entire timeslot, the node updates its estimated ASN value. For example, on the right side of Figure 6.4, the desynchronized node estimates an ASN of 105, while due to the clock drift, the actual ASN used by the network, and the time source neighbor, is 104. By accounting for the clock skew, a node detects that the estimated ASN is off by one because the clock skew is larger than a timeslot.

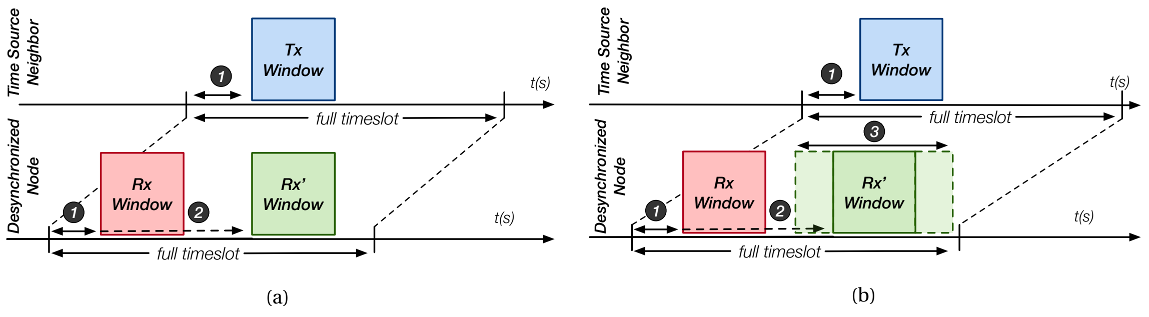


Figure 6.5: A node pair illustrating the predictive rejoin scheme. (1) The standard `TsRxOffset` and `TsTxOffset` values. (2) The predicted relative skew allows for rescheduling of the Rx window (shown as the RX' window). (3) Rx window expands for the unpredictable skew after desynchronization.

Phase 3. In the final step, the desynchronized node looks at its stored schedule for the first upcoming Rx slot for communication with its time source neighbor, and it goes to sleep until the Rx slot. When the node wakes up, it reestimates its additional skew and adjusts if necessary. The schedule contains the channel offset of the respective Rx slot, and combined with the estimated ASN, the node can tune its radio to the right frequency channel. The node uses the estimated skew to reschedule the Rx window such that it aligns with the Tx window of its time source neighbor (see Figure 6.5a). The node activates its radio for a full PGT interval. If the time source neighbor does not send a frame in the Rx slot, the node repeats the final step and locates the next Rx slots. When the node successfully captures an incoming frame from its time source neighbor, it recalibrates its clock with the frame-based synchronization method. The low-power timer resets, and the node is again synchronized with its time source neighbor, and by extension, the entire network. The captured frame does not necessarily have to be addressed to the desynchronized node, but the transmitter must be the node's time source neighbor. The node can use any frame from the time source neighbor to synchronize (its contents are not relevant), which means that an encrypted frame can also be used to resynchronize, only the arrival time is needed to recalibrate the clock.

6.1.3.2 Compensation for Unpredictable Drift Causes

Unforeseen drift fluctuations after desynchronization and finite arithmetic precision during the skew estimation may create an undetectable skew between the Tx window of the time source neighbor and the Rx window of the desynchronized node, see Figure 6.5b. We must keep in mind that the drift estimation of Phase 1 is only an ephemeral snapshot of the situation. The drift can change any moment after desynchronization due to external causes, e.g., temperature fluctuations. The second cause of scheduling errors is the finite precision when the node calculates the relative skew. Some platforms only use 16-bit integer arithmetic. Rounding errors become significant when a node is desynchronized for more extended periods.

The adaptive listening window grows steadily over time (see Figure 6.5b), to compensate for the skew caused by unforeseen drift causes. Its size depends on the total time the node was desynchronized, the arithmetic precision, and the worst-case external drift change, e.g. the largest temperature change, expected in the network. The window expands in both directions as the desynchronized node does not know in which direction the relative drift will evolve. Nodes with a temperature sensor and having sufficient energy can apply drift compensation in real-time. Its goal is to keep the actual relative drift as close as possible to the previously established relative drift estimation. If the node obtains a stable relative skew by compensating the drift changes that occur after desynchronization, our scheme gives a more precise prediction of the relative skew and the instant of the Tx window used by the time source neighbor. This results in a smaller adaptive listening window, and thus lower energy consumption.

6.1.4 Experimental Evaluation

To validate our prediction-based rejoining scheme, we implemented it on the TelosB, OpenMote, and GreenNet platforms [159, 14, 160], using the OpenWSN networking stack. We use the TSCH minimal schedule with 16 frequency channels and a slotframe of 101 timeslots, 15 ms each, to obtain a 1% duty cycle [161]. The EB interval can be calculated by adding a random value from the interval $[-3,48; +3,48]$ to a fixed 30 s offset. This approach lowers the probability of an EB collision when all network nodes use the same EB interval and timeslot for advertising. We compare our scheme with the proposals by Vogli et al. [157] and Duy et al. [158].

6.1.4.1 Node Join Latency

In the first experiment, we evaluated the join latency of our scheme. We use a single-hop setup with two nodes. The first node is configured as the network sink, while a second node is set up as a network leaf. At the beginning of the experiment, we wait until the leaf is synchronized with the sink. Once the initial synchronization occurs, we wait an additional 5 min for the leaf node to learn the relative skew to the sink node, i.e., its time source neighbor (see Phase 1 in the rejoining scheme). Once the leaf node establishes an initial relative drift estimation, the sink node stops all communication, effectively simulating a jamming attack. After a full KA interval, without communication, the leaf node enters the desynchronized state, and the resynchronization procedure starts. We keep the sink node silent for a total of 5 min. Following this delay, we instruct the sink to resume its normal behavior, marking the end of the DoS attack. Starting from this instant, we start measuring the time for the leaf node to resynchronize with the sink. In this experiment, the rejoin procedure becomes active directly after the node desynchronizes. During the entire 5 min jamming, the leaf node will try to resynchronize. Table 6.1 shows the average join latency for three different scenarios. We obtain the experimental results for our scheme after 35 runs on the OpenMote platform, and we compute the average join latency for the other schemes based on the theoretical expressions, experimentally validated by the authors in their respective papers [157, 158]. Below the results for the other proposals, we show the ratio with respect to our scheme.

TABLE 6.1: Average join latencies.

Join Scheme	**EB** (30 s)	**EB** (15 s)	**EB** (6 s)
Predictive Rejoin	7.51	5.9	5.25
Fast|Rapid Join [157, 158]	255 ×34	127.5 ×22	51 ×10
Rapid Join [158] with 4 EBs	63.75 ×8	31.88 ×5	12.75 ×2

We can see that our scheme significantly outperforms the other proposals in each scenario. The upper bound join latency for our scheme never surpasses the interval between two EBs as we will always capture every frame sent by the time source neighbor for a PDR (Packet Delivery Ratio) of 100% and a correct drift estimation. However, other frames such as RPL, KA, or application data frames will very often speed up the resynchronization process. Adding extra EB frames will therefore only slightly improve the speed of our scheme. The other proposals only rely on EBs for resynchronization and do not predict the transmission times of the time source neighbor. Fast Join by Vogli [157] sends one EB per node in the network. Rapid Join Scheme by Duy [158] has two modes: the first mode emulates Fast Join, the second mode lets every node send multiple EBs (corresponds to the scheme in the last row of Table 6.1). The latter mode obtains faster node rejoining, but consumes more energy from the network perspective as every node sends extra EBs. A typical energy cost for an EB, depending on its size and hardware platform, is 72.4 µJ with a battery at 2.2 V [158].

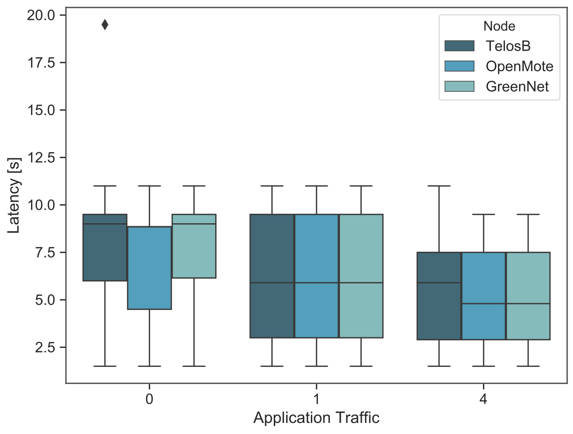


Figure 6.6: Rejoin latencies with respect to application traffic.

We further tested the influence of the application traffic on the performance off the predictive rejoin scheme. Figure 6.6 shows the synchronization speed for three different applicative traffic loads. In the first setup, there is no application compiled on top of the stack, and nodes only rely on EBs, RPL, and KA frames to resynchronize. In the second setup, the time source neighbor node transmits one additional data frame every EB period, and in the last scenario, the sink sends four extra data frames.

6.1.4.2 Adaptive Radio Reception Window

The arithmetic precision of our calculations plays an vital role in the accurate scheduling of the adaptive reception window. The OpenWSN code is written in a portable way so it can execute on a wide range of different platforms. We used integer arithmetic to compute the relative skew estimation. The usage of floating-point arithmetic is either not supported, or its cost too elevated on low-power platforms. We optimized our code to obtain a relative drift precision of 0.5 ppm. The experimental results in Table 6.2 were obtained using the same setup as described above, but we kept the leaf node desynchronized for 1 h. After 1 h, we measured the difference between the skew prediction and the real skew, using the logic analyzer running at 25 MHz. If the error on the skew prediction is smaller than the PGT/2 (1300 µs), a successful resynchronization is possible without an expanded reception window.

TABLE 6.2: Impact of the arithmetic precision on the drift prediction.

Platform	Predicted	Real	Prediction Error
TelosB	39.26	39.94	0.68
OpenMote	2.13	2.39	0.26
GreenNet	21.67	23.65	1.98

We already saw that temperature changes could affect the absolute clock drift of a device. We set up an experiment where a node is exposed to fluctuating temperature, ranging from 10 °C to 35 °C. We performed the experiment twice. At first, the desynchronized node did not have a temperature sensor, and therefore, it could not compensate for additional temperature-based drift. Secondly, the desynchronized is equipped with a temperature sensor. It uses the temperature measurements to compensate for the drift in real-time. In both experiments, the sink was kept at room temperature (25 °C). The leaf measured its temperature, and the relative clock drift every minute and sent the results to the sink as plotted Figure 6.7.

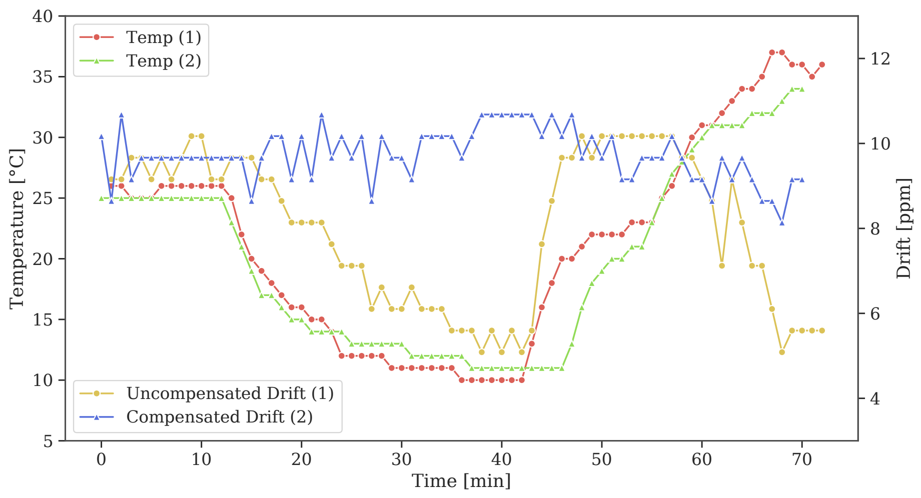
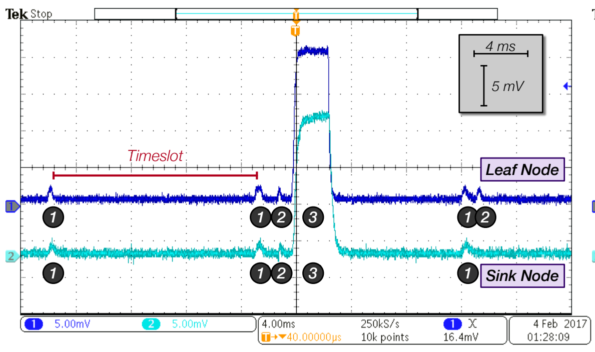


Figure 6.7: Temperature and drift in time for compensated and uncompensated devices.

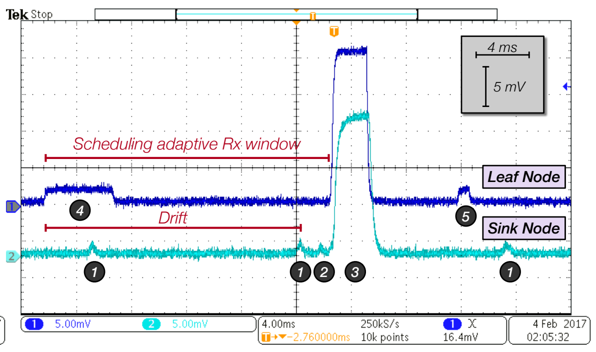
To compensate for the drift in software, we used the relation $-0.035\ \text{ppm}/°\text{C}^2 \times (T - T_0)^2$ [162]. When the nodes started communicating, the temperature was 25 °C, and both the clocks operated at their nominal temperature. The relative drift measured during the first 10 min of the experiment, at a constant temperature, was approximately 9.5 ppm for both node pairs. After 10 min, we gradually changed the ambient temperature of the leaf node. The node, without a temperature sensor, could not compensate, and the relative drift for node pair ① dropped drastically. Notice that Phase 1 of our scheme accounts for the observed 9.5 ppm, and the adaptive listening window must compensate for the additional drift. Node pair ① experienced a drift change of 5 ppm caused by the temperature changes (see Figure 6.7), which amounts to a clock skew of 18 ms after 1 h. Because we cannot predict how the drift changes, we need to expand the listening window in both directions: $(2 \cdot 18\,\text{ms}) + \text{PGT} = 38.6\,\text{ms}$. Node pair ② only needs to compensate for a change of 1.5 ppm, which amounts to a smaller Rx window of 13.4 ms.

6.1.4.3 Energy Consumption

Figure 6.8 presents the measured current consumption for a synchronized pair of TelosB nodes (a) and when the predictive rejoining scheme is active (b). We use the simplified energy model by Vilajosana et al. [155] to obtain the overall energy consumption from the current measurements. We split the energy consumption of the platform into three states: CPU idle, CPU active and radio active. The different states draw 0.66 mJ, 2.86 mJ, and 39.6 mJ, respectively.



(a)



(b)

Figure 6.8: Current measurements with a Tektronix MD03012 oscilloscope of the TelosB platform (battery at 2.2 V).

In Figure 6.8, ❶ marks the start of a TSCH timeslot. The CPU executes code to check if a frame needs to be sent. The frame gets prepared for transmission in state ❷. In ❸, we observe the activation of the radio. Figure 6.8b starts with the desynchronized node, detecting a potential incoming transmission from the time source neighbor and subsequently calculating the relative clock skew offset (❹). It reschedules the adaptive listening window in order to capture an incoming frame. Resynchronization fails in Figure 6.8b; thus in ❺, the CPU continues with the same procedure. The total used energy in a timeslot is calculated as the sum of the energy spent in each state (Idle, Active, Radio), where T, I and V_b represent the time spent in a state, the current drawn in a state, and the battery voltage, respectively:

$$E_{\text{tot}} = (T_{\text{Idle}} I_{\text{Idle}} + T_{\text{Active}} I_{\text{Active}} + T_{\text{Radio}} I_{\text{Radio}}) \cdot V_b \tag{6.4}$$

An “idle slot” is a slot during which only the CPU is responsible for energy consumption. Slots during which the radio is active are “radio slots”. Fast and Rapid Rejoin Schemes [157, 158] do not use prediction to localize the incoming EB. Therefore, they need to keep their radio active all the time. The multi-beacon option of Rapid Join also requires additional network energy because it sends multiple EBs in the same EB interval. The sink uses a duty cycle of 1%. Since our scheme only wakes up the desynchronized node when it expects an incoming transmission, the overall duty cycle remains very low. It uses one double-sized active slot to reschedule the listening window and 99 idle slots per slotframe (see Figure 6.8b). This approach facilitates the rescheduling of the listening window when it is located in between two timeslots. The energy drawn for 99 idle slots and one double-sized active slot amounts to 1.29 mJ, while the energy for 101 active slots is 59.99 mJ. Table 6.3 shows the computed energy for the measured join latencies presented in Table 6.1. Below the used energy for the other proposals, we show the ratio with respect to our scheme.

TABLE 6.3: Estimated energy consumption of the join schemes (mJ).

Join Scheme	**EB** (30 s)	**EB** (15 s)	**EB** (6 s)
Predictive Rejoin	6.45	5.15	4.51
Fast|Rapid Join [157, 158]	15.3e3 × 2372	7.65e3 × 1485	3.06e3 × 678
Rapid Join [158] (4 EBs)	3.82e3 × 592	1.91e3 × 370	0.76e3 × 168

6.2 Thermal Covert Channel in BLE Networks

6.2.1 A Primer on Bluetooth Low Energy

BLE was first introduced in Bluetooth Core specification v4.0 [163]. In contrast to classic Bluetooth, which can transfer a lot of data, BLE is optimized for low-power applications that only periodically need to communicate. Similarly to TSCH, BLE allows devices to go into a sleep mode between data transmissions, effectively lowering the overall power consumption of the entire protocol. The Bluetooth communication stack roughly splits into three parts [164]. At the bottom, we can find the controller part. The controller contains the lowest two layers of the Bluetooth stack: the physical layer and link-layer. The controller is typically implemented in the firmware of the Bluetooth radio chip. The host encompasses the higher levels of the Bluetooth stack, such as logical link control, attribute protocol, generic access profile, etc. The host is often a part of the operating system of the device that uses Bluetooth networking. On top of the host, we can find the Bluetooth applications. These live traditionally in userspace. The BLE link-layer design philosophy is similar to that of IEEE 802.15.4E. BLE devices share a global network clock to synchronize the devices. BLE devices follow a strict schedule to minimize energy consumption and interference issues. The schedule pseudorandomly hops through 40 different frequency channels. Three of those channels are used purely for advertisement, while the remaining transport data.

Although in the most recent update of the Bluetooth Core specification [165], version 5.0, changes were introduced to support multi-hop networking like TSCH, BLE is most frequently used to establish a P2P connection between two devices. The devices either fulfill the *slave* role or *master* role. Typically slave devices are very simple, e.g. a temperature sensor or light bulb that generates some application data. The master is a more powerful device, for example, a smartphone. The master device discovers a nearby slave device by listening for advertisement frames. The slaves periodically emit advertisements during an *advertisement event*. When a master device has captured an advertisement frame, it can start a negotiation phase by responding with a *connection request frame*. The connection setup establishes the parameters for the connection. The connection request frame of the master device contains, among other a *connection interval* and a pseudorandom hopping sequence. The connection interval indicates how often the master and slave will wake-up to exchange data, known as a *connection event*, and the pseudorandom sequence describes the order in which the frequency hopping takes place. Values for the connection interval range from 7.5 msec to 4 sec and are a multiple of 1.25 msec. The start of a connection event is called an *anchor point*. The master device initiates a connection event by sending a frame to the slave, see Figure 6.9. This frame can contain application data, but more often, it merely acts as a poll request, asking the slave to respond with its application data. The slave answers either with its data or with an empty frame in case it has no data to transmit. Both devices keep following this schedule until the master disconnects.

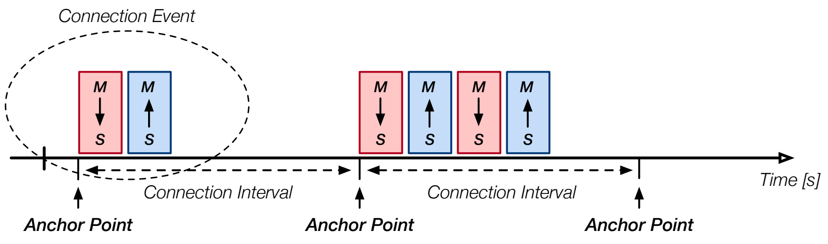


Figure 6.9: Bluetooth Low Energy in connected mode. During a connection event multiple frames can be exchanged.

The master and slave can also negotiate a *slave latency* parameter. This parameter defines the number of consecutive connection events that the slave device can ignore, allowing the slave to spend more time in sleep mode to conserve energy.

6.2.1.1 The BLE Synchronization Mechanism

The Bluetooth Core specification specifies two different clock accuracies. During a connection event or advertising event, the devices use the active clock accuracy, with a drift less than or equal to ±50 ppm. When in sleep mode, the devices use the sleep clock accuracy, which can have a maximal drift of ±500 ppm.

Because of the potentially significant drift and the resulting clock skew, there is an uncertainty in the slave device of the exact timing of the next anchor point. BLE uses a highly similar PGT mechanism as TSCH to ensure frame reception when experiencing clock skew. When the slave receives a frame from the master, the slave updates its anchor point [165] and always responds with a (potentially empty) frame unless the slave latency parameter is not 0. The slave is required to resynchronize its clock to its master at each connection event to avoid accumulating a large clock skew. To ensure that the slave wakes up in time to receive the first frame of the master during a connection event, it estimates the position of the next anchor point, taking into account the possible clock drift. During the connection setup, the master can indicate its sleep clock accuracy. The slave can use the master sleep clock accuracy and its local sleep clock accuracy to more precisely estimate the relative clock skew between both devices and ,thus, the positions of the next anchor points. In contrast to standard TSCH which uses a fixed size for the PGT, BLE devices calculate a window widening parameter according to the following formula,

$$\delta_{window} = \Delta t \Big[(D_m + D_s)/1 \times 10^6 \Big] \tag{6.5}$$

The slave starts listening for a frame from the master a full window widening value before the expected anchor point, see Figure 6.10. The Δt parameter indicates the elapsed time since the last anchor point (and resynchronization of the slave to the master). The values D_m and D_s are the master's and slave's clock inaccuracies, respectively.

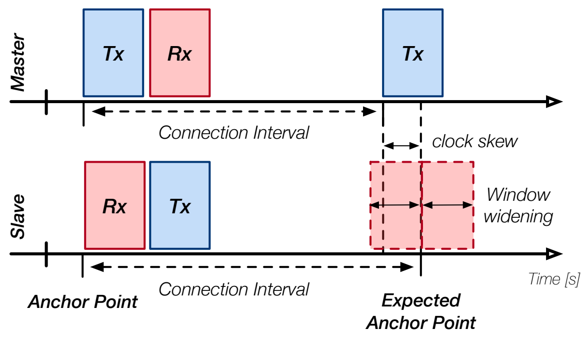


Figure 6.10: The slave uses an adaptive PGT to compensate clock skew and guarantee reception of the master frame.

6.2.2 Attack Scenario

In a typical attack scenario, a user gets tricked into installing a malicious application. The malicious application could be impersonating a legitimate application, or the legitimate application itself was compromised due to a supply chain attack [166]. The malicious application has access to sensitive data such as passwords or encryption keys (e.g., a password manager). However, the application cannot use any networking functionalities, as the user did not grant the application those privileges. It has no direct way to extract sensitive data from the victim device. Instead, the malicious application modulates the sensitive data on top of the frames exchanged by long-lived BLE connections of legitimate applications. Examples of such long-lived connections are a smartwatch connected to a smartphone or a smartphone connected to wireless headphones or smart loudspeakers.

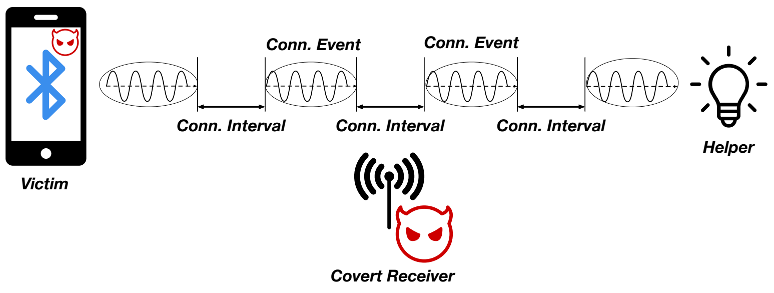


Figure 6.11: The covert receiver measures the induced clock skew between the connection intervals.

Figure 6.11 depicts the attack scenario. There are three distinct entities: the victim device, a neutral helper device, and the covert receiver, which is fully controlled by the attacker. By scheduling well-timed CPU intensive calculations, the malicious application can control the heat emission of the CPU. These emissions directly influence the frequency of crystal oscillator. The malicious application uses the impact of the $E(t)$ in equation 5.2 to cause a varying, measurable clock skew. Bits can be encoded under the form of these clock skew variations. The victim has to take on the role of master in the BLE connection as this forces the helper device to stay synchronized to the victim even if the malicious application triggers a significant clock skew. The covert receiver can be positioned anywhere in the BLE transmission range and acts as a hidden slave device. There are no physical or logical connections between the victim and the covert receiver. The covert receiver can even be located in an adjacent room. We assume that the covert receiver can measure with sufficient precision the intervals between the consecutive BLE transmissions.

6.2.3 Covert Communication Protocol

Next, we describe the communication protocol between the victim device and the covert receiver. The communication channel is unidirectional; thus, we allow for a significant overhead in the protocol dedicated to error correction codes as there is no way the covert receiver can ask for the retransmission of data.

6.2.3.1 Encoding and Decoding

In our communication protocol, we use a simple On-Off Keying technique to transmit data. A substantial variation in the clock skew is triggered, to encode a `1`-bit, by heating the oscillator. A `0`-bit is encoded as the absence of a varying clock skew. We used several threads of cryptographic calculations to generate a high CPU load. The malicious application must keep the temperature steady for a while, ensuring that the heat can propagate throughout the device and reach the oscillator. When the malicious application releases the CPU load, the device and crystal oscillator cool down, and the clock skew returns to its previous value. During this process, the covert receiver is measuring the elapsing connection intervals (CI_j) by creating a timestamp (T_i) at the reception of the first packet of the master at the start of a connection event.

$$\mathrm{CI}_j = (T_i - T_{i-1}) \tag{6.6}$$

By subtracting the subsequent connection intervals, the attacker calculates the clock skew for the connection interval.

$$\Delta C_{\text{skew}} = \Delta\mathrm{CI} = \mathrm{CI}_j - \mathrm{CI}_{j-1} \tag{6.7}$$

The attacker then uses a low-pass filter to extract the trends in the clock skew behavior, $S(t)$. By calculating the derivative, $\frac{dS}{dt}$, over the filtered signal, the attacker can detect sudden important variations in the clock skew. When the absolute value of the clock skew variation is higher than a predefined threshold value V_{th} a `1`-bit is detected; otherwise, the filtered signal is decoded as a `0`-bit.

$$\text{Bit string}(t) = \begin{cases} 1 & |\frac{dS}{dt}| \geq V_{th} \\ 0 & |\frac{dS}{dt}| < V_{th} \end{cases} \tag{6.8}$$

The threshold value V_{th} is discovered by calculating the standard deviation (σ) over $\frac{dS}{dt}$. A scaling factor α is experimentally derived.

$$V_{th} = \sigma\Big(\frac{dS}{dt}\Big) * \alpha \tag{6.9}$$

A preamble prepends each message. It marks the start of a data transmission. A preamble of a single `1`-bit denotes the start of a transmission. We must choose a preamble starting with a `1`-bit because the covert receiver can detect only a change in the clock skew. A `0`-bit is encoded as the absence of varying clock skew and can, therefore, not be identified by the decoder.

6.2.3.2 Calibration of the channel

Before the actual transmission of data, we have to calibrate the channel. The calibration values must be given to the malicious application before installation on the victim. We can derive the parameters on devices identical to the victim platform. Since the attack relies on hardware characteristics, the parameters must be derived for each platform individually. Initially, the base temperature of the device, T_b, must be measured. The base temperature corresponds to the temperature of the CPU when it is idle. Secondly, the target temperature, T_t, must be defined. Thirdly, we must set the time interval, ΔT, for which the malicious application must hold the CPU temperature at T_t when encoding a binary 1. Finally, the malicious application needs to know the cooldown time, T_c. This variable describes the time needed by the system to return to the base temperature T_b and for the clock skew to settle on its previous value.

6.2.3.3 Error correction and data whitening

The malicious application uses two techniques to provide reliable transmission: error correction codes and data whitening. A possible candidate for the error correction codes are the BCH codes [167]. BCH codes can correct multiple errors, depending on the construction of the code word. For example, the BCH$(63,31)$ code adds maximally 31 check bits to a data payload of 32 bits and allows the covert receiver to correct up to 5 bitflips in the received codeword.

Data whitening scrambles the data before transmission to prevent long sequences of ones or zeros. Because each binary 1 is encoded as an increase in CPU temperature, there exists a risk that several consecutive encoded 1-bits could saturate the sensitivity of the oscillator to temperature changes. As remnant heat builds up in the system, the device does not have enough time to cool down; hence, we can no longer precisely control the skew. We borrow the data whitening technique described in the Bluetooth Core specification [165]. A simple LFSR is used to generate a pseudorandom bit string. This bit string is then XORed with the original data to obtain a pseudorandom bit sequence, ready for transmission.

6.2.4 Experimental Evaluation

6.2.4.1 Setup

We discuss our implementation of the attack on three hardware platforms: Raspberry Pi 3B, Motorola X 2014, and an iPhone 5s. Each experiment we performed, followed the attack scenario shown in Figure 6.11. The experiments took place in an office at room temperature. The role of the helper device was fulfilled by a simple BLE-enabled light bulb [168]. As a covert receiver, we used an Ubertooth One [154] connected to a PC. The distance between the different devices was approximately 2 m.

6.2.4.2 Clock Skew Tracking

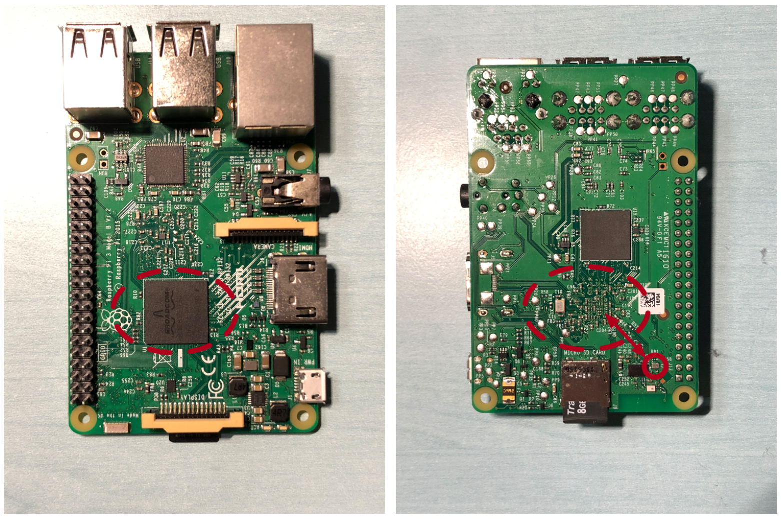
The Ubertooth is capable of sniffing a specific BLE connection when it can overhear the initial connection setup. The handshake negotiates all the parameters necessary to calculate anchor points and to derive the frequency hopping pattern. The Ubertooth firmware registers timestamps at the reception of every frame. The timestamps are generated by the Ubertooth's CPU [169] by reading the current timer value at the reception of radio DMA (Direct Memory Access) interrupt. The timer runs with a period of 50 MHz. The Ubertooth sends the timestamps to the computer. We only keep the timestamps of the first frame emitted by the victim/master device at every connection event. The difference between two consecutive timestamps corresponds to the connection interval of the victim, see equation 6.6. The connection interval remains stable when the oscillator operates at a fixed temperature (a constant clock skew between the victim and the helper/covert receiver). In this scenario, all the terms in equation 5.2 are stable, resulting in a stable clock skew. If the oscillator experiences a fluctuating ambient temperature, the $E(t)$ term causes a varying clock skew, and the Ubertooth measures the resulting varying connection intervals.

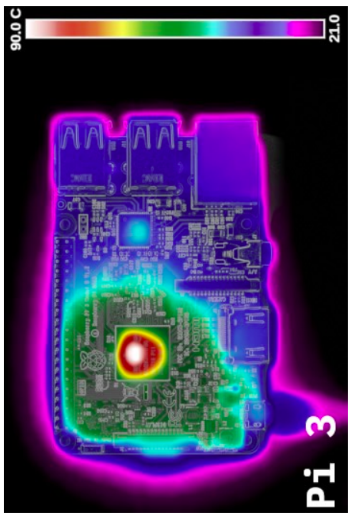
6.2.4.3 Victim configuration

We used three different platforms with different software stacks to test feasibility of our attack: a Raspberry Pi 3B [170] running Arch Linux, a Motorola X 2014 running Lineage OS 14.1, and an iPhone 5s with iOS 11.4.1. Before the start of the experiment, during the reconnaissance phase, the physical characteristics of the victim device are investigated. We try to register the base temperature of each device, the temperature of a device when idle, the amount of heat that was effectively generated, and how fast the heat propagated throughout the device and finally how quickly this heat dissipated. We also verified the size of the clock skew variations due to the increasing or decreasing temperature. Significant variations point towards a direct thermal path between the CPU and the oscillator or the use of an uncompensated crystal oscillator, while small variations might be caused by the CPU and oscillator being thermally isolated or the use of a TCXO (Temperature-Compensated Crystal Oscillator).

6.2.4.4 Raspberry Pi 3B

Reconnaissance phase The Raspberry Pi 3B board uses a quad-core 1.2 GHz Broadcom BCM2835 64-bit CPU and a BCM43438 chipset for Wi-Fi and BLE connectivity [171]. Figure 6.12a shows the positions of the CPU and the wireless chipset with crystal oscillator on the board. The distance between the CPU and the oscillator is approximately 1.5 cm. Both the CPU and BLE chip are attached to the same board, allowing for a direct thermal path, see Figure 6.12b.


(a) The position of the CPU and the crystal oscillator on the PCB.



(b) Thermal image (source: [172]).

Figure 6.12: The Raspberry Pi 3B platform on which we used to test the covert channel.

After inspection of the board and the datasheet of the BCM43438, we suspect that the board uses an uncompensated crystal oscillator, although verification of this assumption is not possible as the majority of the Raspberry Pi schematics are not publicly available. The board also exposes one internal CPU temperature sensor. In the first instance, we analyzed how well the heat of the CPU propagates throughout the board. With a thermocouple [173], we track in real-time the temperature of the oscillator as we vary the load on the CPU. We use this information to derive the values to calibrate the covert channel.

TABLE 6.4: Calibration values for the covert channel: RPI-3B

Constant	Value	Comments
T_b	39 °C	Temperature for idle CPU
T_t	47 °C	$T_t + 1$ for every consecutive ‘1’ bit
Δt_h	10 s	
Δt_c	65 s	Cool down time: $T_t \rightarrow T_b$

Figure 6.12b shows the heat spread from the CPU throughout the board. The article [172] discusses the changes in heat propagation between the Raspberry Pi 3B and the Raspberry Pi 3B+. Although the newer model has a better heat spread, the heat propagation is sufficiently wide to reach the oscillator. Measurements with thermocouple [173] indicate 10 °C to 15 °C difference between the temperature measured at the CPU package and the oscillator.

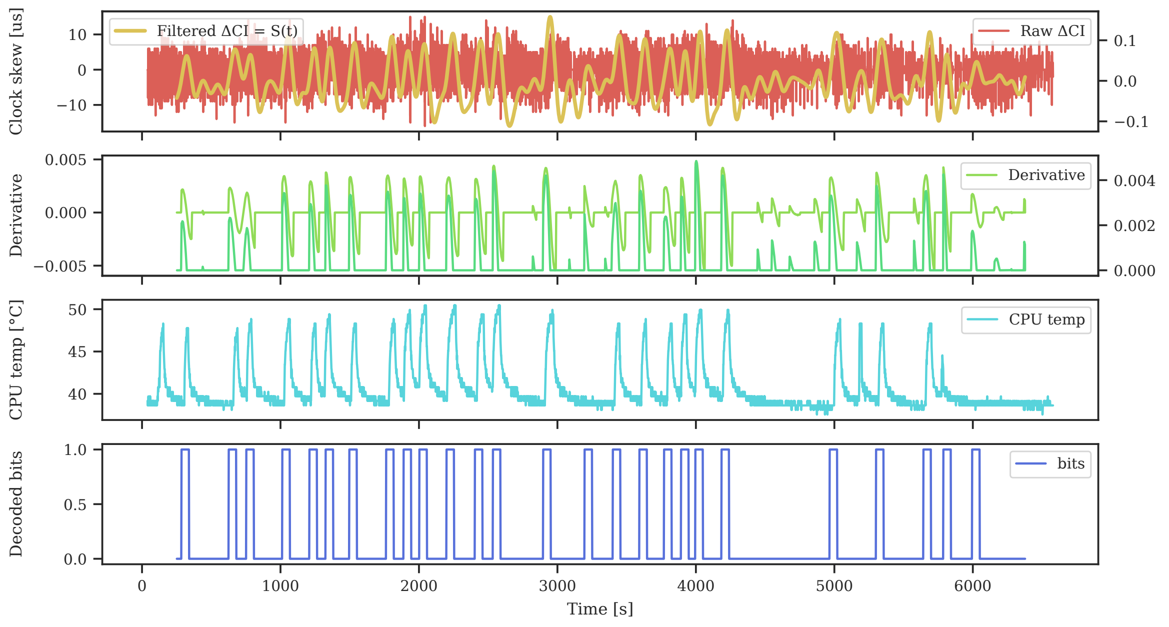


Figure 6.13: Implementation of the covert channel on the Raspberry Pi 3B. From top to bottom: raw and filtered skew, the derivative of the filtered skew, the CPU temperature and the decoded bits.

Covert Channel Performance With the characteristics of the victim known, the malicious software on the Raspberry Pi can set up a covert channel. Figure 6.13 shows a covert transmission of approximately 2 h. The raw skew data is directly measured by the Ubertooth and logged by the computer. By filtering the signal, we extract the clock skew trend from the original noisy signal. The second graph in Figure 6.13 shows the derivative of the filtered signal. We also plot the derivative a second time with the negative values set to 0. We observe that the peaks in CPU temperature match well with clock skew variations, shown by the peaks in the derivated signal. Finally, we use equation 6.8 and equation 6.9 to extract the bits, shown in the lowest graph of the figure. The throughput of the channel is approximately 38 bits per hour.

To improve the throughput of the channel, we needed to accelerate the cooldown phase of the covert channel. In a second experiment, we assumed that the malicious software also controlled a fan. This fan could be activated to help cooling the CPU and the oscillator. We used the same base temperature and target temperature, but the cooldown time, ΔT_c, was reduced to 30 s. The results are shown in Figure 6.14. The throughput of the new cooled channel is approximately 45 bits per hour.

6.2.4.5 Motorola X 2014

Reconnaissance phase The Motorola X 2014 uses a Qualcomm Snapdragon 801 8974-AC CPU [174] and a Qualcomm WCN3680 802.11ac Combo Wi-Fi/Bluetooth/FM chipset [175]. The wireless chipset of the phone uses, highly probable, a TCXO [176]. Because we have no access to the actual schematics of the phone, we cannot verify this assumption. The phone's hardware has 15 different temperature sensors. Lineage OS makes the sensor values readable under `/sys/class/thermal/thermal_zone[1-15]/temp` in the filesystem. Although many of the sensor names are cryptic, some of them can easily be attributed to hardware components of the phone, e.g., `chg_temp` exposes the temperature sensor of the battery. After carefully testing the temperature values, returned by the sensors during CPU intensive calculations, we pick a sensor for our experiment, which corresponds well to the measured clock skew.

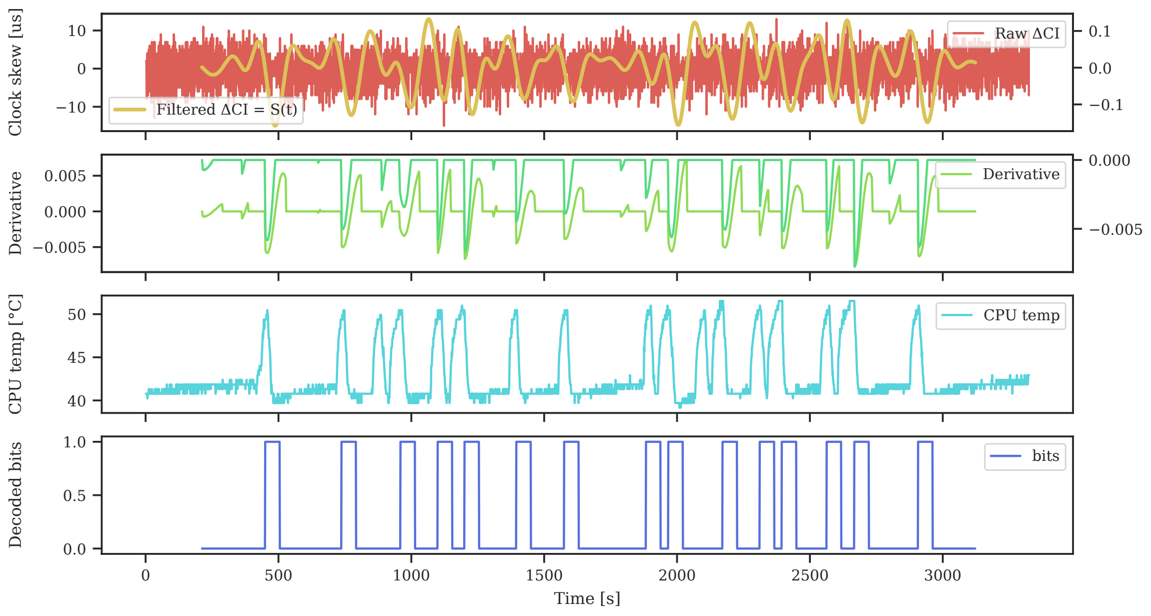


Figure 6.14: Cooled covert channel on the Raspberry Pi. We observe that the additional cooling allows for a faster return to the T_b temperature. Similarly the clock skew recovers faster.

TABLE 6.5: Calibration values for the covert channel Motorola

Constant	Value	Comments
T_b	27 °C	Temperature for idle CPU
T_t	32 °C	
ΔT_h	1 s	
ΔT_c	±400 s	Cool down time: $T_t \rightarrow T_b$

Covert Channel Performance The Motorola X 2014 uses a Qualcomm Snapdragon 801 8974-AC CPU [174] and a Qualcomm WCN3680 802.11ac Combo Wi-Fi/Bluetooth/FM chipset [175]. The wireless chipset of the phone uses, highly probable, a TCXO. A full teardown of a first-generation Motorola X shows the approximate locations of the CPU and the BLE chip [177]. They seem to be far apart. A teardown of the second-generation Motorola (used in our experiments) also shows metal heatsinks covering the individual chips. This hardware layout would allow the phone to evacuate the majority of the generated heat before it reaches the BLE chip. If the chip additionally uses a TCXO, the effects of the CPU heat on the induced clock skew could be negligible. Because we have no access to the actual schematics of the phone, we cannot verify these assumptions.

Compared to the experiment with the Raspberry Pi 3B, the throughput of the covert channel on the phone is much smaller, see Figure 6.12a. The throughput is approximately 6 to 8 bits per hour. Several reasons can be found that explain this performance drop. The heat-up time and cooldown time are much larger. The slow increase in temperature is probably due to the use of metal heat sinks over the different chips, preventing the heat of the CPU from spreading to the other components on the motherboard. Once the oscillator has reached its target temperature, we cancel the CPU load. In contrast to the Raspberry Pi, the phone's hardware is protected by its exterior casing, preventing the build-up heat from dissipating quickly. The usage of the TCXO also impedes strong clock skew variations which make the detection of peaks in the filtered signal harder.

6.2.4.6 iPhone 5s

Reconnaissance phase The iPhone 5s uses Apple's A7 chip, a custom ARMv8 1.3 GHz dual-core processor [178]. The Wi-Fi/Bluetooth chipset is based on Broadcom's BCM4334 [179]. The BCM4334 is similarly

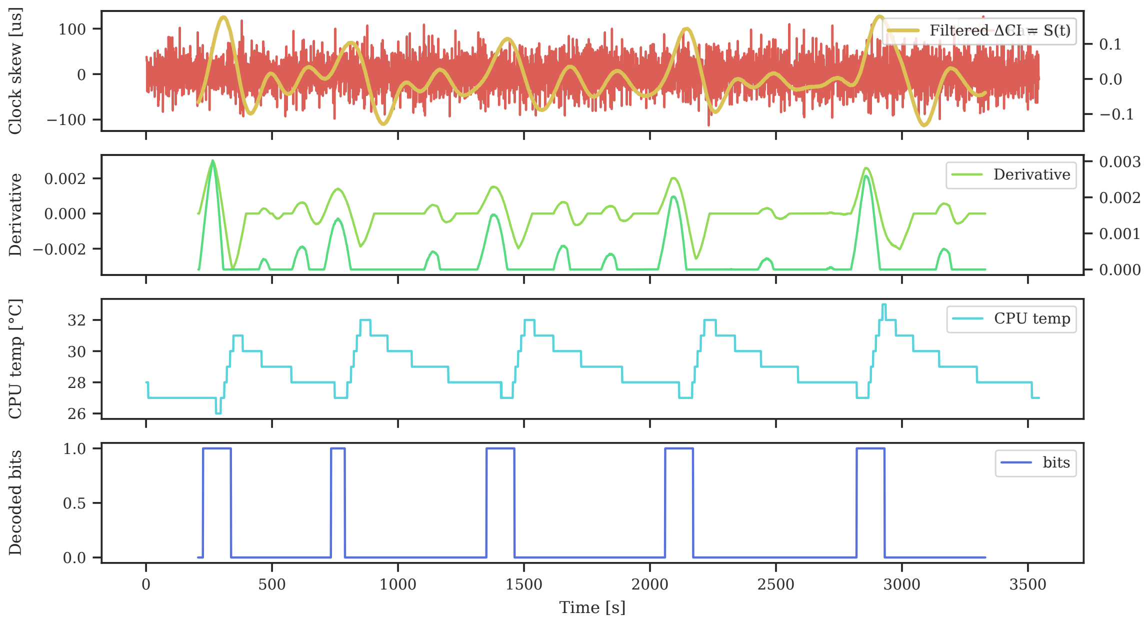


Figure 6.15: Covert channel on the Motorola X 2014. Compared to the filter clock skew on the Raspberry Pi platform, the clock skew variations are much more suppressed.

to the BCM43438 capable of using a TCXO to compensate temperature variations. The iPhone's OS does not expose any internal temperature sensors. A teardown of the iPhone 5s reveals the positions of the A7 and BLE chip [178]. Compared to Motorola's logic board, the iPhone's BLE chip seems to be in proximity to the A7 processor.

TABLE 6.6: Calibration values for the covert channel: iPhone 5s

Constant	Value	Comments
T_b	-	No access to temperature sensors
T_t	-	No access to temperature sensors
Δt_h	± 60 s	
Δt_c	± 70 s	Cool down time: $T_t \rightarrow T_b$

Covert Channel Performance The iPhone 5s has a much higher throughput compared to the Motorola. It is approximately 32 bits per hour. The influence of the CPU's heat on the oscillator is rapidly noticeable. Additionally, the clock skew returns fast to its base value when we stop heating the CPU. We cannot say with certainty why the iPhone's oscillator is more susceptible to the temperature changes, but we guess that the close vicinity of the BLE chip to the A7 application processor allows for a direct thermal path. The phone's heat sink also seems to be more efficient, which allows for fast heat dissipation. A metal band additionally surrounds the iPhone's exterior casing, which functions as a large heatsink.

6.2.5 Performance Discussion

The bandwidth of our covert channel depends on several factors, including the time required to heat or cool down the oscillator to a specific temperature, and the error rate in the transmission. Both parameters, in turn, depend on the hardware architecture of the victim and the distance between the heat source and crystal oscillator. Another critical factor is the sensitivity of the crystal to sudden temperature changes. On high-end hardware, such as smartphones, the clock skew variations are suppressed due to the use of a TCXO. The latter hurts the throughput of the channel. It makes the decoding of the signal more challenging and error-prone. Table 6.7 summarizes the different performance characteristics obtained.

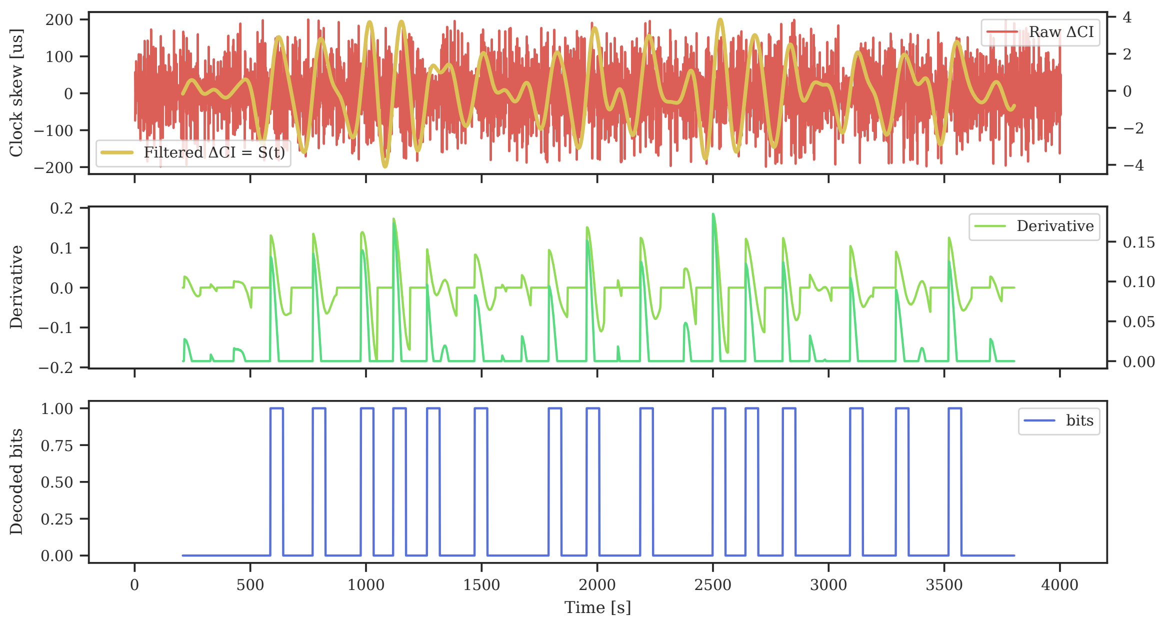


Figure 6.16: Covert channel on the iPhone 5s. No temperature information was easily available, so the heat graph is omitted.

TABLE 6.7: Error rate on different hardware architectures

Platform	Throughput (bph)	Bits sent	Errors	Error rate
RPi 3B	38	61	3	5%
RPi 3B (cooled)	45	33	2	6%
Motorola X	8	6	0	< 1%
iPhone 5s	32	27	0	< 1%

The calibration phase, prior to the attack, helps to improve the throughput and accuracy of the channel. Our experiments showed that the exact calibration values are not always available, e.g., on the iPhone, no base or target temperature could be discovered. In this scenario, we experimentally derived and fixed the values for the heat-up time and cooldown time. By adding a safety margin to these values, we can try to compensate situations where the base temperature of the device differs from the situation under which the experimental values were established. To improve throughput even further, we can try to speed up the heating up or cooling down phase. We showed that with the addition of a cooling fan to the control of the malicious application, we could improve the bitrate. Additionally, throughput could be improved by using multi-level encoding techniques if the CPU heat emissions can control the clock skew with acceptable precision, and the clock skew variations are sufficiently large.

Compared to the work in [180], our thermal covert channel is slower. Matsi et al. directly use the CPU temperature to establish the covert channel, which makes it more reactive to changes in CPU load. Our thermal covert channel must indirectly be measured through the effects on the clock skew. The drawback of the work in [180] is the range of the channel. Covert communication is limited to applications running on the same physical machine. The work presented in Bitwhisper [181] is capable of extending the range of the thermal covert channel to 40 cm. Because of the air-gapped constraints, they cannot sniff wireless transmissions, and therefore the range stays limited. Our approach has a similar throughput as Bitwhisper, but the range of the covert channel extends to the full transmission range of BLE. There is no need for a LoS which improves on the attack's stealthiness. Moreover, the covert receiver does not need to interact with the victim device. It suffices to sniff the legitimate traffic passively.

Conclusion

In this chapter, we analyzed two vulnerabilities intrinsic to time-synchronized link-layer protocols. Both vulnerabilities are a clear example of how the strong optimization of networking protocols can introduce new security problems. The first vulnerability relies on the fact that the initial cost of network joining is highly disproportionate compared to the cost of the regular link layer operation. Contrary to slave advertisement in BLE, TSCH does not limit the network advertisement to a few channels. The already scarce EB transmissions are spread out over multiple channels, making it significantly more costly, both in terms of join latency and energy consumption, to overhear an EB. Since the network synchronization relies on frequent message exchanges, an attacker can easily interfere with the method without the need for advanced equipment, deliberately triggering the expensive rejoin procedure. We were able to mitigate this vulnerability by sharply reducing the cost of network rejoining. At the same time, the overall join latency is minimized. Resynchronizing nodes can use any frame originating from their time source neighbor. Rejoining a network to which nodes were previously synchronized, does no longer require disproportionate power and time compared to the regular network operation.

The second vulnerability exploits the highly predictive behavior of time-synchronized networks. Due to strict synchronization, an external entity can easily predict the time instances where communication will occur. If an attacker can to modify this behavior freely, the comparison between the expected behavior and the measured behavior can be used as a means to encode bits. The initial data obligatory frame from the master at the start of a connection provides the perfect anchor point to measure any offsets to communication protocol. Since the BLE slaves are obligated to adapt to any drift changes, the normal conventional of the protocol is not compromised. Note that this attack is equally possible on TSCH-enabled hardware since its schedule also provides the means to predict communication instances. However, allocated slots in the TSCH schedule merely provide the possibility for communication. If the transmitting node has no pending frames, the slot will go unused. We also showed that the attack is not limited to devices with a rather simple hardware layout. Both on the Raspberry Pi and the iPhone, the covert channel achieved a respectable throughput. In future work, the channel could be tested on even larger systems, such as PCs. They provide an additional challenge since these systems can regulate their temperature through the use of fans.

Chapter 7

Performance of Transport Layer Security over IEEE 802.15.4E Networks

Contents

Introduction

IoT applications can protect data in transit with either transport layer security or object security. The OSCORE protocol, which implements the object security paradigm for use with CoAP, has many advantages compared to DTLS, the designated security protocol for the IoT [82]. The CORE working group designed OSCORE from scratch, with all the best security practices taken into account. It does not need to provide backward compatibility with older versions of the protocol. OSCORE also leverages newly designed encoding mechanisms, such as COSE and CBOR, optimized for low complexity and code footprint. In contrast to the transport layer security protocols, it supports several indispensable features such as secure group communication, application-level translation proxies, and caching of encrypted messages. These advantages will prove invaluable for securing the constrained IoT.

Nonetheless, the IoT revolution is happening right now. The IETF ratified the OSCORE RFC only recently, in July of 2019, but many supporting protocols are not yet ready. The OSCORE specification only describes how to transform a CoAP message into an OSCORE message. Mechanisms for secure key establishment and OSCORE endpoint authentication are still under development at the IETF. Besides, it will take some time until the new IoT protocols become widely adopted.

IoT product designers are therefore falling back to DTLS. DTLS is a mature protocol which enjoys extensive support. Several state-of-the-art software libraries provide open-source, vetted implementations that can be used out-of-the-box in IoT projects. Libraries such as mbed TLS [51] and wolfSSL [182] are highly configurable and allow for minimalistic implementations, reducing the overall code size. However, DTLS is poorly optimized for usage in highly constrained devices. The most critical an expensive part of the protocol is the handshake phase. It establishes a shared master secret and subsequently derives the session keys. It requires numerous computationally expensive cryptographic operations and large RAM buffers to store the incoming and outgoing DTLS datagrams. Since the underlying transport layer, UDP, provides no transmission reliability, the DTLS handshake reimplements multiple aspects of TCP. More precisely, it adds sequence numbers to the DTLS headers and employs a naive retransmission mechanism with back-off. DTLS must also performs transport-layer fragmentation to prevent oversized datagrams. Developers can parametrize these features up to some extent to obtain better performance in constrained environments, but they must make multiple assumptions about the capabilities of the other endpoint. In real-life deployments the characteristics of the other DTLS endpoint are not always known.

While DTLS and TLS have approximatively the same overhead, TLS requires a reliable transport layer protocol, i.e., TCP. TCP has long been considered too heavyweight as a transport protocol for the IoT. However, due to the concourse of several circumstances, discussed in Chapter 3, a resurrection of TCP in the context of the IoT is possible. The many options and customizations of TCP allow us to squeeze out significant performance gains in scenario's where the overall reliability of the communication link is questionable. The above motivated us to perform an in-depth study of the TLS and DTLS handshake over low-power lossy networks. We analyzed several factors of the handshake phase in both protocols. We investigated how both protocols handle the reliability and fragmentation of the handshake packets. Next, we studied handshake latency and the total number of transmitted link-layer frames and bytes.

7.1 A Detailed Overview of the (D)TLS 1.2 Handshake

The TLS and DTLS 1.2 handshake protocols are responsible for negotiating a session that contains: a session identifier, an optional peer certificate (X.509v3), a compression method, a cipher suite, a master secret and finally a boolean value stating if the session is resumable. Recall that the handshake messages are carried in the body of record layer messages, the lowest layer of the (D)TLS protocol. The RFCs also use the term *fragment* to refer to the body of a record layer message [183, 109]. To avoid confusion with messages that are the explicit result of a fragmentation mechanism, possibly on other layers of the networking stack, we will consistently use the term *record fragment* to denote the body of a record layer message. The (D)TLS session negotiation happens in different phases. We illustrated the full handshake protocol in Figure 7.1. Initially, the peers exchange hello messages (*ClientHello* and *ServerHello*), agreeing on a compression method and cryptographic algorithms. The messages also contain random values and state the support for session resumption (the client uses an empty session ID if no session resumption is wanted or supported). The client can also request additional functionalities of the server by including extensions in its hello message. The client can use, for example, the `signature_algorithm` extension to indicate to the server which signature and hash algorithm pairs it can use when computing digital signatures. If the client wishes to use elliptic curve cryptography, it must include the `supported_groups` and the `ec_points_format` extensions to indicate which elliptic curves it supports and how the elliptic curve points will be exchanged, either compressed or uncompressed.

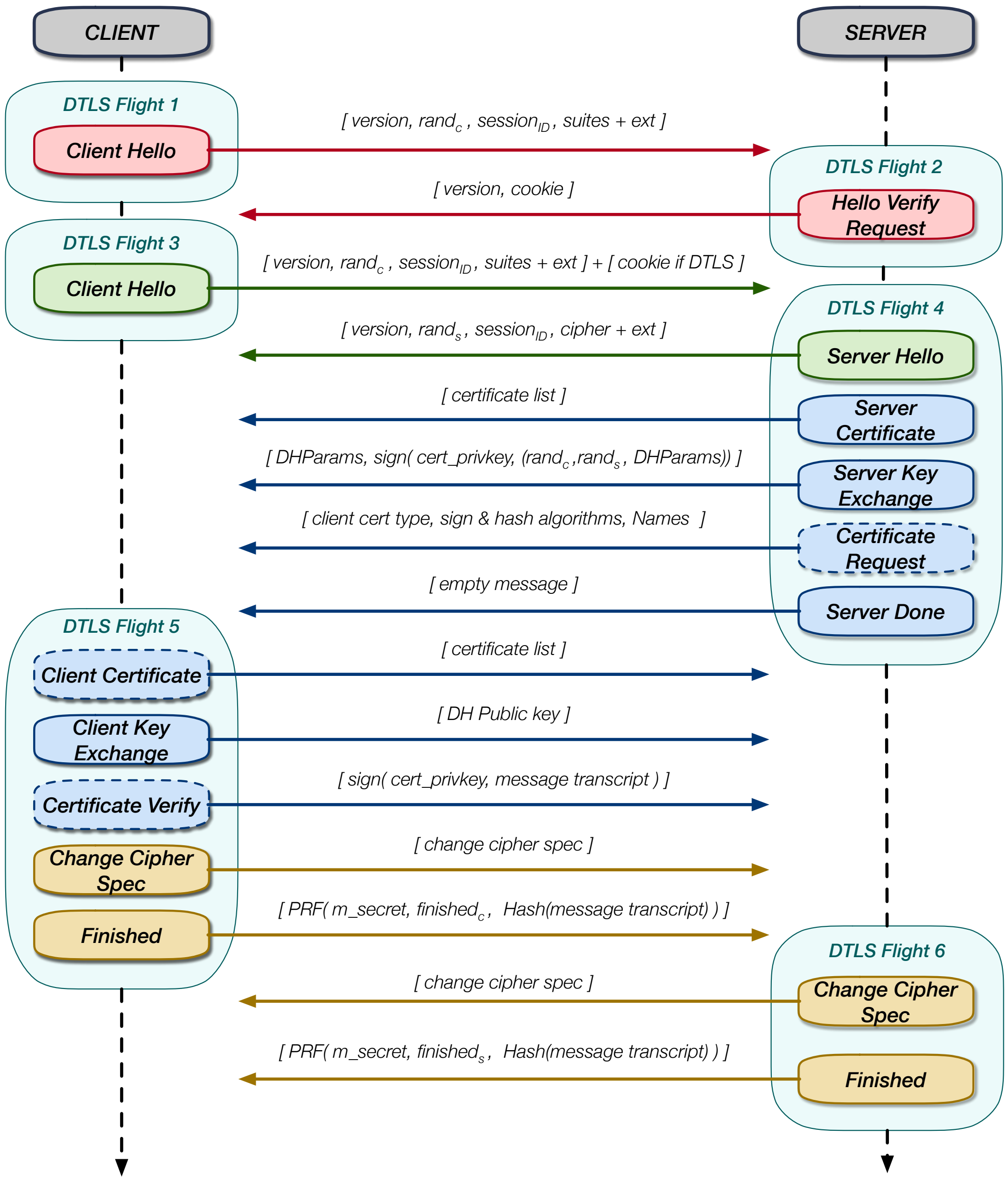


Figure 7.1: (D)TLS handshake protocol. The first two messages are a DTLS-only feature.

In the second phase, cryptographic parameters are transfered, necessary to derive the shared master secret. The peers also swap information to perform (mutual) authentication. When an appropriate cipher suite is chosen, e.g., when the suite uses `ECDHE_ECDSA` as a key exchange method, the server will send a *ServerCertificate* message. This message contains a list of X.509v3 certificates. The first certificate in the chain is the server's certificate, the last one is a self-signed certificate (potentially from a root CA (Certificate Authority)) and represents the trust anchor. If the client provided a `signature_algorithm` extension, then all certificates provided by the server must be signed by a signature and hash pair that appears in the extension. PSK authentication and raw public keys are also supported. The latter is defined in a TLS extension (RFC 7250 [184]). Following the certificate chain, the server sends a *ServerKeyExchange* message, if the client and the server negotiated an ephemeral key exchange mechanism. In case a static key exchange algorithm was selected, the client already has all the information necessary to continue with the key establishment. When the server wishes to perform mutual authentication on the transport layer, the server will request a client certificate through a *CertificateRequest* message. The server then notifies the client with the *ServerDone* message that it has no more data to send. In response to a *CertificateRequest* message, the client immediately sends its certificate chain. The client then follows-up with the *ClientKeyExchange* and the *CertificateVerify* message. The last two messages provide the server with the client's key share and a signature calculated with client's private key over the message transcript. The server can verify the authenticity of the client and derive the shared master secret.

In the final phase of the handshake protocol, the client sends a *ChangeCipherSpec* message, indicating that the new derived keys are installed in the record layer. This message must arrive before the client's last message, the *Finished* message. The latter contains the shared master secret, a string and a hash of the entire handshake encrypted with the negotiated encryption algorithm and the session key, derived from the master secret. The server responds with its own *ChangeCipherSpec* message and *Finished* message. The last roundtrip allows both peers to verify if all the handshake steps completed successfully.

7.1.1 Differences between TLS & DTLS

DTLS majorly follows the TLS specification, but there are some essential modifications due to the unreliability of the UDP transport layer. We will now briefly present the most important changes.

7.1.1.1 Record Protocol Changes

The DTLS record layer header has two additional fields compared to TLS. An *epoch* field and a *sequence number* field, called the RSN (Record Sequence Number), see Figure 7.2. Epoch numbers are used by endpoints to determine which cipher state has been used to protect the record fragment. The endpoints increment the epoch numbers on each *ChangeCipherSpec* message. Epoch numbers are required to resolve the ambiguity that arises when data loss occurs during a session renegotiation or when multiple handshakes are performed in close succession.

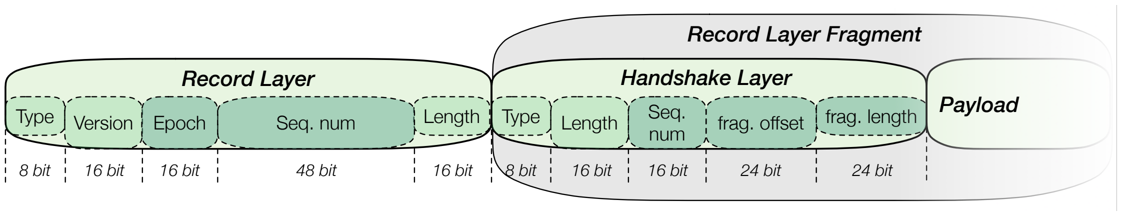


Figure 7.2: DTLS handshake message, wrapped in a record message

TLS employs implicit sequence numbers for replay protection. The sequence number is also used in the MAC calculation of the TLS records. RSNs play a similar role in DTLS but must be explicitly specified since records can get lost or can be delivered out-of-order. The DTLS record layer combines the RSN and the epoch number in a single 64-bit value while computing the MAC. RSNs increment by 1 for each record and are reset to zero whenever the cipher state is rolled over due to a session renegotiation. Thus, DTLS implementations must make sure the RSN/epoch pair is unique. DTLS can optionally performs replay detection by using the sliding window mechanism (defined in RFC 2401 [185]), see Figure 7.3. If datagrams always arrived in order, it would suffice for a DTLS endpoint to track the most recent RSN seen in order to detect replays. Since datagrams may also arrive out-of-order, a sliding replay window is required. The endpoints reject duplicate datagrams, and datagrams with RSNs that are lower than the left edge of the window.

DTLS records must fit in a single UDP datagram to prevent buffering of incomplete records on the DTLS record layer. DTLS should attempt to size records so that they do not trigger IP fragmentation along the way. Loss of a single IP fragment would result in the loss of the entire datagram. Also, NAT (Network Address Translation) devices and firewalls might drop IP fragments, and IP fragmentation is no longer supported by default in IPv6. It is noteworthy that both TLS and DTLS support an extension that can explicitly limit the size of the record fragments, called the MFL (Maximum Fragment Length) extension. Although, it is

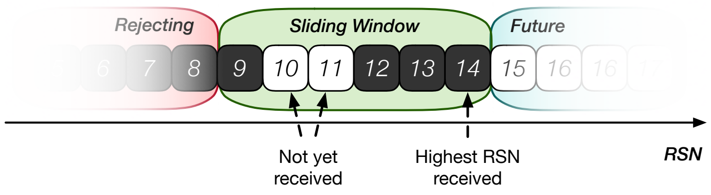


Figure 7.3: DTLS sliding window for replay detection.

not widely support by TLS servers, since the TCP protocol can perform segmentation to avoid oversized IP datagrams[1].

7.1.1.2 Handshake Protocol Changes

The unreliability of the transport layer had a major impact on the design of the DTLS handshake protocol. Unlike application data, the handshake protocol must exchange its messages reliably. They are necessary to derive a shared security context successfully. Therefore, the DTLS handshake protocol reimplements some of the mechanisms we can find in the TCP protocol, to compensate for UDP's unreliability. More precisely, the handshake protocol employs a retransmission mechanism and message sequence numbers. The handshake message sequence numbers are independent of the record layer sequence numbers. The retransmission operation works as follows: the DTLS handshake is divided into multiple *flights*, see Figure 7.1. Each time one of the peers has sent a full flight, it arms a timer. If the timer expires and the peer has not yet received the entire next flight, it will resend the entire previous flight. Handshake messages can grow larger than the PMTU (Path MTU), mostly due to long certificate chains. To prevent IP fragmentation DTLS, does support fragmentation, but only on the handshake layer. Handshake messages have a fragment offset field and fragment length field, see Figure 7.2, to enable reassembly at the receiver side.

DTLS uses a connectionless transport protocol, which makes it vulnerable to two types of DoS attacks. The first attack is a resource consumption attack where a large amount of malicious clients only send *ClientHello* packets to exhaust the resources of the server. The second attack is an amplification attack where malicious clients spoof the IP address of a victim device and send *ClientHello* to the server. The server then responds with an entire DTLS flight (containing the *server hello, server certificates, server key exchange*, and *server hello done*). To mitigate these attacks, DTLS can optionally use a cookie exchange technique. At the start of the handshake protocol, the client must replay a cookie provided by the server in order to demonstrate that it is capable of receiving packets at its claimed IP address. The cookie exchange adds one full RTT to the handshake latency compared to TLS. The technique is depicted in the first two messages in Figure 7.1. TLS does not suffer from the described attacks since it runs over TCP. The TCP handshake takes place before the first TLS message can be sent. It provides an efficient way of detecting address spoofing and potential DoS attacks.

7.2 The (D)TLS Handshake over IEEE 802.15.4E

Now that we understand the mechanisms behind the (D)TLS handshake, we can start our study of both protocols in the context of the IoT. For our comparison, we use the standard IoT networking stack. The lowest two layers are occupied by the IEEE 802.15.4E protocol, followed by 6LoWPAN, IPv6 and finally, either UDP and DTLS or TCP and TLS. We describe a scenario where a (D)TLS connection is established between a node in the IEEE 802.15.4E network and a device outside of the LoWPAN.

7.2.1 Efficient Handshake Reliability

Both protocols require reliability during the handshake phase, but we saw in the previous section that they tackle the issue in different ways. DTLS has a straightforward, albeit naive retransmission mechanism. DTLS schedules a retransmission when it has transmitted an entire DTLS flight. The retransmission occurs

[1]The TCP protocol can explicitly negotiate the maximum size of its segments through the MSS (Maximum Segment Size) option.

if the next flight is not received entirely within the timeout delay. The default timeout value is 1 s, and it doubles each time the transmission fails, maxing out at 60 s. Since neither UDP nor DTLS uses acknowledgment packets, there is no way of informing the peer which DTLS messages inside the flight were well received and which ones got lost. Therefore, DTLS is obligated to retransmit the entire flight.

The DTLS retransmission behavior is ill-suited for low-power networks for several reasons. First, both endpoints must be capable of estimating the latencies in the network correctly. The latter is tricky because TSCH network latency depends on the number of hops a UDP datagram needs to traverse, the density of the TSCH schedule, and the size of the UDP datagram. The TSCH network fragments large UDP datagrams into multiple 6LoWPAN frames. Each one of them is sent in an available active slot in the TSCH schedule. A network with a depth of multiple hops, combined with a sparse schedule, will induce a high latency on the end-to-end connection. Although the constrained node endpoint in the TSCH network might be aware of these factors impacting the RTT, the DTLS endpoint out on the Internet is probably not. Secondly, the cryptographic operations necessary to complete the handshake often require considerable time on low-power devices without hardware support, see Chapter 3. The operations need to complete before an answer can be sent back, adding a significant delay to the RTT. Unless the Internet DTLS endpoint has a very conservative initial retransmission value for the DTLS handshake, the cryptographic latency will cause many unnecessary retransmissions. Since DTLS retransmits entire flights, which can reach more than 1000 bytes for flights four and five, this has dire consequences for the operation of the TSCH network.

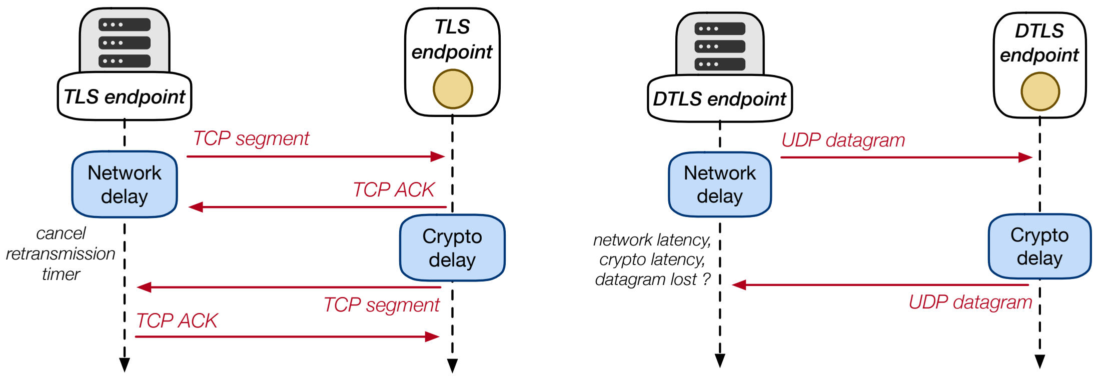


Figure 7.4: DTLS endpoints, not aware they are communicating with a constrained device that needs additional time for the cryptographic operations, are at risk of needlessly retransmitting entire DTLS flights.

TLS relies wholly on TCP to provide the necessary handshake reliability. TCP segments carry entire or partial TLS records, with each segment having its proper retransmission timer. Contrary to UDP, TCP uses acknowledgment packets in combination with its retransmission timers. The acknowledgments are cumulative, meaning that they acknowledgment all prior segments. They provide TCP with an accurate estimation of the RTT. The retransmission timer is initialized at 1 s upon completion of the TCP handshake and gets updated throughout the lifetime of the connection. TCP's flexible roundtrip estimation provides a significant advantage compared to the static values DTLS uses. However, the estimation happens on a per-segment basis and does not account for possible fragmentation on the lower layers. Several small TCP segments, not requiring any fragmentation, can traverse the TSCH network rather quickly, lowering the overall RTT estimation. If these small segments are followed by a large TCP segment, which requires 6LoWPAN fragmentation, the current RTT value might be too aggressive, triggering an unnecessary retransmission.

The behavior of the TCP retransmission mechanism can be improved for use in low-power networks. First, by employing the TCP's SACK (Selective Acknowledgment) option [186], the number of unnecessary retransmissions can be reduced. The SACK option allows the receiver to communicate to the sender which segments it has received and which require retransmission. The option is advantageous in a scenario where multiple TCP segments are on the wire, and segments successfully received are interleaved with segments lost. A receiver could then add one or more SACK blocks to its acknowledgment to precisely indicate which segments need retransmission.

A second improvement is less conventional and based on a PEP (Performance-Enhancing Proxy) [187, 188]. TCP PEPs are a commonly used technique to accelerate TCP connections over satellite links. TCP spoofing, a term synonymously used for TCP PEP functionality, intercepts a TCP connection before the

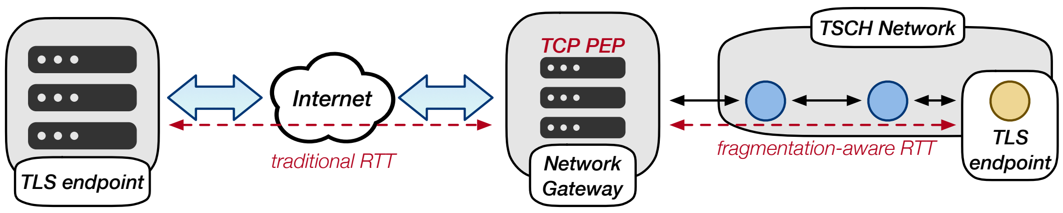


Figure 7.5: TCP spoofing to improve handshake reliability.

segments are sent to the satellite and terminates the connection as if the interceptor is the intended destination. It forwards the TCP segments further to the actual destination, but it accounts for the specificities of the satellite link, notably a long RTT. For the original TCP endpoints, the TCP PEP is transparent. Since the PEP generates the acknowledgments, they arrive much faster at the sender, allowing the TCP protocol on both endpoints to sustain a much higher throughput compared to a TCP satellite link without a PEP. In the context of the IoT, the PEP could be co-hosted with the network gateway. Its goal is to prevent unnecessary retransmissions that could occur due to the high latency induced by the peculiarities of the TSCH network. The network gateway intercepts the TCP connection and quickly generates acknowledgments for segments originating from the Internet endpoint. Since the network gateway is aware of the specifics of the TSCH network, it can calculate a novel RTT, which takes into account the fragmentation ratio of TCP segments.

7.2.2 Multi-Layer Fragmentation

The maximum size of a (D)TLS record is 16,535 B. Since a (D)TLS record is the *unit of protection*, meaning that the encryption and MAC are calculated over an entire record, the record must be fully received before decryption is possible. In practice, both (D)TLS endpoints must be capable of allocation an input and output of buffer 16,535 B to store incoming and outgoing records. Typically constrained devices cannot support buffers of this size. Several approaches are available to limit the size of the incoming and outgoing records. We can assume the application running on both endpoints is aware of the fact that at least one of the peers is constrained. The application should thus avoid passing large blobs of application data to the (D)TLS layer, as this would result in oversized records. Alternatively, the constrained peer can negotiate the MFL extension during the handshake. If both endpoints support the MFL extension and negotiate its usage, it restricts the maximum size of the record fragments during the handshake and the later exchanges of application data. Valid sizes are 512 B, 1024 B, 2048 B, and 4096 B. There are some drawbacks to the extension. Since this is an optional feature not all (D)TLS implementations support it. In addition, only clients can request the use of MFL extension. (D)TLS servers have no way of restricting the size of record fragments[2].

As mentioned before, the DTLS handshake protocol, by default, supports the fragmentation of handshake messages to make sure that they do not exceed the PMTU. Large handshake messages can frequently occur in the messages containing the certificate chains. Thus, the maximum record fragment size and by extension, the maximum length of a handshake message fragment, for DTLS depends on the PMTU. In a constrained network, the IPv6 MTU depends on the 6LoWPAN fragmentation layer. If the TSCH nodes are IPv6 compliant, the nodes should support the minimal IPv6 MTU of 1280 B. This requires a 6LoWPAN layer, which must be able to buffer up to 11 link-layer frames. Alternatively, the maximum fragment length extension, described above, can be used to set a maximum record fragment length that is smaller than the PMTU, e.g. 512 B, even if the IPv6 PMTU allows larger datagrams.

Just as DTLS, TLS clients can try to use the maximum fragment length extension to limit the size of the record fragments. No handshake fragmentation mechanism exists for TLS handshake messages since the TCP layer produces segments which it automatically limits in size accordingly to the PMTU. The TCP protocol also implements the TCP MSS option. This option allows both the client and the server to limit the size of the TCP segments, independent of the underlying PMTU. The latter is very helpful in environments where constrained nodes communicate over IPv6 but are not fully capable of supporting 1280 B IPv6 frames. The MSS option ensures that the IPv6 frames carrying a TCP segment will never exceed a specific value. Of course, TLS can also use the MFL extension to reduce the record fragment size.

[2]To address these issues the "record size limit" extension was defined. It is valid for all (D)TLS version and supposed to replace the deprecated MFL extension. [189]

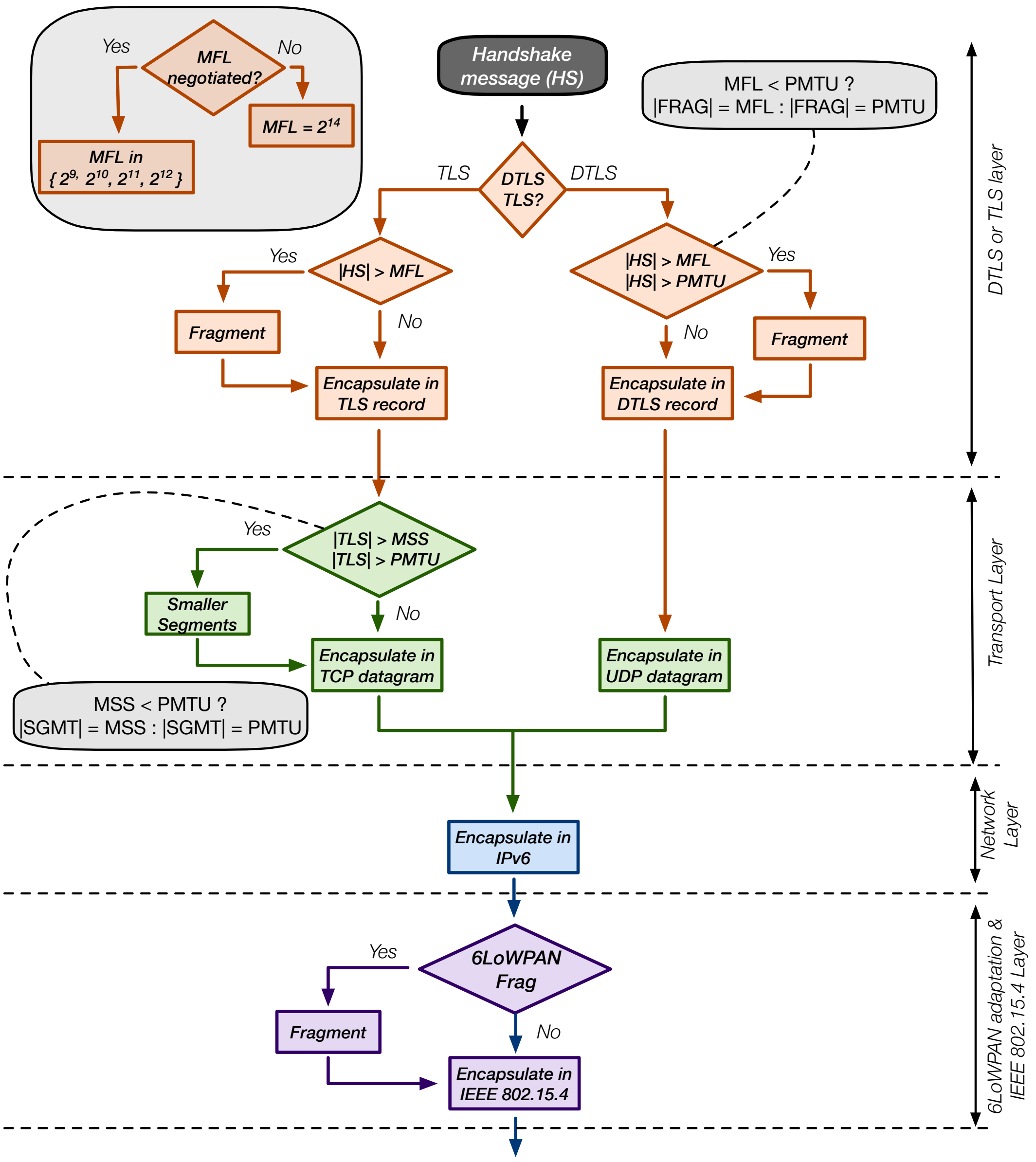

MFL negotiated?
Yes
No
MFL = 2^{14}
MFL in { 2^9, 2^{10}, 2^{11}, 2^{12} }
Handshake message (HS)
MFL < PMTU ?
|FRAG| = MFL : |FRAG| = PMTU
DTLS TLS?
TLS
DTLS
|HS| > MFL
|HS| > MFL
|HS| > PMTU
Yes
No
Fragment
Encapsulate in TLS record
Encapsulate in DTLS record
DTLS or TLS layer
|TLS| > MSS
|TLS| > PMTU
Smaller Segments
Encapsulate in TCP datagram
Encapsulate in UDP datagram
MSS < PMTU ?
|SGMT| = MSS : |SGMT| = PMTU
Transport Layer
Encapsulate in IPv6
Network Layer
6LoWPAN Frag
Fragment
Encapsulate in IEEE 802.15.4
6LoWPAN adaptation & IEEE 802.15.4 Layer

Figure 7.6: Multi-layer fragmentation for (D)TLS records.

7.2.3 Handling Burst Traffic

Typical applications running on top of constrained wireless networks periodically generate small amounts of data. TSCH uses a scheduling function to manage slot allocation and deletion in the schedule. The default scheduling function is called the MSF (Minimal Scheduling Function) and is being developed by the IETF [190]. It uses commands from 6top to manipulate the TSCH schedule. The MSF tracks the usage of the slots in the schedule. If slot usage surpasses a threshold value, the scheduling function will try to allocate more slots. If slot usage drops below a threshold, it will delete slots from the schedule. This approach works well when the applications only sporadically generate data. It keeps the duty cycle of nodes minimal, reducing energy consumption in the network. However, the (D)TLS handshake does not follow this paradigm. It generates a massive burst of traffic, which could easily overflow the small packet buffer sizes on the nodes, leading to congestion and packet losses. The MSF function is not adapted to handle these traffic patterns. Any allocation done for burst traffic should not be limited to the communication link between the traffic originator and its parent but should extend over all the hops to the network sink. Currently, the 6top does not provide a command that can instruct nodes to build such a "tunnel", but the 6top `addCell` command does contain a 2 B field for opaque metadata that should be passed directly to scheduling algorithms running on top of MSF. Alternative scheduling functions or future updates of the MSF IETF draft [190] could use this field to instruct the nodes to forward the allocation request to all the hops between the originator and the network sink.

Congestion control is not available when using UDP as transport protocol and DTLS does not incorporate any of the TCP congestion control mechanism. One solution might be to allow the application running on the constrained node to explicitly request the lower layers to allocate more slots and potentially build a tunnel to quickly evacuate the burst traffic from the network, as discussed above. The application should make this request before it starts the handshake procedure. When the handshake completes, it is the application's responsibility to tear down the tunnel. DTLS messages could also be carried over DCCP (Datagram Congestion Control Protocol) [191], which provides congestion control for unreliable datagrams.

TCP provides multiple mechanisms to handle congestion traffic inside the network. However, the traditional congestion mechanisms (AIMD (Additive Increase/Multiplicate Decrease) and TCP slow start) are not well-suited for wireless networks. The congestion mechanism increases or decreases the size of the congestion window, which is typically up to 4 times the MSS [192]. Alternatively, RFC 6928 [193] defined an experimental new value for the initial congestion window, which in practice results in an initial window of 10 times the MSS. The latter is nowadays used in many TCP implementations [98]. Since typical constrained applications periodically send small bits of information, the congestion mechanism is not useful. In case of burst traffic like the (D)TLS handshake, an initial small congestion window could limit the sending rate, if a low enough MSS was negotiated, but then rapidly allow for more we data in flight. A more appropriate congestion mechanism for highly constrained networks is Nagle's algorithm. Nagle's algorithm limits the amount of data in flight to a single full-sized MSS, unless another full-sized segment is available. Nagle's algorithm is depicted in Algorithm 7. Constrained nodes typically use statically allocated buffers, and Nagle's algorithm will ensure that they are utilized at maximum efficiency. Without Nagle, many small segments could occupy buffer spaces foreseen for bigger segments, thus wasting a lot of RAM.

Algorithm 7 Nagle's Algorithm

```
1: procedure TRANSMIT($D$)
2:     if wnd_size >= mss and $|D|$ >= mss then
3:         Send($D$)
4:     else
5:         if data in flight then
6:             Queue($D$)
7:         else
8:             Send($D$)
```

The IETF draft on lightweight TCP [98] also mentions the use of the ECN (Explicit Congestion Control) bit in combination with TCP to limit network congestion. ECN allows a router (intermediate node) to signal a warning for looming congestion by setting a bit in the IP header of a packet, e.g., when the internal buffer has reached 75% of its capacity. An ECN-enabled TCP receiver will echo back the congestion warning to the TCP sender by setting the ECN flag in its next acknowledgment. The sender then triggers congestion control measures as if a packet loss had occurred. Finally, the use of delayed acknowledgment packets can also help reducing congestion in the network. TCP's delayed acknowledgments are meant to reduce the number of acknowledgment packets sent within a TCP connection, thus reducing network overhead. However, it is well-known that delayed acknowledgments should not be used in combination with Nagle's algorithm since this would hurt throughput in the network.

7.3 Performance Evaluation

7.3.1 Experimental Setup

7.3.1.1 Software stack

In order to obtain experimental results on the performance of DTLS and TLS on top of IEEE 802.15.4E, we ported the mbed TLS library to the OpenWSN project [194]. We had to overcome several challenges before we could start with the evaluation. Two core parts are missing in the current version of the OpenWSN stack:

there is no 6LoWPAN fragmentation mechanism to handle large IPv6 datagrams, and the TCP protocol is not implemented. We implemented both protocols and integrated them with the OpenWSN stack. We designed two abstraction layers, called `opendtls` and `opentls`, which function as a wrapper around the mbed TLS library and allow applications to interact with the (D)TLS stack and open secure channels.

At the lower layers, OpenWSN uses timers and interrupts to implement the TSCH schedule. On the boundary of each timeslot, the node wakes up, e.g., every 20 ms. It checks the slot type and acts accordingly. During an active slot, the software will schedule additional timers implementing the internal timeslot behavior, described in Chapter 6 and Figure 6.1. When a frame is sent or received, the IEEE 802.15.4E software layer schedules an upper-layer task (6top task). OpenWSN uses a simple non-preemptive task scheduler to execute the tasks. It executes the tasks according to a statically specified priority. OpenWSN does not use any heap memory; all variables are global or live on the stack. Since mbed TLS needs a heap to perform its cryptographic operations, it provides its proper heap allocation functions. A large memory buffer is declared as a global variable and passed to mbed TLS to use as a heap.

7.3.1.2 Hardware

We used state-of-the-art OpenMote hardware to perform our experiments. The OpenMote platform uses the CC2538 SoC [14]. It has 32 KiB of memory and a 32 MHz Arm Cortex-M3 processor. Additionally, it provides a cryptographic co-processor for AES and SHA functions and hardware acceleration for several big integer and elliptic curve operations. The sink node of the TSCH network is connected through a serial interface to a PC implementing the network gateway, see Figure 7.7. The network gateway software, called openvisualizer [195], is also provided by the OpenWSN project. It implements the 6LoWPAN compression, decompression, fragmentation, and reassembly of TSCH frames. It uses a TUN interface to route the packets to their final destination.

7.3.2 Handshake Measurements

We performed several experiments using different sets of configuration parameters to assess the impact on the performance of both the TLS and DTLS stack. During the experiments, a low-power node opens a (D)TLS connection to a server on the Internet. The TCP/ip network stack of the server is not modified and is unaware it is talking with a constrained device. The (D)TLS handshake always uses mutual authentication through the exchange of certificates, unless noted otherwise. The certificate chains exchanged are only one certificate long. The received certificate is matched against a stored root certificate which acts a trust anchor for the authentication.

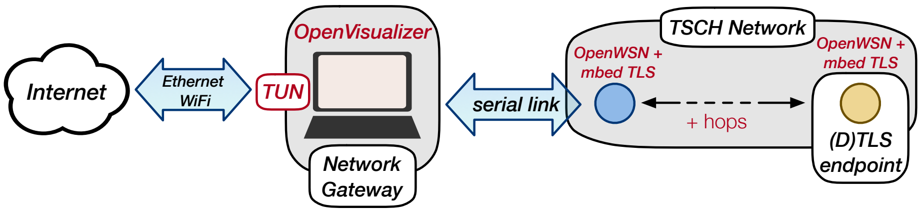


Figure 7.7: Experimental setup to analyze (D)TLS handshake performance.

7.3.2.1 Cryptographic Impact

The different asymmetric primitives used during the secure connection setup can significantly impact the duration of the handshake. We test two elliptic curves of different sizes (`secp256r1` and `secp192r1`). The chosen key establishment cipher suite is `ECDHE_ECDSA`. We can configure our devices to either use the software-only implementation of the primitives or leverage the hardware support on the OpenMote platform. The mbed TLS stack provides an exciting feature for the software-only implementations. It allows us to interrupt the cryptographic operations and treat other high-priority tasks first. Later we can return to the operation and continue where it was interrupted. In single-threaded environments, with non-preemptive scheduling, this keeps the system responsive even when the cryptographic operations take several seconds to complete (see Chapter 2). Figure 7.2 shows how cryptographic operations can impact the total duration of the handshake.

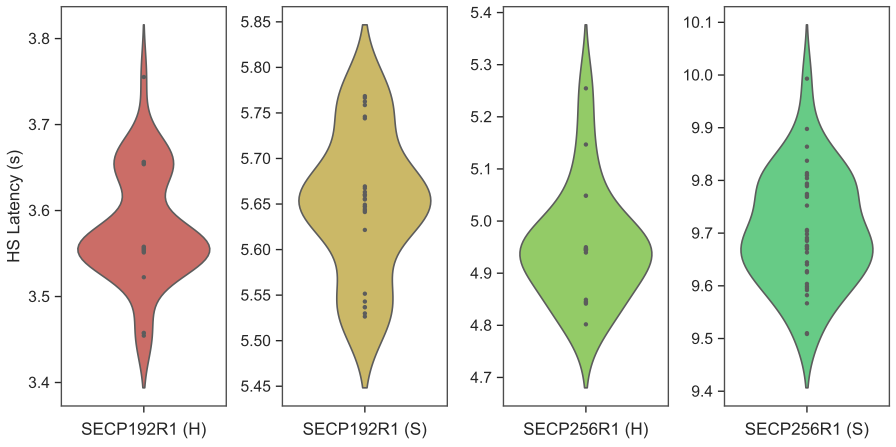
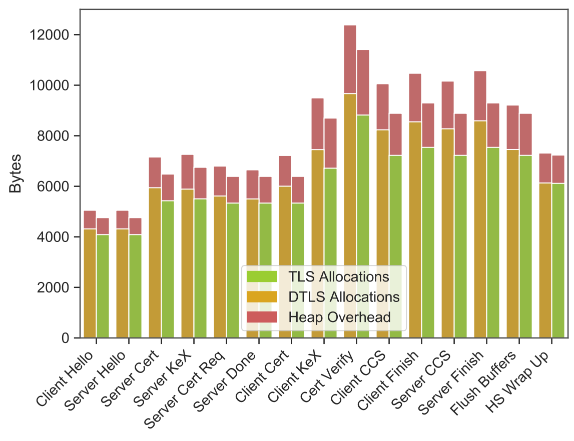
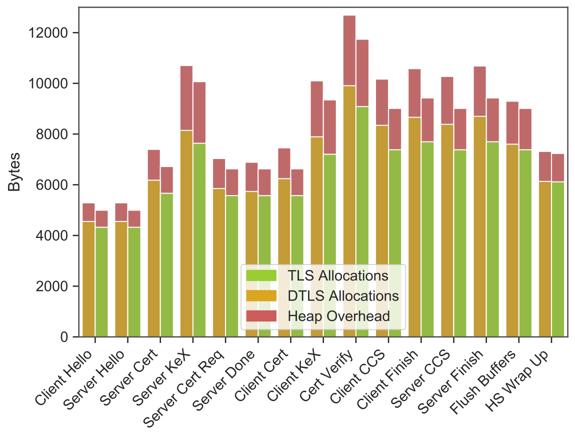


Figure 7.8: TLS handshake latency using `secp192r1` and `secp256r1`, with and without hardware support.

In addition to the impact on the handshake latency, the memory pressure devices experience also depends on the chosen curve and the availability of hardware support. Every heap allocation comes with an additional header block, which contains metadata about the allocation. In mbed TLS, this header block is 32 B long. Figure 7.9 shows the heap allocations throughout the TLS handshake. The lower bar is the sum of all allocation for a given (D)TLS state, while the upper part signifies the overhead of metadata on the heap.



(a) Cryptography accelerated



(b) Software-only cryptography

Figure 7.9: Memory pressure of the (D)TLS handshake, performing mutual authentication with certificates.

We observe that there are less heap allocations when using hardware support for specific steps in the handshake. The difference is evident when the node treats the *ServerKeyExchange* record. For other steps in the handshake, the difference is limited. Two factors explain this behavior. Firstly, we mentioned before that the OpenMote does not have a co-processor for asymmetric algorithms, it has hardware acceleration for specific operations on elliptic curves, e.g., scalar multiplication, point addition, etc. Therefore, we cannot accelerate the entire asymmetric algorithms, merely subparts of it. Secondly, the mbed TLS software only exports specific function signatures that can be overwritten to provide internal calls to hardware acceleration. We also notice that even before the handshake starts, there is roughly 4600 B already allocated on the heap. Mbed TLS stores multiple data structures on the heap before the handshake commences. These contain the device's root certificate, private key, and variables describing the TLS state machine.

7.3.2.2 Impact of Network Stack Configuration

Maximum Transmission Unit defines the largest payload size that can be transmitted on the network layer. IPv6-compliant IoT devices should support an MTU of 1280 B [196]. However, the tight memory constraints on low-power IoT devices make this a challenging requirement. When we compile the OpenWSN stack with our TCP implementation, it supports IPv6 datagrams with a maximum payload size of 864 B. Combining OpenWSN with UDP allows for an MTU of 1377 B. Since TCP requires more RAM than UDP, there is fewer buffer space left to store 6LoWPAN fragments, that can be reassembled in a large IPv6 datagram, resulting in a lower MTU. TCP requires RAM for the variables maintaining the connection state, the TCP state machine, and the send and receive buffers. The size of the MTU affects the amount of bytes transmitted during the (D)TLS handshake.

TABLE 7.1: Impact of the MTU on the TLS and DTLS handshake

SSL-type	Restrictions [B]		MTU [B]	Latency [s]		Bytes TxRx [B]		Nagle
	MSS	MFL		μ	σ	μ	σ	
TLS	844	-	864	4.495	0.096	3102	5.49	✓
	844	-	864	4.285	0.076	3358	0.60	-
	336	-	356	4.539	0.089	3345	1.25	✓
	336	-	356	4.241	0.118	3671	3.74	-
DTLS	-	512	529	3.891	0.090	2841	0.83	-
	-	1024	1041	3.795	0.064	2680	1.00	-

Table 7.1 shows the handshake latency and the total number of exchanged bytes between both endpoints to establish the shared (D)TLS security contexts. We count the bytes of the received and transmitted IEEE 802.15.4E frames containing handshake data. We tested various MTU configurations. As explained before, during the TLS handshake, we can limit the size of the TCP segments, and by extension, the IPv6 datagrams by choosing a small MSS value during the TCP handshake. When we compare the TLS handshakes with the MTUs at 864 B and 356 B, respectively, it shows that a larger MTU results in fewer transmitted bytes. Since more TLS data fits in a TCP segment, it reduces the overhead of the TCP header and the number of acknowledgments necessary. By activating Nagle's algorithm, we can further reduce the number of transmitted bytes.

Recall that in the DTLS protocol, records must fit entirely into a UDP datagram. Because of the potentially sizeable handshake messages, the DTLS handshake layer implements a fragmentation mechanism. Using the MFL extension, a client can inform the server of the maximum record fragment size it supports. With an IPv6 MTU of 1377 B, the node can use two different MFL sizes: 512 B and 1024 B. Both values indicate the size of the record fragment, thus to derive the MTU, we add 13 B for the DTLS record header, and a 4 B compressed UDP header. Similarly to TCP, a higher IPv6 MTU results in fewer datagrams and thus less overhead caused by additional headers. Compared to TLS, DTLS uses fewer bytes to complete the handshake. Several factors contribute to this difference. UDP does not use acknowledgment packets, and UDP and TCP have significantly different header sizes. The UDP header is even further compressed from the standard 8 B down to 4 B through 6LoWPAN header compression, while TCP uses an uncompressed header of 20 B.

The second aspect we evaluated is the handshake latency. We observe that the MTU only slightly impacts the overall latency of both the TLS and DTLS handshake. When we activate Nagle's algorithm in the TCP algorithm, it incurs an additional delay. Without Nagle, TCP can pipeline the segments, having multiple unacknowledged segments in transit. Nagle's algorithm optimizes the segment space but limits the number of unacknowledged segments in flight. To time the latency of the TLS, we start the clock when the client sends it first *syn* segment to open the TCP connection. While timing the DTLS handshake latency, we start the clock when the client sends the initial *ClientHello*, triggering DoS protection on the server endpoint. To prevent unnecessary DTLS retransmissions, we set the DTLS timeout value to a conservative 3 s interval. To avert redundant TCP retransmissions, we drop the initial *synack* segment issued by the server, while performing the TCP handshake. The latter causes a single *synack* retransmission but also opens the connection with a more conservative fallback RTO (Retransmission Timeout) value of 3 s instead of 1 s, ensuring that TCP does not needlessly retransmits during the TLS handshake [197]. It does, however, induce an additional 1 s delay to the TLS handshake.

The final parameter that is impacted by the MTU is the internal 6LoWPAN reassembly buffer pressure on the constrained (D)TLS endpoint. We did not consider the buffer pressure of the intermediate router nodes since they use fast fragment forwarding [115]. The 6LoWPAN fragments are not reassembled on the routers and are directly forwarded to the next hop, see Chapter 3. As long as the intermediate routers have sufficient active slots, dedicated to communication with their parents and children, their buffers will not be at risk of overflowing.

The size of the MTU affects the buffer pressure since a node must be able to store all the individual fragments before it can reassemble the IPv6 datagram and treat it in the higher layers of the stack. Figure 7.10 shows the results for varying MTU configurations for both TLS and DTLS. We remark that for large MTUs, Nagle's algorithm increases the buffer pressure (see the top plot in Figure 7.10). The large segments require many 6LoWPAN fragments. We can clearly distinguish two phases in the handshake protocol. The initial peak corresponds to the reception of the *ServerCertificate*, *ServerKeyExchange*, *CertificateVerify*, and *ServerDone* packets. The second bump is the accumulation of 6LoWPAN fragments being queued for transmission, after the node has prepared its *ClientCertificate*, *ClientKeyExchange*, *CertificateVerify*, *ChangeCipherSpec*, and *Finished* packets. Without Nagle, the endpoints are not obligated to put all the TLS records in a single TCP segment, resulting in fewer 6LoWPAN fragments per IPv6 datagram.

When we lower the MTU to 356 B, the behavior is inverted. Without Nagle, the node quickly sends multiple, shorter TCP segments, which are all fragmented and subsequently queued for transmission. Interestingly, the initial peak, caused by the incoming server TLS handshake messages, has almost completely disappeared. With Nagle activated and an MTU of 356 B, the client's handshake messages are split into two TCP segments. The first one is filled and transmitted directly; the second segment is not full and can thus not be sent until an acknowledgment is received for the first segment (see Algorithm 7).

Lowering the MTU even further does not reduce buffer pressure. With a lower MTU, TCP can fill many segments. Since Nagle, tells us that multiple filled segments can be transmitted without having received an acknowledgment, all the segments are queued for transmission, creating many 6LoWPAN fragments in the internal buffer.

The DTLS handshake behaves similarly to a TLS connection that uses a large MTU and Nagle's algorithm. DTLS sends entire flights at once, causing many 6LoWPAN fragments in transit. They must all be correctly received before reassembly can take place. Lowering the MTU lowers the buffer pressure for incoming messages, but does nothing to alleviate buffer pressure for outgoing messages.

Performance Enhancing Proxies are an alternative solution to prevent TCP's retransmission problem without modifying the TCP implementation. By enabling the gateway as a PEP, we can partially circumvent the additional delays introduced by the IEEE 802.15.4E link layer. Table 7.2 depicts the results. We notice that the TLS handshake takes one second less to complete, with the PEP enabled. It even completes faster than the DTLS handshake. The latter might seem surprising since the TLS connection needs to exchange more data to establish the secure connection. However, the DTLS handshake (with an MTU of 1041 B) requires 25 IEEE 802.15.4E frames to complete. Only one of those frames is not a 6LoWPAN fragment and, thus, does not induce fragmentation and reassembly delays. The TLS handshake requires 30 IEEE 802.15.4E frames to complete. The additional overhead comes from the TCP acknowledgments. Ten of the 30 frames are sufficiently small to traverse the network without fragmentation, 20 frames are 6LoWPAN fragments. We can thus conclude that with the configuration depicted in Table 7.2, DTLS spends a significant time waiting for all the 6LoWPAN fragments, because of the high fragmentation rate of the UDP datagrams.

TSCH's schedule has the most significant impact on the handshake latency. Until now, all measurements have used a schedule with only active slots. All slots were either allocated for transmission or reception. Figure 7.11 shows the handshake latency with respect to the number of active slots in the schedule. The slots are allocated at random in the slotframe by MSF. We use the network gateway as a PEP, and we choose a conservative DTLS timeout to prevent needless TCP retransmissions when we grow the TSCH schedule and thus lower the duty cycle of the nodes. For both TLS and DTLS, we maximize the MTU, and we activate

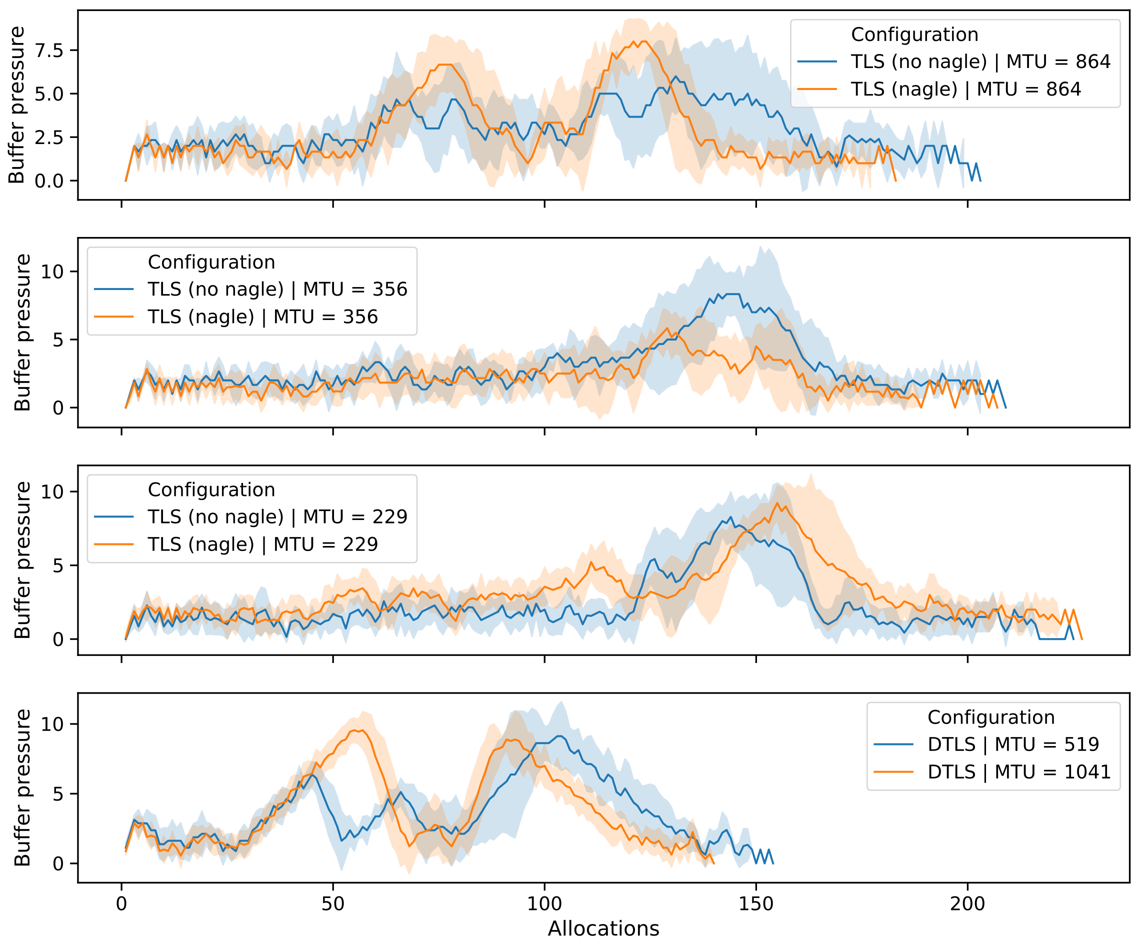


Figure 7.10: Impact of the MTU choice on the internal reassembly buffer of the constrained (D)TLS endpoint.

TABLE 7.2: Performance-enhancing proxy and TLS

SSL-type	Restrictions	MTU [B]	Latency [s]		Bytes TxRx [B]		Nagle
	MSS		μ	σ	μ	σ	
TLS	844	864	3.323	0.125	3098	2.13	✓
	844	864	3.209	0.083	3359	1.15	-

Nagle's algorithm for TCP. For the single-hop setup, the latency of both the TLS and DTLS handshake is quite similar. Until 25% active slots, TLS performs slightly better, but when we increase the slotframe size further, the delay induced by the TCP acknowledgment packets becomes more significant. When we repeat the same experiment with a two-hop network, the behavior is similar and amplified.

7.3.2.3 Handshake Reliability

We already saw that both TLS and DTLS approach the issue of handshake reliability differently. Recall that TLS entirely relies on the functionalities of TCP to handle packet losses, while DTLS comes with a simple retransmission mechanism. IEEE 802.15.4E provides a rather reliable link and physical layer. Channel hopping and dedicated slots in the schedule mitigate many of the transmission errors due to multi-path fading and collisions. The link layer uses acknowledgments and up to 15 retransmissions combined with a back-off mechanism to minimize losses. However, IEEE 802.15.4E frames can be dropped quickly due to limited buffer sizes in the nodes. In the previous section, we saw that some transport configurations incur a high buffer pressure. To test how both TLS and DTLS behave when the lower layers do not provide 100% reliability, we set up an experiment where 6LoWPAN fragments had a 5% chance of being dropped

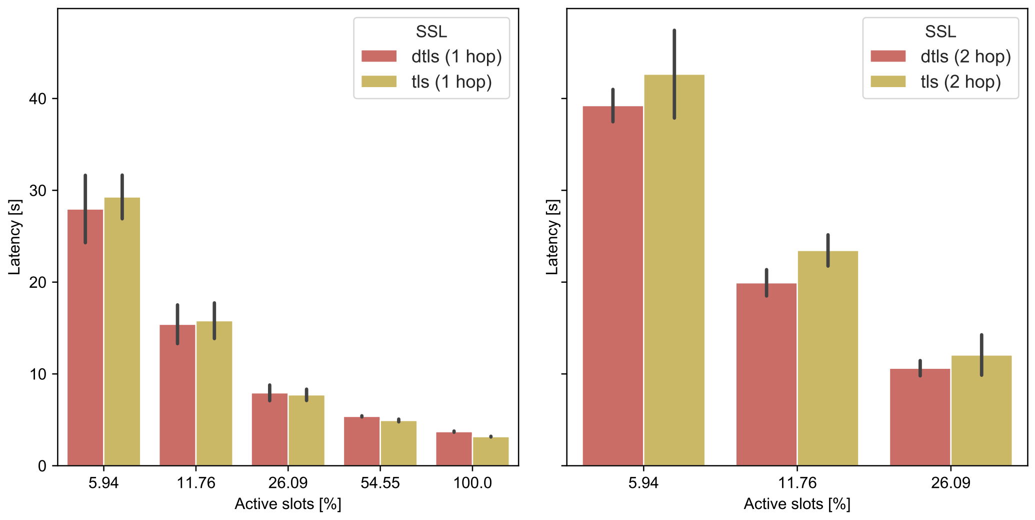


Figure 7.11: Impact of the TSCH schedule on the handshake latency.

when traversing the TSCH network. Figure 7.12 shows the results for the TLS handshake. As we already discussed in the previous sections, a lower MTU results in more bytes exchanged to establish up the secure connection. However, when there are losses, the smaller TCP segments allow to more precisely indicate, through the use of the TCP SACK option, which segments require retransmission. When the receiver can accurately indicate which losses occurred, the transmitter can minimize the data it has to retransmit. This means there are fewer 6LoWPAN fragments on the network, and the handshake will finish faster.

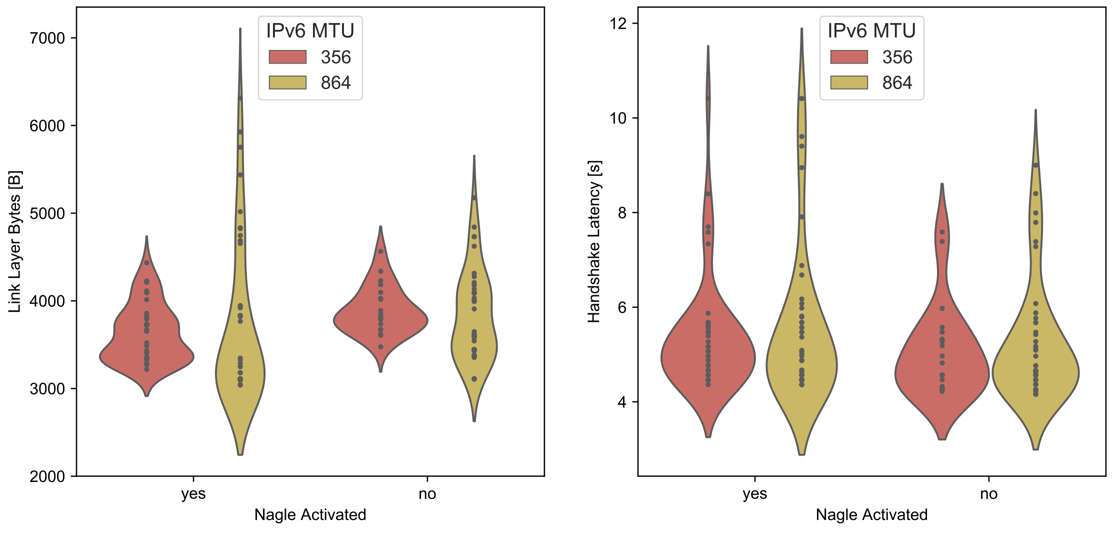


Figure 7.12: Impact of lossy links on the TLS handshake protocol.

In a second experiment, we looked at the behavior of the DTLS handshake over lossy links. The simple retransmission mechanism of DTLS does not provide any feedback mechanisms between transmitter and receiver. Since all the DTLS records are combined in only a few large UDP datagrams, loss of a single 6LoWPAN fragment results in the loss of numerous DTLS records, and the entire flight is retransmitted. Figure 7.13 depicts the comparison between the lossy TLS and DTLS handshake.

We directly notice that the lack of control over the retransmission mechanism provided by DTLS, results in a drastic increase in the number of bytes exchanged between the DTLS endpoints. The latter also clearly affects the handshake latency.

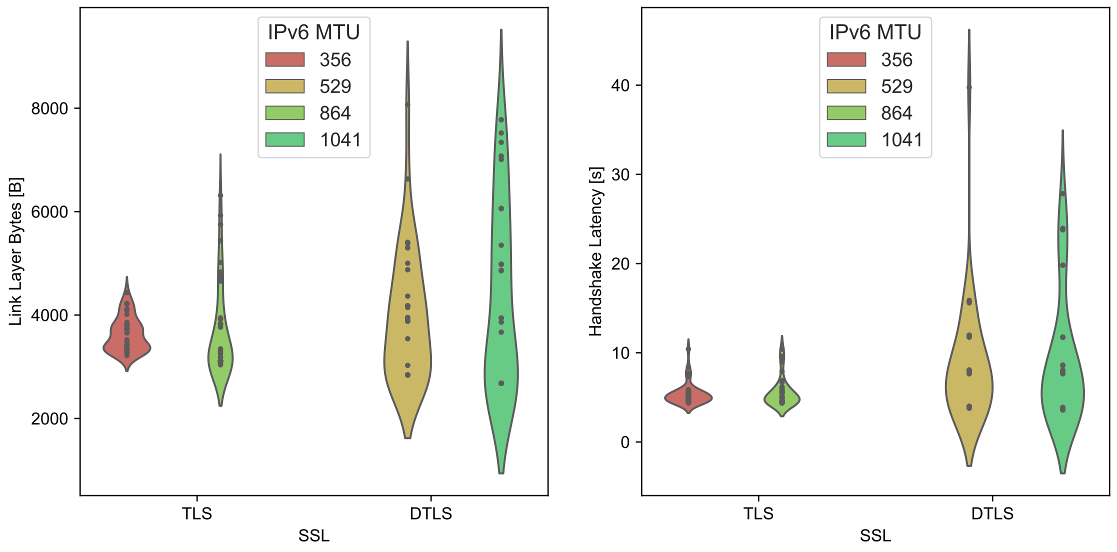


Figure 7.13: Comparison of (D)TLS handshake behavior over lossy links.

Conclusion

This chapter presented our third contribution. It provided an in-depth analysis of the most critical part of the (D)TLS protocol: the handshake. We structurally presented the differences between TLS 1.2 and DTLS 1.2 and analyzed how they impact the handshake's performance when the messages are carried in IEEE 802.15.4E frames. We showed that DTLS needs fewer bytes to complete the handshake. It has a lower handshake latency than TLS when the (D)TLS endpoint is a leaf node whose frames need to traverse multiple hops before they reach the gateway. However, there are several other aspects where TLS achieves better results than DTLS.

Firstly, TLS has a slightly lower memory consumption than DTLS during the handshake. We can attribute the difference to the fact that the DTLS handshake protocol must use an internal buffer to reassemble the fragmented handshake messages. When we also account for the difference in memory consumption due to the complexity of TCP compared to UDP, the results are different. They depend majorly on the sizes of the TCP send and receive buffers.

Secondly, during the handshake, TLS can take full advantage of the many TCP optimization mechanisms. The MSS option and Nagle's algorithm can collaborate to relieve reassembly buffer pressure. The TCP MSS and SACK options are extremely helpful in lossy environments, and they allow the TLS handshake to minimize its impact when packet losses occur. The DTLS handshake suffers from high latency and numerous redundant retransmission over lossy networks.

Finally, the usage of a TCP PEP allows a constrained device to interact with a powerful server. It is not necessary to tweak the server's behavior and accommodate for the limited capabilities of the low-power client. The TCP acknowledgments allow the endpoints to distinguish between delays induced by the network and delays caused by the cryptographic computations. DTLS does not have such a feedback mechanism. A server performing the handshake protocol does not know if the delays it measures between outgoing and incoming messages are due to network latencies or cryptographic computations. DTLS requires prior knowledge of the network latencies and the capabilities of the other endpoint. Otherwise, unnecessary retransmission might occur.

Future work should focus on the analysis of the new (D)TLS 1.3 protocol. The speedup of the handshake protocol in (D)TLS 1.3 (from two to one roundtrip) should also benefit the constrained devices. In addition, a comparison with the new object security protocols tailored for the IoT would be very valuable.

Since we have demonstrated that TLS is a worthy replacement for DTLS in certain scenarios, it is also interesting to revive the abandoned 6LoWPAN header compression mechanism for TCP [97]. Initial results show that the 20 B header can be compressed to merely 6 B in 95% of the cases [198]. The TCP header compression would then easily save up to 250 B during the TLS handshake.

Chapter 8

Security Architectures for the Internet of Things

Contents

Introduction

In our previous contributions, we focussed primarily on challenges unique to the characteristics of the constrained IoT, i.e., the lack of hardware-enforced security, vulnerabilities in low-power link layer protocols and the difficulty of establishing a secure channel over lossy networks. Besides these issues, the IoT also inherits various practical problems from the traditional Internet. In our last contribution, we present the issues of authorization, authentication, and secret key establishment for the IoT.

The IoT exposes vast amounts of data generated by billions of sensors. This data, to which we will further refer as *protected resources*, is only valuable if the interested parties, a.k.a. *clients*, can access it. The *resource owners* plausibly want to limit access to the protected resources. Only authorized clients should be able to retrieve the resources. The owner could wish to restrict access on the grounds of privacy or because the protected resources are part of its revenue model. In any case, some form of negotiation has to occur between the resource owner and the possible clients. In Chapter 3, we presented two frameworks that solve this issue in a scalable manner. On the web, the most common approach is to use the OAuth (Open Authorization) 2.0 framework, which authorizes clients to access specific protected resources on behalf of the resource owner. OAuth 2.0 extensively uses access tokens, issued by a trusted third-party denoted as the AS (Authorization Server). The tokens encode claims and delegate protected resource access rights to the token possessor. The ACE (Authentication and Authorization for Constrained Environments) working group adapted the OAuth 2.0 framework to the context of the IoT. The resulting ACE framework implements both authorization and authentication between the clients and resource servers while possibly both sides are constrained.

Both OAuth and ACE rely entirely on the underlying transport protocol to provide secure communication. It is the responsibility of the security protocol to establish an authenticated shared secret and encrypt the connections between the various endpoints. OAuth 2.0 should always be used in combination with a secure communication protocol, i.e., TLS, to defend against access token theft. ACE defines multiple security profiles. ACE currently supports DTLS and OSCORE (Object Security for Constrained RESTful Environments) as underlying security profile. ACE and OAuth share a common limitation. In addition to the necessary trust anchor for endpoint authentication, e.g., PKI certificates for TLS and DTLS or authentication servers for OSCORE and EDHOC (Ephemeral Diffie-Hellman over COSE), OAuth and ACE define the central AS as an trusted third-party. In the ACE framework, the AS fulfills several security-critical operations. Firstly, it generates the access tokens, based on rules provided by the resource owner, granting the client access to the protected resources on the resource servers. Secondly, it performs token introspections in case of highly constrained resource servers, and finally, it distributes the cryptographic PoP (Proof-of-Possession) keys, enabling the client to proof its legitimate ownership over the token.

In this final chapter, we propose two novel security frameworks inspired by OAuth, ACE, EDHOC and OSCORE. In both proposals, we mainly keep the same architectural design and protocol flows as in ACE, but we append and integrate elements to build a framework that provides both authorization and authenticated key establishment while minimizing the trust in the AS. In the first proposal, we describe an architecture that uses self-protecting and self-contained tokens, through the use of COSE (CBOR Object Signing and Encryption) objects and the CBOR (Concise Binary Object Representation) web token format, which serve two goals: authorizing the access to the protected resources and providing mutual authentication for key establishment. To reduce the overall trust the participating devices must place in the AS we rely on the well-established PKI. To lessen the impact of the asymmetric operations, we designed reusable tokens. On each reiteration of the token, except the initial token exchange, only symmetric key operations are necessary. The reusable tokens also diminish the reduces the amount of communication in the network.

In the second proposal, we increase the robustness of the authorization architecture by replacing the single trusted AS endpoint by a decentralized authorization system, based on blockchain technology. The role of the AS is fulfilled by blockchain miners, which manage authorization requests through the use of smart contracts. The smart contracts are published by the resource owners. Key servers hold the symmetric keys that are used by the resource servers to encrypt protected resources. The smart contracts add access tokens for the authorized to the blockchain. A client can subsequently issue a request to the key server for a set of resource decryption keys. If the key server can find a valid token for the client on the public blockchain, it transfers the keys. Finally, the client recovers the encrypted resources from the resource server or an intermediate cache and decrypts them locally.

8.1 Token-based Authenticated Key Establishment

8.1.1 Drawbacks of Standard ACE

In the ACE framework, access tokens issued by an AS are instantly valid for all the resource servers in their domain. This powerful ability turns the AS into a valuable target for attackers. A compromised AS can obtain all protected resources. The PoP mechanisms cannot defend against a compromised AS. PoP binds either a symmetric key or public key to the access token. In the former case, the AS generates the symmetric key and performs the binding operation, i.e., signs the token. The AS has thus access to the secret key. In the latter scenario, the client creates an asymmetric key pair and sends the public part to the AS to bind it to the token. Here, the AS does not have access to the private key, but since the key pair is not authenticated in any way, e.g., by a certificate chain, it does not add protection against a rogue AS.

Since ACE also considers that the client can be constrained, it recommends the use of long-term tokens. To provide this feature, the ACE framework proposes to use token introspection. The client is not issued a real token by the AS, but a reference to it. When the client sends the reference to the resource server, it will in turn contact the AS to learn which claims are associated with the token reference. The tokens are thus not self-contained and they force frequent communication between the AS and the resource server.

8.1.2 Security Goals

Before we detail the design of our adapted ACE framework any further, we need to outline the different threats to which our architecture must be resilient. The threat model lists token-specific threats and threats against endpoints in the architecture. The token-related threats are:

- **Access token integrity:** We want to prevent unauthorized entities from forging valid tokens or from freely modifying the content of the token after its creation.
- **Token theft:** A token theft detection mechanism must be in place to stop unauthorized entities from using stolen tokens to access protected resources.

We list the threats against the different endpoint in the architecture below:

- **Resource server security:** The resource servers are often constrained devices, deployed in remote areas. The lack of physical protection and memory isolation makes the devices vulnerable to a wide range of physical attacks, network attacks, and remote code injections. Therefore, we want to minimize the storage of secrets in the resource servers and continuously authenticate the devices during communication.
- **Client security:** The clients can be very heterogeneous. They can be laptops, smartphones, cloud services, or other smart devices. The tokens stored on these devices must be protected from theft. The clients must be authenticated during communication, and we must prevent them from forging valid tokens or modifying issued tokens. Additionally, we want to preserve the privacy of the clients.
- **Authorization server security:** The AS creates tokens for the clients after they successfully authenticate. It is our goal to block a compromised AS from obtaining the protected resources.

8.1.3 Architecture Description

We designed a new token-based access control scheme with integrated authenticated key establishment for IoT platforms. The access tokens consist of a set of encoded access rights, known as token claims, wrapped in a COSE object. Access tokens organize the claims according to the CWT (CBOR Web Token) specification [199]. The architecture is standalone and does not depend on the security of the underlying transport method. It follows the same approach to security as the original OAuth 1.0*a* framework [200]. OAuth 1.0*a* differs from OAuth 2.0 in that it defines its proper security model. There is no dependency on the properties of the underlying transport layer. OAuth 1.0*a* is deemed more secure than OAuth 2.0, but more challenging to implement [201].

Similar to ACE, the new architecture handles resource servers and with intermittent Internet connectivity by using self-contained tokens. Constrained clients with poor connection to the AS can obtain long-lived access tokens. Long-lived tokens support multiple subsequent authentications, but contrary to ACE, they are still self-contained. By integrating the security directly in the framework, it can effortlessly be deployed in complex, multi-hop environments where the underlying network stack can change and, therefore, does not always ensure secure communication. Tolerance for intermittent connectivity is particularly valuable in duty-cycled environments. We make several assumptions about the endpoints in our framework; the constrained devices are capable of lightweight asymmetric cryptography, and every device can either contact a certificate authority or has a root certificate embedded. The latter enables the different endpoints to sign messages and verify the identities behind the signatures.

8.1.3.1 Access Token Generation

Before a client can retrieve a valid access token from the AS for a protected resource, it must receive permission from the resource owner. Communication between the client, AS and resource owner, can be protect with EDHOC and OSCORE. In Chapter 3, we discussed the most prominent ACE protocol flows: the "authorization grant flow" and the "client credential flow". The former flow obtains permissions dynamically, while the latter uses statically configured rules. Since the various entities possess PKI certificates, the resource owner can grant permission based on the information included in the client certificate. Clients should of course proof ownership of the certificate by signing a challenge with the corresponding private key. With a valid permission issued by the resource owner, the client can request an access token from the AS. The AS sets the token claims according to the permission and wraps the token claims in a `COSE_sign` object, depicted in Figure 8.1. The CWT RFC defines most of the token claims used in this chapter [199]. The AS attaches its signature to the COSE object and transfers it to the client. The client verifies the signature of the AS and signs the COSE object with its private key. The token subdivides the claims into three parts: the token identity information fields, token scope fields, and token protection fields.

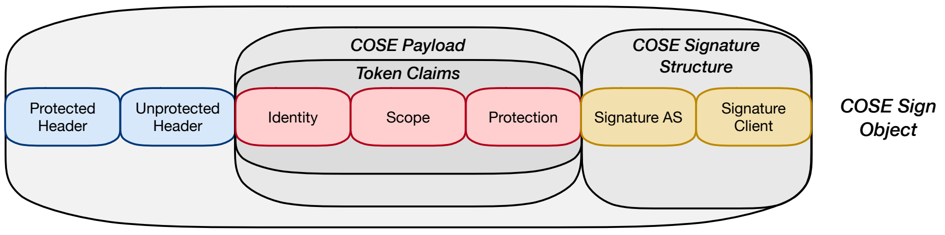


Figure 8.1: COSE wrapped access token. The `COSE_Sign` object allow for multiple signers on the object.

Token identity information field contains a *subject claim* and an *audience claim*. The former identifies a precise resource server that is the target of the particular access token, while the latter stores an identifier that points towards the clients credentials. A recent IETF draft started specifying the different attributes that can be carried inside the COSE headers to refer to X.509 certificates [202]. The attributes are:

- `x5bag`: It contains a bag of X.509 certificates.
- `x5chain`: It contains an ordered array of X.509 certificates. The chain is ordered starting with the certificate identifying the client and ends with the trusted certificate, i.e. a CA certificate.
- `x5t`: Identifies a certificate by its hash.
- `x5u`: Identifies a certificate through a URL.

Both the subject claim and audience claim contain the `x5t` attribute to identify the resource server and client, respectively. The entire certificate chains are exchanged during the EDHOC phase, see below.

The issuer claim defined in CWT is omitted in our access token model because the AS includes a key identifier or certificate attribute in the protected or unprotected header of signature structure of the enveloping COSE object.

Scope field of the token is defined by the AS based on the permission grant that was issued by the resource owner. It describes the various actions a client can perform once the resource server successfully validates the token. These actions are application-dependent. In addition, the scope includes the *expiration, not-before*, and *issued-at claims*, which store timing information on the token's validity period and the time of creation.

Token protection field includes two replay counters. The long-term replay counter is responsible for replay protection between distinct tokens acquired from the AS. The counter is incremented by the AS each time the client requests a fresh token. The AS maintains per client a long-term replay counter value that is independent of other clients. The short-term replay counter provides replay protection throughout the lifetime of a specific token. The resource server stores, per client, both replay counter values. The value of the short-term replay counter must change on every token use to achieve the PoP principle, see below. When the resource server uses the token timing information, such as the expiration, not-before, and issued-at fields, to limit the token usage, the short-term replay counter is a simple incrementing counter. If the resource server does not possess precise time-keeping hardware, the AS leaves the expiration field in the token scope blank and sets the short-term replay window to a precise value during the token's creation. The resource server then decrements the short-term window for every token iteration until the counter reaches 0. At 0, the token expires, and the client will have to request a new token at the AS.

8.1.3.2 Authenticated Access Token Exchanges

A client initiates an authenticated token exchanged by starting the EDHOC protocol, see Chapter 3. During the key, both parties use ephemeral elliptic curve keys to derive a forward secure shared secret. The exchanges are authenticated with the COSE X.509 attributes described above [202]. Since our protocol relies on PKI. The identifying information corresponds to a certificate chain, which ends in a trusted root certificate. During the third and last message of the protocol, the initiator, i.e., the client, can send data that is protected with the shared secret in parallel. We make use of this feature by instructing the client to send the fresh token together with the last message of the EDHOC protocol, see ① in Figure 8.2. When the resource server has processed the final EDHOC message, it can immediately decrypt the fresh token and process it. The resource server always processes tokens in two phases. The steps differ for tokens that are fresh and tokens that have been used before. In the first phase, the resource owner validates the fresh token as follows:

1. It verifies that the token has not expired. It either checks the token scope field, containing the timing information or the short-term and long-term counter values in the token protection field.
2. It verifies if the hash contained in the audience claim matches with the hash of the first certificate in the chain delivered in message 3 of the EDHOC protocol.
3. It verifies the signatures of the AS and the client. The signature of the AS can be verified through its static public key, embedded in the AS certificate. The AS certificate should be installed on each resource server during an enrollment phase. The client signature over the token is calculated with the same ephemeral key pair that was generated for the EDHOC protocol. Since the resource server received the public key during message 1 of the EDHOC protocol it can verify the client's signature.
4. If the resource server has no previous record on the client, the resource server uses the audience claim to create a client ID. It then stores both replay counters for this ID.

In the second phase, the resource server creates a `COSE_encrypt` object. It copies the token claims from the received `COSE_sign` object and updates the short-term replay protection window. It encrypts the payload according to the COSE specification using an AEAD cipher. The key for the encryption algorithm, K_{sh}, was established during the EDHOC protocol. Before transferring the token back to the client, the resource server calculates an HMAC over the payload of the `COSE_encrypt` object, and stores it locally under the client's ID. The resource server generates the key, K_{rs}, necessary for the HMAC locally, and the key never leaves the device. This mechanism prevents the client from tampering with the token, i.e., modifying the token claims. The resource server then sends the encrypted updated token back to the client, see ② in Figure 8.2.

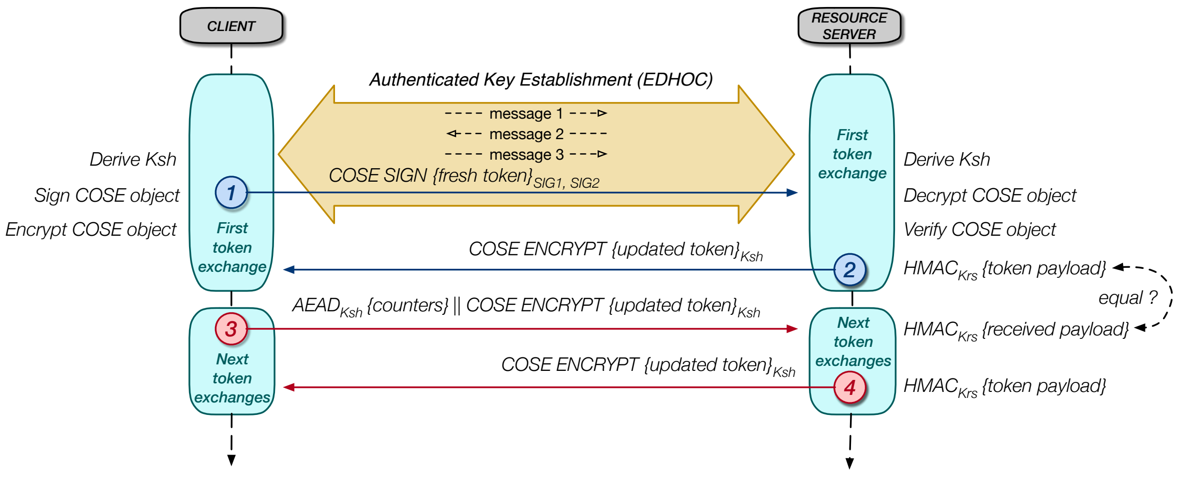


Figure 8.2: Token exchanges between the client and resource server: `SIG1` and `SIG2` correspond to the signatures of the AS and client, respectively

The initial two steps are expensive due to the asymmetric cryptography involved in verifying a fresh token and setting up of the shared secret with the EDHOC protocol. However, the verification of a fresh token bootstraps a *chain of trust* between the client and the resource server that allows them to use solely symmetric cryptography in the following token exchanges.

The client then receives the updated token. It can verify the authenticity because the token was encrypted and authenticated, through the AEAD cipher, with the shared key, K_{sh}. The shared key acts as a PoP key. In the original ACE framework, the AS performs the binding between the PoP key and the token. Here, the token is bound to the K_{sh} key by the resource server on the first use. The next time the client wants to authenticate to the resource server, it uses the same token. To proof its possession of the PoP key, the client encrypts and authenticates, with the AEAD cipher, the concatenation of the long-term and short-term replay counters which are embedded in the token. Together both values are unique for a particular client. The client sends both the encrypted token and ciphertext of the counters to the resource server, see ③.

On access token reception, resource server again uses two phases to process the token. It now executes the following steps:

1. It decrypts the token and counter information contained in the PoP message.
2. It checks if the counter information corresponds to the information in the token. If they match the client has proven it possesses the PoP key.
3. It detects token tampering by comparing a freshly calculated HMAC of the token with the previously stored HMAC.
4. It verifies the token's expiration time and counters.

If all checks are valid, the token originates from the rightful client and it has not been tampered with. Because with each token use the resource server updates the short-term replay window. Every time the client uses the access token it must generate a new PoP response and send it along.

8.1.4 Security Considerations

Eavesdropping Attacks are thwarted by the AEAD ciphers used to protect the token exchanges. Both fresh tokens and updated tokens are encrypted during transport. The fresh token is protected by the EDHOC exchanges and the updated token are explicitly protected by our protocol through `COSE_encrypt0` objects. The EDHOC protocol also safeguards the privacy of both the client and resource server.

Replay Attacks are mitigated by the long-term and short-term replay counters. The long-term counter provides protection against replay when an attacker tries to reuse a token previously issued by the AS. The long-term counter only increases, for a given client, for each new token obtained from the authorization server. The value is set by the authorization server and checked and stored by the resource server. The short-term replay counter provides protection when a client uses the same token more than once. The initial value is set in the fresh token and is then incremented or decremented by the resource server on every usage.

MITM (Man-In-The-Middle) Attacks are prevented by the use of signatures or AEAD ciphers. A fresh token carries two signatures. One from the authorization server and one from the client. The resource server uses its embedded trusted certificated to validate the identities. While exchanging the updated tokens we employ the proof-of-possession concept from ACE. The shared key K_{sh}, derived through EDHOC, is used by the client to authenticate and encrypt the updated token and to generate a unique PoP message.

Rogue Clients cannot forge tokens. A fresh token must have a valid signature from the authorization server. The integrity of updated tokens is protected by the HMAC calculated by the resource server with K_{rs}. This key is only known to the resource server. A resource server could also use a simple cryptographic hash, such as SHA-256, instead of the HMAC primitive to verify the integrity of the claims, but this would allow an adversary to perform an offline hash collision search.

Compromised Authorization Servers cannot freely generate tokens. The attacker must compromise the authorization server and a client with a valid certificate to create access tokens.

Compromised Resource Servers lose all the data and key material stored on the device. Since non of key material is shared with other resource servers or the authorization server, e.g. no network-wide symmetric keys, the impact is limited.

8.1.5 Implementation Considerations

8.1.5.1 Computational Impact and Memory Overhead

We did not provide a full implementation of the proposed architecture. However, since many of the the building blocks were separately discussed throughout this thesis, we can form a rough estimation on the computational overhead of the architecture. We refer the reader to Chapter 2 and Chapter 7 for more details on the exact impact of the computations associated with public-key cryptography and certificates.

8.1.5.2 Bandwidth Limitations

Constrained devices use communication protocols with limited packet sizes, such as IEEE 802.15.4E, i.e., 127 bytes. The size of the exchanged tokens is therefore an important factor. We implemented a COSE library in Python[1] and tested the COSE object sizes for a `COSE_sign` object and a `COSE_encrypt0` object. The former being the COSE object type being used to protect the fresh token, while the latter holds the updated token. We used example token claims for test purposes:

- **Subject claim:** Contains an `x5t` attribute referencing the resource server's certificate that is exchanged during EDHOC. Currently the IETF's draft on COSE X.509 attributes does not describe truncated hashes [202]. A tradeoff between security and payload size is possible by limiting the size of the `x5t` attribute. In this example we truncate the certificate's hash to 128 bit.
- **Audience claim:** Contains an `x5t` attribute referencing the resource server's certificate. Similarly, we truncate the hash to 128 bit.
- **Long-term replay window:** 48 bit. Counter that is incremented each time the client request a new token at the AS.
- **Short-term replay window:** 16 bit. If no timing information is set in the token scope, this field contains the total amount of allowed token uses.

[1] `https://github.com/TimothyClaeys/COSE-PYTHON`

- **Scope claims:** the encoding and size of the resource scope claims is highly dependent on the application. Here, we allocate 32 B for the claims.

The fresh token size amounts to 72 B. Next, we wrap the fresh token in a `COSE_sign` object. The COSE object is shown in Figure 8.1. The protected and unprotected headers of the enveloping COSE structure are empty. The payload of the COSE object contains the fresh token and the signature structure is a CBOR array with two signatures. Each signature has a protected and unprotected header, according to the COSE RFC. The internal headers contain the information on the used signature algorithm. In our example, both the AS signature and client signature are calculated with the deterministic ECDSA algorithm using `secp256r1` as curve. The resulting COSE signature structure amounts to 290 B. The different parts are then encoded as a CBOR array holding the enveloping headers, payload and signature structure. The size of the `COSE_sign` object is 370 B.

The updated token, uses a `COSE_encrypt0` to protect the contents. It consists of the protected and unprotected headers and the payload, authenticated and encrypted with an AEAD cipher. The different elements are again encoded as a CBOR array. The total token size now amounts to 72 B. The updated token is much smaller than the fresh access token, but it still requires 6LoWPAN fragmentation is we want to carry it over the standardized IoT stack (IEEE 802.15.4E, IPv6, UDP and CoAP), as presented in Chapter 3. If we want to prevent fragmentation in IEEE 802.15.4E networks, the `COSE_encrypt0` object cannot be larger than 62 B. Some space can be gained by using a COSE AEAD cipher with truncated authentication tags.

8.2 IoTChain: A Blockchain Architecture for the IoT

The goal of the IoTChain architecture is to decentralize the AS and in parallel, provide authenticated key establishment for an object security protocol, e.g., OSCORE. Figure 8.3 depicts the architecture. It shows the essential entities in the network and the sequence of operations leading to authorized and authenticated access of a protected IoT resource. We introduce some additional terminology:

Authorization Blockchain is a permissionless blockchain[2] used to authorize endpoints. It generates access tokens through the execution of smart contracts. Recall that a blockchain is a distributed ledger shared by all the participants. The ledger contains chained blocks, which in turn hold transactions. The consensus protocol, e.g., PoW (Proof-of-Work) or PoS (Proof-of-Stake), provides security and ensures that everybody in the network converges to the same blockchain state. A more detailed introduction to blockchain technology is provided in Chapter 3.

Key Server is an entity that manages the cryptographic material that guarantees the confidentiality and integrity of the protected resources. The key server generates and distributes the keys to the resource servers and the authorized clients. Potentially, resource servers encrypt protected resources with different keys to provide distinct access groups. In general, a single key server can maintain the keys for a set of resource servers belonging to the same resource owner. In some scenarios, the key server may coincide with the resource server, thus simplifying the key exchange procedure.

8.2.1 Authorization Blockchain

8.2.1.1 Authorization and Authentication Flow

The authorization and authentication flow is split into four phases. When it finishes, the client has successfully obtained the protected resources.

In the first phase, the resource owner creates a smart contract. The smart contract is a compiled program that generates an access token for a client when certain conditions are met. The resource owner publishes its smart contract to the blockchain, see ①.

In the second phase, a client that wishes to access a protected resource activates the corresponding smart contract ②. The client verifies that the owner of the smart contract is the resource owner of the

[2]A permissionless blockchain is public; everybody can participate.

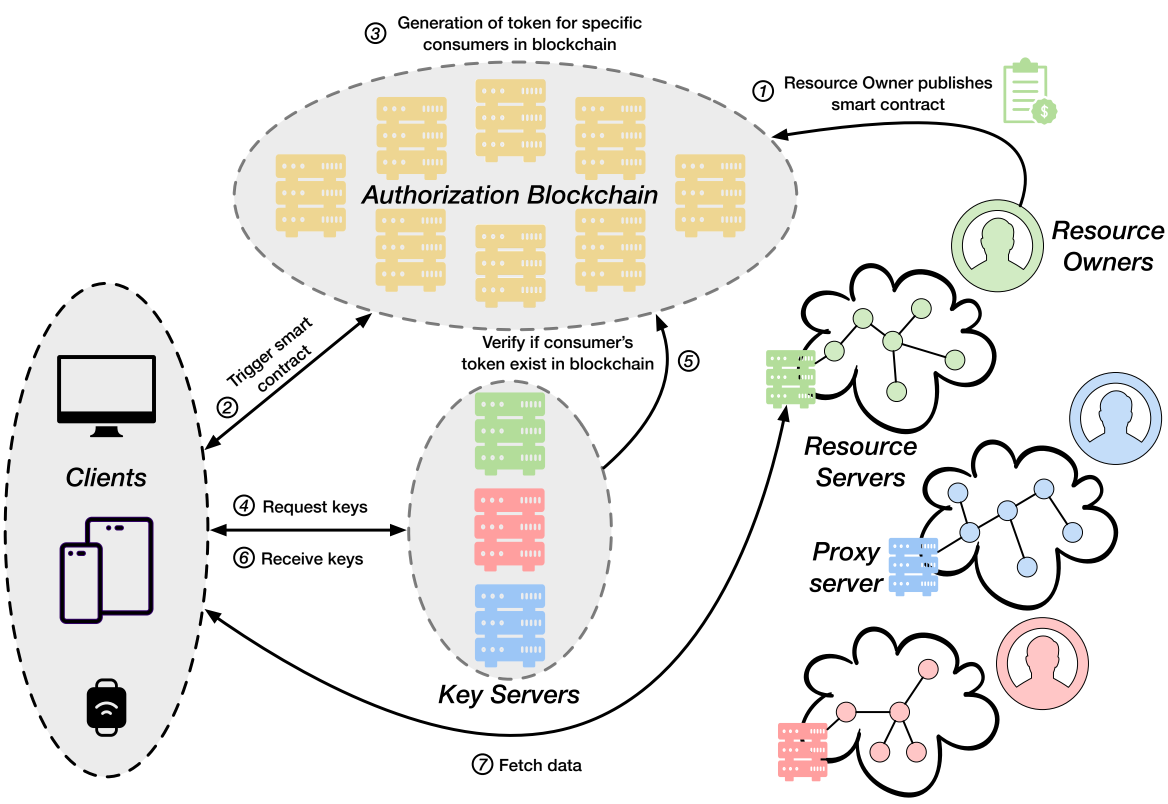


Figure 8.3: The IoTChain architectures.

protected resources it wishes to access. Verification of the resource owner's identity happens out-of-band. A client can trigger a smart contract by directing a transaction to its address. The transaction propagates through the blockchain network. Every authorization server (i.e., miner) that includes the transaction in its current block validates the transaction parameters and executes the smart contract. The transaction is only valid when the client provides the correct input data for the smart contract. For example, an energy company (the resource owner) has deployed a smart meter (the resource server) at the client's house. To authorize the clients to interact with the smart meter and extract information, the company has published a smart contract on the blockchain. The smart contract requires proof that the client lives at a specific address to execute correctly. During the execution, the contract generates an access token for the client ③. The contract then stores the token in its internal storage. The generated access token references the public key of the client. The public key will later be used to set up a PoP challenge. The token also specifies a lifetime and describes which resources can be accessed. A resource owner can deploy multiple smart contracts for the same resource server on the blockchain network. Each contract takes different input parameters and generates tokens with different privileges.

In the third phase, the client can request the cryptographic keys necessary to decrypt the protected resources ④. The key server has a copy of the blockchain but does not participate in the consensus protocol. The key server interacts with the smart contract and verifies if an access token was issued for the client ⑤. Subsequently, it checks the access rights encoded in the token. Before the key server sends the keys to the client, it generates a PoP challenge based on the public key referenced in the token. If the client successfully responds, the key server then sends the keys that give access to the corresponding protected resources, to the client over an encrypted link (DTLS, TLS or OSCORE). Only the legitimate client that triggered the smart contract and provided the public key could have solved the PoP challenge ⑥. The key server must wait for n new blocks since the creation of the token to handle scenarios where the blockchain temporarily forked. The value n is a security parameter. A large n gives the key server strong guarantees on the validity of the token but might incur a more significant latency, e.g., in the Ethereum network, 12 block confirmations are required, which corresponds to approximately 3 min latency.

During the final fourth phase, the client downloads the encrypted resources either from a proxy server or directly from the resource server ⑦. Both entities provide a Restful CoAP API that allows to `GET`, `PUT`, and POST resources based on their URI. No further authentication is necessary as an unauthorized client cannot obtain the cryptographic keys to decrypt the protected resources. When the protected resources are acquired directly from the resource server, protected resource are encrypted and integrity protected with an AEAD cipher. When the resource server publishes the protected resources to a proxy server, it should sign the encrypted resources with a private key, which prevents colluding proxy servers and clients from corrupting the protected resources. The public key to verify this signature is also distributed by the key servers.

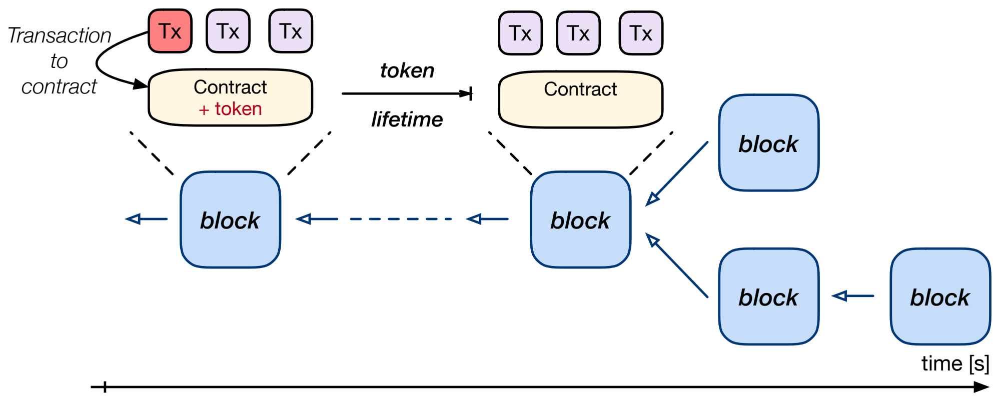


Figure 8.4: Token lifetime on the blockchain.

8.2.1.2 Adding and Revoking Entities

Adding a new client to the architecture is trivial. Since the blockchain is permissionless, no steps are required before a client can interact with the network. By providing the necessary information in a transaction to a smart contract, it generates a token that allows the new client to access the decryption keys for a specified resource. To easily revoke clients, new encryption keys should be issued frequently by the key server. While a token can be valid for an extended period, the key server can distribute new keys to the resource server daily. The resource server should always use the latest keys to encrypt new resources, e.g., new temperature measurements. As long as the client token is valid, the client can recover the new keys from the key server. When the token expires, the client must rerun the smart contract on the blockchain to generate a new valid token and obtain the access rights for the keys, see Figure 8.4. When the client misbehaves, the authorization servers in the blockchain network can add a transaction in block, which invalidates the client's access token. The key server will receive this block once it is added to the blockchain and will know the client is no longer authorized to receive the decryption keys.

Similarly to clients, compromised authorization servers can be revoked. Periodically, the blockchain authorization network adds a block to the blockchain containing the hashes of the identities of the legitimate authorization servers. In this way, a client can be sure that it sends its transaction to a valid authorization server, or a key server can be sure it downloads the blockchain from a legitimate authorization server.

8.2.2 Security Considerations

Proof-of-Possession is, similarly to ACE, used by the IoTChain architecture. In ACE, the client can provide a public key to the AS. The AS then binds the public key to the access token. Only the legitimate client has the corresponding private key, which it uses in the PoP challenge-response exchange. In the IoTChain architecture, the client that activates the smart contract provides (an ephemeral) public key. The generated token references this public key, either through a hash or by storing it directly. The key server that verifies the token then creates a challenge based on the public key.

Client Privacy is an important issue since the blockchain is permissionless and public; anyone can read the content of the blocks. Clients can protect their privacy by generating new asymmetric key pairs for each transaction. This allows the client to isolate each of its transactions. Smart contracts and by extension, the resource owners cannot see what other smart contracts the clients have activated if they use ephemeral keys.

Denial-of-Service attacks aim to prevent the regular operation of the blockchain. Because of the halting problem, we cannot predict if a smart contract deployed on the blockchain will terminate. The Ethereum blockchain solves the problem by requiring an upfront payment for the execution of a contract [77]. A user needs to estimate how much it will cost to execute the contract, and the user then pays the contract with a special currency denoted as *gas*. If the user does not provide sufficient gas, the contract will not execute entirely, and it will revert all the changes. Because we do not require a cryptocurrency in the IoTChain and the smart contracts merely generate access tokens, we can set a limit on the execution time of a smart contract.

Since there is no cryptocurrency, malicious clients are not actively disincentivized to launch a DoS (Denial-of-Service) attack on the network by continually triggering smart contracts. Since each AS broadcasts the transactions, and every AS needs to verify the transactions before executing the smart contract, the network may become saturated. A simple defense can require each client to solve a cryptographic puzzle before each transaction. A similar approach is proposed in the HIP (Host Identity Protocol) protocol [203].

Secure Communications between the key servers, resource servers and clients are provided by DTLS or OSCORE. Authentication is based on certificates or through a challenge-response exchange, i.e., between the key server and the client. Transactions sent on the blockchain network have signatures to protect their integrity, but they are sent in clear. The blockchain is a public ledger, so all the transactions can be read. Clients should, therefore, never put confidential information in the transactions.

8.2.3 Implementation on the Ethereum Private Testnet

To evaluate feasibility of our architecture, we implement the authorization blockchain on top of a private Ethereum blockchain network. We use the go-ethereum implementation[3] of the protocol to set up our network. The implementation turns a device into a full node in the Ethereum network. It also provides the possibility to connect to the Ethereum testnet. The testnet lets developers test and debug smart contracts without spending real cryptocurrencies, i.e., *Ether*. Because syncing with the entire Ethereum testnet would take too much time and resources, we decide to mount a local private Ethereum blockchain. To lower the difficulty of the PoW algorithm currently used in Ethereum, we define a custom genesis block for the blockchain. The blockchain network consists of five nodes, maintaining the local blockchain, see Figure 8.5.

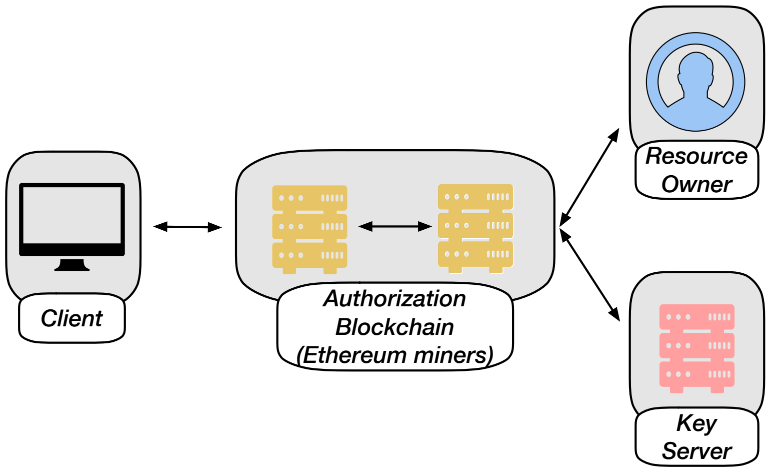


Figure 8.5: Experimental Ethereum testnet.

The resource owner deploys the smart contract shown in Figure 8.1. Once the contract is added to the blockchain, clients can interact with it by calling its individual public functions. The function `addToken` creates a new access token for a client.

[3]The Ethereum foundation provides an opensource golang implementation of the Ethereum protocol (https://github.com/ethereum/go-ethereum)

```
pragma solidity ^0.4.0;

contract AccessToken{
    mapping (address => Token) issued;

    struct Token{
        address res;
        uint ttl;
        bytes4 claims;
    }

    function addToken(...) public returns(bool){
        issued[clt].res = _res;
        issued[clt].ttl = _ttl;
        issued[clt].claims = _claims;
        return true;
    }

    function getToken(...) public constant returns(...){
        return (issued[clt].res, issued[clt].ttl, issued[clt].claims);
    }

    function deleteToken(...) public returns(bool){
        delete(issued[clt]);
    }
}
```

Listing 8.1: Smart contract for access token

The access token is stored in the contract's persistent memory. The token contains the client address and the resource server address. The `ttl` field of the access token holds a life time parameter, checked by the key server when the client requests a decryption key. The `deleteToken` function might be called to revoke the client access rights earlier than foreseen. The smart contract in Fig. 7 can be triggered by everyone in the blockchain network. More complex contracts may require modifiers that restrict access to certain functions or check if a condition is met before the function is executed by the Ethereum Virtual Machine (EVM), e.g., has this client paid for the access token.

Conclusion

In the final chapter of this thesis we discussed authorization and authentication architectures for the IoT. In both of our proposals the idea is to make the existing ACE infrastructure more robust by preventing an compromised AS, jeopardizes the entire network.

In the first proposal, we make use of the PKI infrastructure to remove the role of trusted third party from authorization server. Instead of relying on the protection mechanisms offered by the underlying transport protocols, we propose the use of self-protecting tokens, in line with the security model proposed by OAuth 1.0a. In addition, the tokens are designed to be reusable. The initial token exchange is expensive due to its large size and the asymmetric cryptography associated with PKI functionalities. Subsequent token exchanges capitalize on trust anchor installed by the asymmetric signatures to provide authenticated access to the protected resources with solely symmetric cryptography. The updated access tokens are much smaller and with further optimizations they could be carried over a IEEE 802.15.4E without requiring 6LoWPAN fragmentation. All token exchanges are encrypted and by taking advantage of the EDHOC properties, it provides privacy for the clients, i.e., an eavesdropper does not learn which protected resources a client can access.

The second proposal combines blockchain technology with the ACE framework. A blockchain network implements the role of the AS. The authorization requests are treated by smart contracts deployed by the resource owners. To gain access to a protected resource, a client issues a transaction to the address of a smart contract. Upon validation of the transactions parameters, the contract generates a token which it stores in it local storage. Key servers can interrogate the access token store through an API offered by the smart contract. The tokens reference the client's public key which is subsequently used to set up a PoP challenge. In the final stage the client obtains the keys necessary to access the protected resources.

Conclusions

The work presented in this manuscript was conducted in the context of the IoTize project. The goal of this project was to develop a simple turnkey solution to extend embedded systems with two functionalities: a human-machine interface and Internet connectivity. To guide the IoTize development team in building a secure product, we studied the state of the art of IoT security.

In the first part of this thesis, we identified various building blocks that are vital for the security of the IoT. For each block, we analyzed the emerging trends but also listed the liabilities and weaknesses. We moreover briefly summarize the essential takeaways.

Firstly, we pointed out that hardware-enforced security mechanisms are a necessity at the hardware layer: without memory isolation, a single vulnerability suffices to compromise the entire system. Alternative approaches focus on detection instead of prevention, they often rely on strong assumptions about the attacker's capabilities and, therefore, cannot guarantee the same security principles.

Cryptography represents the second cornerstone of IoT security. The conventional cryptographic toolbox provides many secure primitives, but few have an acceptable performance on constrained hardware. As a result, the CAESAR cryptographic competition elected two novel, lightweight symmetric algorithms. We highlighted their performance gain with respect to traditional primitives.

A secure and energy-efficient networking stack is the third pillar for a secure IoT. Its purpose is to protect the vast quantities of data that will be communicated by the constrained devices. The consensus is that encryption should mostly happen on the application layer since this security model best fits the typical IoT use cases. As a result, the IETF is developing new object security protocols that depend on highly-optimized encoding schemes. However, we showed that the push towards an efficient network stack introduced new vulnerabilities, mainly below the network layer, e.g., 6LoWPAN and IEEE 802.15.4e. The studied vulnerabilities are intrinsic to the operation of the protocols and expose the constrained devices to potent DoS attacks.

Finally, the different security architectures guide the secure integration of the IoT with the traditional Internet. We presented the latest developments of the IETF working group on the ACE framework for constrained devices. Their design is highly inspired by OAuth 2.0 and thus exhibits the same weaknesses. Security entirely depends on the communication protocol and the use of trusted third parties.

Summary of the contributions

The second part of the thesis presented our contributions. They aim to improve the security of the IoT globally, and thus they target the different building blocks.

Chapter 5 contains the first contribution. We studied the application of physical fingerprinting techniques in the constrained IoT. By combining the unique identifiers with software-only remote attestation procedures, we strive to improve the security guarantees of the protocol. We split the possible identifiers into two categories: signal-based identifiers and PUF-based identifiers. We derived an initial signal-based fingerprint from the natural clock drift between communicating devices. Unfortunately, the clock drift is unstable when the ambient temperature changes, and it does not provide sufficient entropy to identify a device uniquely among a large group.

We then presented a second identifying protocol built on a Sybil attack detection algorithm. We extended the functionality of the original algorithm and capitalized on the different frequency channels that are available to channel hopping communication protocols. Our proposed algorithm can reliably detect when transmissions originate from distinct locations by comparing RSS measurements over different frequency channels, hence detecting changes in the original position of the transmissions, but does not identify a given device in mobile scenarios. This approach is thus only useable when the devices use fixed pre-defined locations.

In the context of PUF-based identification, we analyzed the SRAM start-up behavior. We studied the inter- and intra-device start-up values and concluded they contain sufficient entropy to generate a unique identifier. Besides, we showed the start-up values are stable under varying ambient conditions. Nonetheless, without memory protection, an attacker can extract the fingerprint from a compromised device and thus spoof its identity.

We thus concluded that none of the existing fingerprinting techniques generate a reliable identifier. Still, in scenarios where IoT devices are entirely compromised, they provide a primary identification mechanism where otherwise none is available.

Chapter 6 details the second contribution. Throughout the chapter we observed that the optimization techniques applied to low-power link layer protocols of the IoT networking stack introduced potential vulnerabilities.

We presented a first vulnerability inherent to the TSCH protocol. TSCH relies on tight time synchronization and channel hopping. While these features reduce energy consumption and provide robust communications, they complicate the bootstrap phase of the network: nodes spend a disproportionate amount of energy during the network join phase. By temporarily jamming the wireless link between two well-chosen devices, an attacker can force the desynchronization of a large part of the network. Consequently, the desynchronized devices reactivate the costly join phase. We designed a fast and energy-efficient algorithm to mitigate this attack. It uses drift prediction combined with static network information to rejoin the network at a lower cost than the classical joining algorithm. Experimental results showed that the proposed solution minimizes join latency while reducing energy consumption with a factor 1000 compared to existing proposals.

The second vulnerability exploits the highly-predictive behavior of time-synchronized networks. Due to strict synchronization among nodes, an external entity can easily predict the time instants where communications will occur. An attacker that can introduce small offsets in the timings of the transmission can use this to build a covert channel. The time-synchronized protocols do not detect such additional offsets because they compensate for naturally occurring clock drift.

We implemented a proof-of-concept of a covert channel on several BLE-enabled devices. Through heat emission from the CPU, we influenced the frequency of the crystal oscillator, which in turn affected the timing of the transmissions. The performance of the attack strongly depends on the hardware layout of the BLE device: devices that have a direct thermal path between the CPU and the crystal oscillator obtained a significantly higher throughput over the covert channel.

Chapter 7 presents the third contribution. It studies the performance of the existing security protocols over constrained networks. We provided an in-depth analysis of the most critical part of the TLS and DTLS protocols: the handshake phase. At first, we present a detailed discussion of the various problems that might occur when carrying large transport layer packets over a low-rate, low-power network. Next, we conducted an experimental evaluation by porting the mbed TLS library on top of the OpenWSN, a TSCH-based networking stack. We additionally implemented TCP and the 6LoWPAN fragmentation mechanism and integrated both with OpenWSN.

The main takeaway is that while the DTLS handshake provides better performance when the communication link is 100% reliable, its performance quickly deteriorates when packet losses occur. The many tweakable parameters and optimization of TCP allow better control of the protocol behavior, which lowers the overhead of retransmission over lossy networks.

Chapter 8 covers the final contribution. It tackles security architectures for the IoT. More precisely, we proposed two adaptations of the upcoming ACE architecture.

In the first adaptation, we leveraged the existing PKI infrastructure to remove the role of a trusted third party from the authorization server. We propose to adopt the new COSE message format to provide self-securing and self-contained tokens. We effectively decoupled the security of the framework from the underlying transport protocols by relying on EDHOC to provide an authenticated key establishment. We designed reusable tokens to alleviate the cost of the expensive cryptographic operations associated with PKI: the initial token exchange bootstraps a trust anchor that allows us to use only symmetric cryptography for the subsequent token exchanges.

The second adaptation depicted an end-to-end security architecture that combines blockchain technology with the ACE framework. Similar to the previous proposal, we aimed to remove any trusted third-parties by substituting the authorization server with an authorization blockchain. The blockchain handles authorization requests through smart contracts.

Future work

Throughout this thesis, we have treated several aspects related to IoT security. This approach allowed us to provide a comprehensive overview of the current state of the art, propose improvements, and identify compelling future research directions. We list these perspectives in a bottom-up fashion.

With the emergence of hardware security in commercial chipsets, the new generation of IoT devices will be better equipped to mitigate software attacks. However, for numerous reasons such as production cost, energy restrictions, and compatibility issues, not all constrained devices will be supplied with these advanced security features. For this class of systems, remote attestation protocols remain a crucial line of defense. Although much of the existing protocols have been broken, the ideas behind the designs are still valid. It seems, therefore, necessary to further advance this field of research.

To provide a stronger foundation and more robust security guarantees for future protocols, Eldefrawy et al. [138] propose to utilize computer-aided formal verification. Complimentary research on physical device identifiers can further help localize malicious activity in a constrained network. We believe that a protocol combining both physical identifiers and formal verification can actively help protect the network by performing targeted attestation and prevent the propagation of malware through low-power devices.

Presumably, in the upcoming decades, the advances in quantum computing will break public-key cryptography based on integer factorization and the DLP. In response, NIST has issued a competition on quantum-resistant algorithms [59]. Even though lattice-based primitives show promising results on constrained hardware, additional efforts concerning the constrained IoT are required. More precisely, the current propositions do not account for side-channel resistance, an increasingly important security characteristic for the IoT [62, 204].

Furthermore, many of the future IoT applications will rely on the OSCORE protocol to provide secure end-to-end communication. However, the OSCORE RFC merely describes how to transform a CoAP message to an OSCORE message. Mechanisms for secure key establishment and constrained endpoint authentication are still under development.

Currently, the IETF is considering two protocols for key establishment. The first proposal, called EDHOC, was extensively described in this thesis. It uses a highly-optimized Diffie-Hellman key exchange protocol. EDHOC leverages the COSE message format to minimize the message overhead. The second proposal, cTLS (compact TLS), is being developed by the TLS working group [205]. It consists of a condensed version of TLS 1.3, designed to take up minimal bandwidth. The main advantage of cTLS is that it can rely on the security of the TLS protocol, and can quickly gain support as the deployment of TLS 1.3 continues.

Due to memory constraints, IoT devices will not provide both implementations. As a result, it seems necessary to perform a study of both key exchange mechanisms and clearly distinguish their use cases.

Both EDHOC and cTLS rely on either pre-shared keys, raw public keys, or certificates to provide endpoint authentication. On the one hand, the first two methods can provide efficient mutual authentication, since they only require minimal bandwidth during the authentication phase. However, it is not feasible to embedded pre-shared keys and raw public keys in constrained devices at the scale of the IoT. On the other hand, the current certificate infrastructure requires a large bandwidth. The individual certificates are too large for use in constrained networks. Henceforth, we currently lack a scalable authentication mechanism dedicated to the constrained IoT.

Bibliography

[1] Gregory D Abowd, Liviu Iftode, and Helena Mitchell. Guest Editors' Introduction: The Smart Phone – A First Platform for Pervasive Computing. *IEEE Pervasive Computing*, 4(2):18–19, 2005. 1

[2] Malisa Vucinic. *Architectures and Protocols for Secure and Energy-Efficient Integration of Wireless Sensor Networks with the Internet of Things*. PhD thesis, 2015. URL http://www.theses.fr/2015GREAM084. Doctoral thesis, guided by Tourancheau, Bernard Rousseau, Franck et Damon, Laurent, Informatique Grenoble Alpes 2015. 1, 12, 14, 59, 60

[3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security '17*, pages 1093–1110, 2017. 1

[4] Saleh Soltan, Prateek Mittal, and H Vincent Poor. BlackIoT: IoT Botnet of High Wattage Devices Can Disrupt the Power Grid. In *27th USENIX Security Symposium (USENIX Security '18)*, pages 15–32, 2018. 1

[5] Joseph Yiu. *The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors*. Academic Press, 2015. 2, 11, 12

[6] The Noun Project, 2019. URL https://thenounproject.com. 2

[7] Goeran Selander, John Mattsson, Francesca Palombini, and Ludwig Seitz. Object Security for Constrained RESTful Environments (OSCORE). Internet-Draft draft-ietf-core-object-security-15, IETF Secretariat, August 2018. URL http://www.ietf.org/internet-drafts/draft-ietf-core-object-security-15.txt. http://www.ietf.org/internet-drafts/draft-ietf-core-object-security-15.txt. 4, 6, 54, 55

[8] Analog Devices. SmartMesh Wireless For Tough Industrial IoT Applications, 2019. URL https://www.analog.com/en/applications/technology/smartmesh-pavilion-home.html#. 10

[9] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer SJ Pister. Smart dust: Communicating with a Cubic-Millimeter Computer. *Computer*, 34(1):44–51, 2001. 10

[10] Li Da Xu, Wu He, and Shancang Li. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, 2014. 10

[11] Pete Beckman, Rajesh Sankaran, Charlie Catlett, Nicola Ferrier, Robert Jacob, and Michael Papka. Waggle: An Open Sensor Platform for Edge Computing. In *2016 IEEE SENSORS*, pages 1–3. IEEE, 2016. 10

[12] Elodie Morin, Mickael Maman, Roberto Guizzetti, and Andrzej Duda. Comparison of the Device Lifetime in Wireless Networks for the Internet of Things. *IEEE Access*, 5:7097–7114, 2017. 10

[13] C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. RFC 7228, RFC Editor, May 2014. URL http://www.rfc-editor.org/rfc/rfc7228.txt. http://www.rfc-editor.org/rfc/rfc7228.txt. 10

[14] *A Powerful System-On-Chip for 2.4-GHz IEEE 802.15.4-2006 and ZigBee Applications.* Texas Instruments, 12 2012. 12, 14, 35, 69, 90, 101, 124

[15] *ATmega8A: AVR 8-bit Microcontroller.* Microchip Technology Inc., 2017. 12

[16] *MSP430F16x: 16-bit Ultra-Low-Power MCU, 48kB Flash, 10240B RAM, 12-Bit ADC, Dual DAC, 2 USART, I2C, HW Mult, DMA.* Texas Instruments, 10 2002. 12

[17] *ARM Cortex-M4 32b MCU+FPU, 105 DMIPS, 512KB Flash/96KB RAM, 11 TIMs, 1 ADC, 11 comm. interfaces.* STMicroelectronics, 01 2015. 12, 42, 90

[18] *2.4 GHz IEEE802.15.4/Zigbee RF Transceiver.* Texas Instruments, 12 2007. 14

[19] *2.4 GHz IEEE802.15.4/Zigbee RF Transceiver.* Texas Instruments, 12 2013. 14

[20] *Low Power 2.4 GHz Transceiver for ZigBee, IEEE 802.15.4, 6LoWPAN, RF4CE, SP100, WirelessHART, and ISM Applications.* AVR, 09 2009. 14

[21] Ana Bildea, Olivier Alphand, Franck Rousseau, and Andrzej Duda. Link Quality Metrics in Large Scale Indoor Wireless Sensor Networks. In *2013 IEEE 24th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1888–1892. IEEE, 2013. 14, 85

[22] Elaine Barker. Recommendation for key management part 1: General (revision 4). *NIST special publication*, 800(57):1–160, January 2016. 18, 24, 39

[23] Lars R Knudsen and Matthew Robshaw. *The Block Bipher Companion.* Springer Science & Business Media, 2011. 18, 30

[24] Altus Metrum. ChoasKey, 2018. URL https://altusmetrum.org/ChaosKey/. 19

[25] Werner Schindler and Wolfgang Killmann. Evaluation Criteria for True (Physical) Random Number Generators used in Cryptographic Applications. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 431–449. Springer, 2002. 19

[26] Yan Yan, Elisabeth Oswald, and Theo Tryfonas. Cryptographic Randomness on a CC2538: A Case Study. In *2016 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–6. IEEE, 2016. 19

[27] Dan Boneh and Victor Shoup. A Graduate Course in Applied Cryptography. *Draft of a book, version 0.3, December*, 2016. 20, 26, 30, 41

[28] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: Exploiting the SSL 3.0 Fallback. *Security Advisory*, September 2014. 21, 34, 58

[29] Hanno Böck, Juraj Somorovsky, and Craig Young. Return Of Bleichenbacher's Oracle Threat (ROBOT). In *27th USENIX Security Symposium (USENIX Security '18)*, pages 817–849, 2018. 58

[30] Nadhem J Al Fardan and Kenneth G Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013. 21

[31] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman and Hall/CRC, 2014. 21, 23, 24, 38

[32] Joe Hurd. Verification of the Miller–Rabin Probabilistic Primality Test. *The Journal of Logic and Algebraic Programming*, 56(1-2):3–21, 2003. 24

[33] Victor Shoup. *A Computational Introduction to Number Theory and Algebra.* Cambridge university press, 2009. 24

[34] Matthew Green. Why I hate CBC-MAC, 2013. URL https://blog.cryptographyengineering.com/2013/02/15/why-i-hate-cbc-mac/. 30

[35] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, RFC Editor, June 2018. 32

[36] Daniel J Bernstein. The Poly1305-AES Message-Authentication Code. In *International Workshop on Fast Software Encryption*, pages 32–49. Springer, 2005. 32

[37] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). RFC 7905, RFC Editor, June 2016. 32

[38] Frederick Mosteller. Understanding the Birthday Problem. In *Selected Papers of Frederick Mosteller*, pages 349–353. Springer, 2006. 33

[39] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor, February 1997. URL http://www.rfc-editor.org/rfc/rfc2104.txt. http://www.rfc-editor.org/rfc/rfc2104.txt. 33

[40] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. 2004. 34

[41] Hequn Chen and Christof Paar. Authenticated Encryption Modes of Block Ciphers, Their Security and Implementation Properties, 2009. 34

[42] Doug Whiting, Russ Housley, and Niels Ferguson. Counter with CBC-MAC (CCM). RFC 3610, September 2003. URL https://rfc-editor.org/rfc/rfc3610.txt. 34

[43] René Struik. Formal Specification of the CCM* Mode of Operation. *IEEE P802*, 15, 2005. 34

[44] Petr Švenda. Basic Comparison of Modes for Authenticated-Encryption (IAPM, XCBC, OCB, CCM, EAX, CWC, GCM, PCFB, CS). URL https://www.fi.muni.cz/~xsvenda/docs/AE_comparison_ipics04.pdf. 35

[45] T. Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, IETF, August 2013. URL http://www.rfc-editor.org/rfc/rfc6979.txt. http://www.rfc-editor.org/rfc/rfc6979.txt. 40

[46] Leonard Adleman. A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography. In *20th Annual Symposium on Foundations of Computer Science*, pages 55–60. IEEE, 1979. 40

[47] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K Lenstra, Emmanuel Thomé, Joppe W Bos, Pierrick Gaudry, Alexander Kruppa, Peter L Montgomery, Dag Arne Osvik, et al. Factorization of a 768-bit RSA modulus. In *Annual Cryptology Conference*, pages 333–350. Springer, 2010. 40

[48] René Schoof. Elliptic Curves over Finite Fields and the Computation of Square Roots mod p. *Mathematics of computation*, 44(170):483–494, 1985. 40

[49] Hongjun Wu. ACORN:A Lightweight Authenticated Cipher (v3). Technical report, Division of Mathematical Sciences Nanyang Technological University, 09 2016. 42

[50] Florian Mendel Christoph Dobraunig, Maria Eichlseder and Martin Schläffer. Ascon v1.2 Submission to the CAESAR Competition. Technical report, Institute for Applied Information Processing and Communications, Graz University of Technology and Infineon Technologies Austria AG, 09 2016. 42

[51] ARM. ARM MBEDTLS, 2019. URL https://tls.mbed.org/. 42, 116

[52] Vampire and ECRYPTII. SUPERCOP, 01 2019. URL https://bench.cr.yp.to/supercop.html. 42

[53] NIST. Cryptographic Algorithm Validation Program. URL https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program/. 42

[54] Daniel J Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006. 44

[55] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, January 2016. URL https://rfc-editor.org/rfc/rfc7748.txt. 44

[56] Matthew Green. The Many Flaws of Dual_EC_DRBG), . URL https://blog.cryptographyengineering.com/2013/09/18/the-many-flaws-of-dualecdrbg/. 44

[57] Daniel J Bernstein, Tanja Lange, and Ruben Niederhagen. Dual EC: A Standardized Back Door. In *The New Codebreakers*, pages 256–281. Springer, 2016. 44

[58] Jerome Solinas and David E. Fu. Elliptic Curve Groups modulo a Prime (ECP Groups) for IKE and IKEv2. RFC 5903, June 2010. URL https://rfc-editor.org/rfc/rfc5903.txt. 44

[59] Lily Chen, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on Post-Quantum Cryptography*. US Department of Commerce, National Institute of Standards and Technology, 2016. 45, 147

[60] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994. 45

[61] Santosh Ghosh, Rafael Misoczki, and Manoj R. Sastry. Lightweight Post-Quantum-Secure Digital Signature Approach for IoT Motes. Cryptology ePrint Archive, Report 2019/122, 2019. https://eprint.iacr.org/2019/122. 45

[62] Zhe Liu, Kim-Kwang Raymond Choo, and Johann Grossschadl. Securing Edge Devices in the Post-Quantum Internet of Things using Lattice-based Cryptography. *IEEE Communications Magazine*, 56 (2):158–162, 2018. 45, 147

[63] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, RFC Editor, September 2007. URL http://www.rfc-editor.org/rfc/rfc4944.txt. http://www.rfc-editor.org/rfc/rfc4944.txt. 48, 59

[64] Maria Rita Palattella, Nicola Accettura, Xavier Vilajosana, Thomas Watteyne, Luigi Alfredo Grieco, Gennaro Boggia, and Mischa Dohler. Standardized Protocol Stack for the Internet of (Important) Things. *IEEE communications surveys & tutorials*, 15(3):1389–1406, 2013. 48, 54

[65] Danny Dolev and Andrew Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. 49, 66

[66] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, RFC Editor, October 2012. URL http://www.rfc-editor.org/rfc/rfc6749.txt. http://www.rfc-editor.org/rfc/rfc6749.txt. 49

[67] M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519, RFC Editor, May 2015. URL http://www.rfc-editor.org/rfc/rfc7519.txt. http://www.rfc-editor.org/rfc/rfc7519.txt. 50

[68] Daniel Fett, Ralf Küsters, and Guido Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1204–1215. ACM, 2016. 50, 51

[69] T. Lodderstedt, M. McGloin, and P. Hunt. OAuth 2.0 Threat Model and Security Considerations. RFC 6819, RFC Editor, January 2013. 51

[70] A. Popov, M. Nystroem, D. Balfanz, and J. Hodges. The Token Binding Protocol Version 1.0. RFC 8471, RFC Editor, October 2018. 51

[71] C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. RFC 7228, RFC Editor, May 2014. URL http://www.rfc-editor.org/rfc/rfc7228.txt. http://www.rfc-editor.org/rfc/rfc7228.txt. 51

[72] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, RFC Editor, June 2014. URL http://www.rfc-editor.org/rfc/rfc7252.txt. http://www.rfc-editor.org/rfc/rfc7252.txt. 51, 53

[73] C. Bormann and P. Hoffman. Concise binary object representation (cbor). RFC 7049, RFC Editor, October 2013. 51

[74] Francesca Palombini, Ludwig Seitz, Goeran Selander, and Martin Gunnarsson. OSCORE profile of the Authentication and Authorization for Constrained Environments Framework. Internet-Draft draft-ietf-ace-oscore-profile-07, IETF Secretariat, February 2019. URL http://www.ietf.org/internet-drafts/draft-ietf-ace-oscore-profile-07.txt. http://www.ietf.org/internet-drafts/draft-ietf-ace-oscore-profile-07.txt. 52

[75] Stefanie Gerdes, Olaf Bergmann, Carsten Bormann, Goeran Selander, and Ludwig Seitz. Datagram Transport Layer Security (DTLS) Profile for Authentication and Authorization for Constrained Environments (ACE). Internet-Draft draft-ietf-ace-dtls-authorize-05, IETF Secretariat, October 2018. URL http://www.ietf.org/internet-drafts/draft-ietf-ace-dtls-authorize-05.txt. http://www.ietf.org/internet-drafts/draft-ietf-ace-dtls-authorize-05.txt. 52

[76] Satoshi Nakamoto et al. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. 53

[77] Gavin Wood et al. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014. 53, 141

[78] Zhetao Li, Jiawen Kang, Rong Yu, Dongdong Ye, Qingyong Deng, and Yan Zhang. Consortium Blockchain for Secure Energy Trading in Industrial Internet of Things. *IEEE transactions on industrial informatics*, 14(8):3690–3700, 2017. 53

[79] Haipeng Yao, Tianle Mai, Jingjing Wang, Zhe Ji, Chunxiao Jiang, and Yi Qian. Resource Trading in Blockchain-based Industrial Internet of Things. *IEEE Transactions on Industrial Informatics*, 2019.

[80] Ramon Alcarria, Borja Bordel, Tomás Robles, Diego Martín, and Miguel-Ángel Manso-Callejo. A Blockchain-Based Authorization System for Trustworthy Resource Monitoring and Trading in Smart Communities. *Sensors*, 18(10):3561, 2018. 53

[81] Diana Smetters and Van Jacobson. Securing network content. Technical report, Citeseer, 2009. 54

[82] Mališa Vučinić, Bernard Tourancheau, Franck Rousseau, Andrzej Duda, Laurent Damon, and Roberto Guizzetti. OSCAR: Object security architecture for the Internet of Things. *Ad Hoc Networks*, 32:3–16, 2015. 54, 116

[83] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman. Specification for DNS over Transport Layer Security (TLS). RFC 7858, RFC Editor, May 2016. 54

[84] MultiMedia LLC. Enable Private DNS with 1.1.1.1 on Android 9 Pie, 2018. URL https://blog.cloudflare.com/enable-private-dns-with-1-1-1-1-on-android-9-pie/.

[85] P. Hoffman and P. McManus. DNS Queries over HTTPS (DoH). RFC 8484, RFC Editor, October 2018. 54

[86] Nagendra Modadugu and Eric Rescorla. The Design and Implementation of Datagram TLS. In *NDSS*, 2004. 54, 58, 63

[87] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033, RFC Editor, March 2005. URL http://www.rfc-editor.org/rfc/rfc4033.txt. http://www.rfc-editor.org/rfc/rfc4033.txt. 54

[88] B. Ramsdell and S. Turner. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751, RFC Editor, January 2010. URL http://www.rfc-editor.org/rfc/rfc5751.txt. http://www.rfc-editor.org/rfc/rfc5751.txt. 54

[89] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880, RFC Editor, November 2007. URL http://www.rfc-editor.org/rfc/rfc4880.txt. http://www.rfc-editor.org/rfc/rfc4880.txt. 54

[90] J. Schaad. CBOR Object Signing and Encryption (COSE). RFC 8152, RFC Editor, July 2017. 55

[91] Goeran Selander, John Mattsson, and Francesca Palombini. Ephemeral Diffie-Hellman Over COSE (EDHOC). Internet-Draft draft-selander-ace-cose-ecdhe-11, IETF Secretariat, January 2019. URL http://www.ietf.org/internet-drafts/draft-selander-ace-cose-ecdhe-11.txt. http://www.ietf.org/internet-drafts/draft-selander-ace-cose-ecdhe-11.txt. 56

[92] Hugo Krawczyk. SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols. In *Annual International Cryptology Conference*, pages 400–425. Springer, 2003. 56

[93] Alessandro Bruni, Thorvald Sahl Jørgensen, Theis Grønbech Petersen, and Carsten Schürmann. Formal Verification of Ephemeral Diffie-Hellman over COSE (EDHOC). In *International Conference on Research in Security Standardisation*, pages 21–36. Springer, 2018. 56

[94] Jon Postel. Transmission Control Protocol. STD 7, RFC Editor, September 1981. URL http://www.rfc-editor.org/rfc/rfc793.txt. http://www.rfc-editor.org/rfc/rfc793.txt. 57

[95] J. Postel. User Datagram Protocol. STD 6, RFC Editor, August 1980. URL http://www.rfc-editor.org/rfc/rfc768.txt. http://www.rfc-editor.org/rfc/rfc768.txt. 57

[96] CG Carles Gomez, Andres Emilio Arcia Moret, and Jonathon Andrew Crowcroft. TCP in the Internet of Things: From Ostracism to Prominence. February 2017. 57

[97] Ahmed Ayadi, David Ros, and Laurent Toutain. Tcp header compression for 6lowpan. Internet-Draft draft-aayadi-6lowpan-tcphc-01, IETF Secretariat, October 2010. URL http://www.ietf.org/internet-drafts/draft-aayadi-6lowpan-tcphc-01.txt. http://www.ietf.org/internet-drafts/draft-aayadi-6lowpan-tcphc-01.txt. 57, 130

[98] Carles Gomez, Jon Crowcroft, and Michael Scharf. TCP Usage Guidance in the Internet of Things (IoT). Internet-Draft draft-ietf-lwig-tcp-constrained-node-networks-08, Internet Engineering Task Force, June 2019. URL https://datatracker.ietf.org/doc/html/draft-ietf-lwig-tcp-constrained-node-networks-08. Work in Progress. 57, 123

[99] Adam Dunkels, Juan Alonso, and Thiemo Voigt. Making TCP/IP Viable for Wireless Sensor Networks, 2003. 57

[100] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014. 57

[101] Adam Langley. PKCS#1 Signature Validation, . URL https://www.imperialviolet.org/2014/09/26/pkcs1.html. 57

[102] Adam Langley. Apple's SSL/TLS bug, . URL https://www.imperialviolet.org/2014/02/22/applebug.html. 57

[103] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH. In *Network and Distributed System Security Symposium – NDSS 2016*, San Diego, United States, February 2016. doi: 10.14722/ndss.2016.23418. URL https://hal.inria.fr/hal-01244855. 58

[104] Luke Valenta, Nick Sullivan, Antonio Sanso, and Nadia Heninger. In Search of CurveSwap: Measuring Elliptic Curve Implementations in the Wild. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 384–398. IEEE, 2018. 58

[105] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *22nd ACM Conference on Computer and Communications Security*, October 2015. 58

[106] Matthew Green. Attack of the week: FREAK (or 'factoring the NSA for fun and profit'), . URL https://blog.cryptographyengineering.com/2015/03/03/attack-of-week-freak-or-factoring-nsa/. 58

[107] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 456–467. ACM, 2016. 58

[108] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. URL https://rfc-editor.org/rfc/rfc8446.txt. 58

[109] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, January 2012. URL https://rfc-editor.org/rfc/rfc6347.txt. 59, 116

[110] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-dtls13-32, Internet Engineering Task Force, July 2019. URL https://datatracker.ietf.org/doc/html/draft-ietf-tls-dtls13-32. Work in Progress. 59

[111] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. Security for the Internet of Things: a Survey of Existing Protocols and Open Research Issues. *IEEE Communications Surveys & Tutorials*, 17(3): 1294–1312, 2015. 59

[112] Carsten Bormann, Zach Shelby, Samita Chakrabarti, and Erik Nordmark. Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 6775, November 2012. URL https://rfc-editor.org/rfc/rfc6775.txt. 59

[113] René Hummen, Jens Hiller, Hanno Wirtz, Martin Henze, Hossein Shafagh, and Klaus Wehrle. 6LoWPAN Fragmentation Attacks and Mitigation Mechanisms. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 55–66. ACM, 2013. 59, 60

[114] Aminul Haque Chowdhury, Muhammad Ikram, Hyon-Soo Cha, Hassen Redwan, SM Shams, Ki-Hyung Kim, and Seung-Wha Yoo. Route-over vs Mesh-under Routing in 6LoWPAN. In *Proceedings of the 2009 international conference on wireless communications and mobile computing: Connecting the world wirelessly*, pages 1208–1212. ACM, 2009. 59

[115] Yasuyuki Tanaka, Pascale Minet, and Thomas Watteyne. 6LoWPAN Fragment Forwarding. march 2019. URL https://hal.inria.fr/hal-02061838. 60, 127

[116] Shahid Raza, Tony Chung, Simon Duquennoy, Thiemo Voigt, Utz Roedig, et al. Securing Internet of Things with Lightweight IPSec. 2010. 60

[117] Kris Pister and Lance Doherty. TSMP: Time Synchronized Mesh Protocol. *IASTED Distributed Sensor Networks*, 391:398, 2008. 61

[118] K. Pister M. Vucinic, J. Simon and M. Richardson. Minimal Security Framework for 6TiSCH. Draft, IETF, November 2018. URL https://tools.ietf.org/pdf/draft-ietf-6tisch-minimal-security-09.pdf. 63

[119] Gabriel Montenegro, Christian Schumacher, and Nandakishore Kushalnagar. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. RFC 4919, August 2007. URL https://rfc-editor.org/rfc/rfc4919.txt. 64

[120] Andrew Martin. The ten-page introduction to Trusted Computing. November 2008. URL https://www.cs.ox.ac.uk/files/1873/RR-08-11.PDF. 66

[121] Pieter Maene, Johannes Götzfried, Ruan De Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2017. 66

[122] arm. TrustZone for Cortex-M, 2019. URL https://www.arm.com/why-arm/technologies/trustzone-for-cortex-m. 66

[123] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):7, 2017. 67, 74

[124] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SWATT: Software-Based Attestation for Embedded Devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 272–282. IEEE, 2004. 68

[125] Taejoon Park and Kang G Shin. Soft Tamper-Proofing via Program Integrity Verification in Wireless Sensor Networks. *IEEE Transactions on mobile computing*, 4(3):297–309, 2005.

[126] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Scuba: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94. ACM, 2006. 69, 70

[127] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 1–16. ACM, 2005.

[128] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote Software-Based Attestation for Wireless Sensors. In *European Workshop on Security in Ad-hoc and Sensor Networks*, pages 27–41. Springer, 2005.

[129] Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. Distributed Software-Based Attestation for Node Compromise Detection in Sensor Networks. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 219–230. IEEE, 2007. 68

[130] Arvind Seshadri, Mark Luk, and Adrian Perrig. SAKE: Software Attestation for Key Establishment in Sensor Networks. In *International Conference on Distributed Computing in Sensor Systems*, pages 372–385. Springer, 2008. 68, 69, 70

[131] Daniele Perito and Gene Tsudik. Secure Code Update for Embedded Devices via Proofs of Secure Erasure. In *European Symposium on Research in Computer Security*, pages 643–662. Springer, 2010. 69

[132] Steffen Schulz, André Schaller, Florian Kohnhäuser, and Stefan Katzenbeisser. Boot attestation: Secure Remote Reporting with Off-The-Shelf IoT Sensors. In *European Symposium on Research in Computer Security*, pages 437–455. Springer, 2017. 70

[133] *LPC1224/25/26/27 User manual.* NXP, 05 2017. 71

[134] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015. 72

[135] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *NDSS*, volume 12, pages 1–15, 2012. 74

[136] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 400–409. ACM, 2009. 78

[137] Yanlin Li, Yueqiang Cheng, Virgil Gligor, and Adrian Perrig. Establishing Software-Only Root of Trust on Embedded Systems: Facts and Fiction. In *Cambridge International Workshop on Security Protocols*, pages 50–68. Springer, 2015. 78

[138] Karim Eldefrawy and Gene Tsudik. Opinion: Advancing Remote Attestation via Computer-aided Formal Verification of Designs and Synthesis of Executables. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 45–48. ACM, 2019. 78, 147

[139] James Newsome, Elaine Shi, Dawn Song, and Adrian Perrig. The Sybil Attack in Sensor Networks: Analysis & Defenses. In *Third international symposium on information processing in sensor networks, 2004. IPSN 2004*, pages 259–268. IEEE, 2004. 82

[140] Boris Danev and Srdjan Capkun. Transient-Based Identification of Wireless Sensor Nodes. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 25–36. IEEE Computer Society, 2009. 83

[141] Yuexiu Xing, Aiqun Hu, Junqing Zhang, Linning Peng, and Guyue Li. On Radio Frequency Fingerprint Identification for DSSS Dystems in Low SNR Scenarios. *IEEE Communications Letters*, 22(11):2326–2329, 2018. 83

[142] David A Knox and Thomas Kunz. AGC-based RF Fingerprints in Wireless Sensor Networks for Authentication. In *2010 IEEE International Symposium ön A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 1–6. IEEE, 2010. 83

[143] David Stanislowski, Xavier Vilajosana, Qin Wang, Thomas Watteyne, and Kristofer SJ Pister. Adaptive Synchronization in IEEE 802.15.4e Networks. *IEEE Transactions on Industrial Informatics*, 10(1):795–802, 2013. 84

[144] Fabian Lanze, Andriy Panchenko, Benjamin Braatz, and Andreas Zinnen. Clock Skew Based Remote Device Fingerprinting Demystified. In *2012 IEEE Global Communications Conference (GLOBECOM)*, pages 813–819. IEEE, 2012. 85

[145] Nouha Baccour, Anis Koubâa, Luca Mottola, Marco Antonio Zúñiga, Habib Youssef, Carlo Alberto Boano, and Mário Alves. Radio Link Quality Estimation in Wireless Sensor Networks: A Survey. *ACM Transactions on Sensor Networks (TOSN)*, 8(4):34, 2012. 85

[146] Sheng Zhong, Li Li, Yanbin Grace Liu, and Yang Richard Yang. Privacy-Preserving Location-Based Services for Mobile Users in Wireless Networks. *Department of Computer Science, Yale University, Technical Report ALEU/DCS/TR-1297*, 2004. 85, 87

[147] Cesare Alippi and Giovanni Vanini. A RSSI-Based and Calibrated Centralized Localization Technique for Wireless Sensor Networks. In *Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06)*, pages 5–pp. IEEE, 2006. 85

[148] Murat Demirbas and Youngwhan Song. An RSSI-Based Scheme for Sybil Attack Detection in Wireless Sensor Networks. In *2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'06)*, pages 5–pp. IEEE, 2006. 85, 87, 88

[149] Jiangtao Wang, Geng Yang, Yuan Sun, and Shengshou Chen. Sybil Attack Detection Based on RSSI for Wireless Sensor Network. In *2007 International Conference on Wireless Communications, Networking and Mobile Computing*, pages 2684–2687. IEEE, 2007. 85, 87, 88

[150] Thomas Watteyne, Steven Lanzisera, Ankur Mehta, and Kristofer SJ Pister. Mitigating Multipath Fading through Channel Hopping in Wireless Sensor Networks. In *2010 IEEE International Conference on Communications*, pages 1–5. IEEE, 2010. 86

[151] G Edward Suh and Srinivas Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *2007 44th ACM/IEEE Design Automation Conference*, pages 9–14. IEEE, 2007. 89

[152] Daniel E Holcomb, Wayne P Burleson, and Kevin Fu. Power-up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. *IEEE Transactions on Computers*, 58(9):1198–1210, 2008. 90, 92

[153] Steven J Murdoch. Hot or Not: Revealing Hidden Services by their Clock Skew. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 27–36. ACM, 2006. 96

[154] Great Scott Gadgets. Project Ubertooth, 2018. URL `https://github.com/greatscottgadgets/ubertooth/`. 96, 108

[155] Xavier Vilajosana, Qin Wang, Fabien Chraim, Thomas Watteyne, Tengfei Chang, and Kristofer SJ Pister. A Realistic Energy Consumption Model for TSCH networks. *IEEE Sensors Journal*, 14(2):482–489, 2014. 97, 104

[156] *IEEE 802.15.4e Low-Rate Wireless Personal Area Networks (Amendment to IEEE Std 802.15.4-2011)*. IEEE Standards Office, New York, NY, USA, 2012. 98

[157] Elvis Vogli, Giuseppe Ribezzo, Luigi Alfredo Grieco, and Gennaro Boggia. Fast Join and Synchronization Schema in the IEEE 802.15. 4e MAC. In *2015 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 85–90. IEEE, 2015. 98, 101, 102, 104

[158] Thang Phan Duy, Thanh Dinh, and Younghan Kim. A Rapid Joining Scheme Based on Fuzzy Logic for Highly Dynamic IEEE 802.15. 4e Time-Slotted Channel Hopping Networks. *International Journal of Distributed Sensor Networks*, 12(8):1550147716659424, 2016. 98, 101, 102, 104

[159] Telosb. URL `https://openwsn.atlassian.net/wiki/display/OW/TelosB`. 101

[160] Liviu-Octavian Varga, Gabriele Romaniello, Mališa Vučinić, M. Favre, A. Banciu, R. Guizzetti, C. Planat, Pascal Urard, Martin Heusse, Franck Rousseau, Olivier Alphand, Étienne Dublé, and Andrzej Duda. GreenNet: an Energy Harvesting IP-enabled Wireless Sensor Network. *IEEE Internet of Things Journal*, 2, 2015. 101

[161] Xavier Vilajosana, Kris Pister, and Thomas Watteyne. Minimal IPv6 over the TSCH Mode of IEEE 802.15.4e (6TiSCH) Configuration. RFC 8180, May 2017. URL `https://rfc-editor.org/rfc/rfc8180.txt`. 101

[162] *MSP430 32-kHz Crystal Oscillators*. Texas Instruments, 8 2006. Rev. 2. 103

[163] *Bluetooth Core specification 4.0*. Bluetooth SIG, 6 2010. 105

[164] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology. *Sensors*, 12(9):11734–11753, 2012. 105

[165] *Bluetooth Core specification 5.0*. Bluetooth SIG, 12 2016. 105, 106, 108

[166] Lily Hay Newman. Inside the Unnerving Supply Chain Attack that Corrupted CCleaner, 2018. URL `https://www.wired.com/story/inside-the-unnerving-supply-chain-attack-that-corrupted-ccleaner/`. 106

[167] Raj Chandra Bose and Dwijendra K Ray-Chaudhuri. On a Class of Error Correcting Binary Group Codes. *Information and control*, 3(1):68–79, 1960. 108

[168] Magic Blue UU Bluetooth Bulb. URL `https://www.gearbest.com/smart-light-bulb/pp_230349.html`. 108

[169] *LPC1759/58/56/54/52/51: 32-bit ARM Cortex-M3 MCU; up to 512 kB flash and 64 kB SRAM with Ethernet, USB 2.0 Host/Device/OTG, CAN*. NXP Semiconductors, 08 2015. URL `https://www.nxp.com/docs/en/data-sheet/LPC1759_58_56_54_52_51.pdf`. 108

[170] Raspberry Pi Foundation. Raspberry Pi 3B, 2016. URL `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`. 109

[171] *CYW43438: Single-Chip IEEE 802.11 b/g/n MAC/Baseband/Radio with Integrated Bluetooth 4.2*. Cypress, 07 2018. 109

[172] Gareth Halfacree. Benchmarking the Raspberry Pi 3 B+. URL https://medium.com/@ghalfacree/benchmarking-the-raspberry-pi-3-b-plus-44122cf3d806. 109, 110

[173] *175, 177, 179 True-rms Multimeters: User manual.* Fluke, 05 2003. URL https://dam-assets.fluke.com/s3fs-public/175_____umeng0200.pdf. 109, 110

[174] *Qualcomm Snapdragon 801 Processor.* Qualcomm Technologies, Inc, 2014. URL https://www.qualcomm.com/media/documents/files/snapdragon-801-processor-product-brief.pdf. 110, 111

[175] *WCN3680B/WCN3660B: device specification.* Qualcomm Technologies, Inc, 07 2017. URL https://developer.qualcomm.com/download/sd410/wcn3680b-wcn3660b-device-spec.pdf. 110, 111

[176] TCXO, Temperature Compensated Crystal Oscillator. URL https://www.electronics-notes.com/articles/electronic_components/quartz-crystal-xtal/tcxo-temperature-compensated-crystal-xtal-oscillator.php. 110

[177] IFIXIT. Motorola Moto X Teardown. URL https://nl.ifixit.com/Teardown/Motorola+Moto+X+Teardown/16867. 111

[178] APPLE IFIXIT. The Teardown: Apple iPhone 5s. 8, November 2013. 111, 112

[179] *BCM4334: Single Chip IEEE 802.11 a/b/g/n MAC/Baseband/Radio with Integrated Bluetooth 4.0 + HS and FM Receiver.* Cypress Semiconductor Corporation, 07 2016. URL http://www.rumjd.com/Attachments/20160820160827_152216_171.pdf. 111

[180] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal Covert Channels on Multi-Core Platforms. In *USENIX Security Symposium*, pages 865–880, 2015. 113

[181] Mordechai Guri, Matan Monitz, Yisroel Mirski, and Yuval Elovici. Bitwhisper: Covert Signaling Channel Between Air-Gapped Computers Using Thermal Manipulations. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pages 276–289. IEEE, 2015. 113

[182] wolfSSL. wolfSSL, 2019. URL https://www.wolfssl.com. 116

[183] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008. URL https://rfc-editor.org/rfc/rfc5246.txt. 116

[184] Paul Wouters, Hannes Tschofenig, John Gilmore, Samuel Weiler, and Tero Kivinen. Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7250, June 2014. URL https://rfc-editor.org/rfc/rfc7250.txt. 117

[185] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, RFC Editor, November 1998. 118

[186] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, RFC Editor, October 1996. 120

[187] Jim Griner, John Border, Markku Kojo, Zach D. Shelby, and Gabriel Montenegro. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001. URL https://rfc-editor.org/rfc/rfc3135.txt. 120

[188] Joseph Ishac and Mark Allman. *On the Performance of TCP Spoofing in Satellite Networks*, volume 1. IEEE, 2001. 120

[189] Martin Thomson. Record Size Limit Extension for TLS. RFC 8449, August 2018. URL https://rfc-editor.org/rfc/rfc8449.txt. 121

[190] Tengfei Chang, Mališa Vučinić, Xavier Vilajosana, Simon Duquennoy, and Diego Dujovne. 6TiSCH Minimal Scheduling Function (MSF). Internet-Draft draft-ietf-6tisch-msf-06, Internet Engineering Task Force, August 2019. URL https://datatracker.ietf.org/doc/html/draft-ietf-6tisch-msf-06. Work in Progress. 122

[191] Sally Floyd, Mark J. Handley, and Eddie Kohler. Datagram Congestion Control Protocol (DCCP). RFC 4340, March 2006. URL `https://rfc-editor.org/rfc/rfc4340.txt`. 123

[192] Dr. Craig Partridge, Mark Allman, and Sally Floyd. Increasing TCP's Initial Window. RFC 3390, November 2002. URL `https://rfc-editor.org/rfc/rfc3390.txt`. 123

[193] Jerry Chu, Nandita Dukkipati, Yuchung Cheng, and Matt Mathis. Increasing TCP's Initial Window. RFC 6928, April 2013. URL `https://rfc-editor.org/rfc/rfc6928.txt`. 123

[194] Thomas Watteyne, Xavier Vilajosana, Branko Kerkez, Fabien Chraim, Kevin Weekly, Qin Wang, Steven Glaser, and Kris Pister. OpenWSN: Open-Source Implementations of Protocol Stacks Based on IoT Standards, 2013. URL `http://www.openwsn.org`. 123

[195] Openvisualizer. URL `https://github.com/openwsn-berkeley/openvisualizer`. 124

[196] Dr. Steve E. Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, July 2017. URL `https://rfc-editor.org/rfc/rfc8200.txt`. 126

[197] Dr. Vern Paxson and Mark Allman. Computing TCP's Retransmission Timer. RFC 2988, November 2000. URL `https://rfc-editor.org/rfc/rfc2988.txt`. 127

[198] Ahmed Ayadi, Patrick Maillé, David Ros, Laurent Toutain, and Tiancong Zheng. Implementation and Evaluation of a TCP header Compression for 6LoWPAN. In *2011 7th International Wireless Communications and Mobile Computing Conference*, pages 1359–1364. IEEE, 2011. 130

[199] Michael Jones, Erik Wahlstroem, Samuel Erdtman, and Hannes Tschofenig. CBOR Web Token (CWT). RFC 8392, May 2018. URL `https://rfc-editor.org/rfc/rfc8392.txt`. 133, 134

[200] Eran Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849, April 2010. URL `https://rfc-editor.org/rfc/rfc5849.txt`. 133

[201] Aimaschana Niruntasukrat, Chavee Issariyapat, Panita Pongpaibool, Koonlachat Meesublak, Pramrudee Aiumsupucgul, and Anun Panya. Authorization Mechanism for MQTT-Based Internet of Things. In *2016 IEEE International Conference on Communications Workshops (ICC)*, pages 290–295. IEEE, 2016. 133

[202] Jim Schaad. CBOR Object Signing and Encryption (COSE): Headers for carrying and referencing X.509 certificates. Internet-Draft draft-ietf-cose-x509-04, Internet Engineering Task Force, September 2019. URL `https://datatracker.ietf.org/doc/html/draft-ietf-cose-x509-04`. Work in Progress. 134, 135, 137

[203] Robert Moskowitz, Tobias Heer, Petri Jokela, and Thomas R. Henderson. Host Identity Protocol Version 2 (HIPv2). RFC 7401, 2015. URL `https://rfc-editor.org/rfc/rfc7401.txt`. 141

[204] Ayesha Khalid, James Howe, Ciara Rafferty, and Máire O'Neill. Time-Independent Discrete Gaussian Sampling for Post-Quantum Cryptography. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 241–244. IEEE, 2016. 147

[205] Eric Rescorla and Richard Barnes. Compact TLS 1.3. Internet-Draft draft-rescorla-tls-ctls-02, Internet Engineering Task Force, July 2019. URL `https://datatracker.ietf.org/doc/html/draft-rescorla-tls-ctls-02`. Work in Progress. 147