Designing a specific Low Power architecture for blockchain and Smart Contracts operations in IoT platform

Roland Kromes

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Conception d'une architecture spécifique Low Power pour les accès Blockchain et Smart Contracts des plateformes IoT

Roland Kromes

Laboratoire d'Electronique, Antennes et Télécommunications (LEAT)

Présentée en vue de l'obtention du grade de Docteur en Électronique **d'**Université Côte d'Azur

Dirigée par :
François Verdier, Dr. Prof.

Devant le jury, composé de :
Jean-Christophe Prévotet, Dr. Prof., INSA de Rennes
Pascal Lafourcade, Dr. MCF, Université Clermont Auvergne
Abdoulaye Gamatié, Dr. CNRS Research Director, LIRMM, Univ. Montpellier,
Christine Hennebert, Dr. Chercheuse, CEA LETI
Parisa Ghodous, Dr. Prof., Université Claude Bernard Lyon 1
Xing Liu, Dr., Kwantlen Polytechnic University
Patricia Guitton-Ouhamou, Dr., Renault
François Verdier, Dr. Prof., Université Côte d'Azur

Soutenue le : 08-12-2021

Conception d'une architecture spécifique Low Power pour les accès Blockchain et Smart Contracts des plate-formes IoT

Designing a Specific Low Power Architecture for Blockchain and Smart Contracts operations in IoT platform

Président du jury

- Abdoulaye Gamatié, Dr. CNRS Research Director, LIRMM, Univ. Montpellier

Rapporteurs

- Jean-Christophe Prévotet, Dr. Prof., INSA de Rennes
- Pascal Lafourcade, Dr. MCF, Université Clermont Auvergne

Examinateurs

- Parisa Ghodous, Dr. Prof, Université Claude Bernard Lyon 1
- Christine Hennebert, Dr. Chercheuse, CEA LETI
- François Verdier, Dr. Prof., Université Côte d'Azur

Invités

- Xing Liu, Dr. Senior Faculty Member, Kwantlen Polytechnic University
- Patricia Guitton-Ouhamou, Dr., Renault

Résumé

De nos jours, de nombreuses applications IoT sont devenues une partie essentielle de la vie des gens, des industries et des écosystèmes modernes. La plupart des applications IoT sont basées sur un système centralisé dans lequel tous les participants au système doivent s'en remettre à une entité centrale. Dans un tel système, l'immuabilité, la traçabilité et la transparence des données ne peuvent être assurées.

La technologie Blockchain est un système entièrement décentralisé dans lequel le tiers de confiance (entité centrale) est supprimé. La particularité de cette technologie est qu'elle prévoit qu'une fois que les données y sont déployées, elles ne peuvent pas être modifiées ou retirées du système. Contrairement aux systèmes centralisés, la blockchain assure la traçabilité et la transparence des données. La plupart des blockchains modernes permettent également le déploiement de Smart Contracts, qui sont des programmes numériques pouvant être lus par tous les participants et exécutés automatiquement en fonction d'un événement sur la blockchain.

Les caractéristiques avantageuses de la technologie blockchain montrent un intérêt évident pour l'intégration des IoT avec la technologie blockchain.

Cette contribution de thèse étudie les possibilités d'intégration des IoT avec la technologie blockchain. L'une des principales parties de la contribution est le développement d'un modèle d'architecture matérielle IoT dédiée à faible consommation d'énergie qui permet la communication avec plusieurs types de blockchains. Le modèle d'architecture est composé d'un CPU basé sur ARM émulé sur QEMU et d'accélérateurs matériels cryptographiques modélisés dans le langage de description matérielle de haut niveau SystemC-TLM. Un système d'exploitation (OS) Linux est exécuté au sommet de l'architecture. Le développement de pilotes de périphériques dédiés au noyau Linux a été nécessaire car les API exécutés sur Linux ne peuvent pas accéder directement à des IP matérielles (propriétés intellectuelles) données.

Les pilotes de périphériques dédiés et la bibliothèque SystemC TLM PwClkARCH ont été utilisés pour mettre en œuvre la gestion de l'énergie de l'architecture afin d'optimiser la consommation énergétique globale de l'architecture lorsqu'une API blockchain donnée est exécutée. Ce travail propose également différentes API de blockchain (Ethereum, Hyperledger Sawtooth) écrites en C++, incluant toutes les exigences de la blockchain donnée, par exemple, l'encodage ABI, la structure de transaction et les primitives cryptographiques.

Les résultats de la contribution montrent qu'une réduction significative de la consommation énergétique globale peut être obtenue lorsque l'opération de multiplication des points de la courbe elliptique est accélérée par le matériel. Les résultats montrent également que lorsque la taille de la charge utile de la transaction augmente, il est intéressant d'utiliser des accélérateurs matériels de hachage pour réduire la consommation d'énergie globale et accélérer l'exécution de l'API donnée.

Mots clés

Blockchain; IoT; Low-Power; Modélisation de haut niveau des SoCs

Abstract

Nowadays, numerous IoT applications have become an essential part of people's lives, industries, and modern ecosystems. Most IoT applications are based on a centralized system in which all of the system participants have to rely on a central entity. In such a system, data immutability, data traceability, and transparency cannot be provided.

Blockchain technology is an entirely decentralized system in which the third trusted party (central entity) is removed. The particularity of this technology is that it provides that once data is deployed on it, it cannot be modified or removed from the system. Contrarily to centralized systems, blockchain provides data traceability and transparency. Most modern blockchains also allow the deployment of smart contracts, which are digital programs that can be read by all participants and executed automatically according to an event on the blockchain.

The advantageous features of blockchain technology show a clear interest in the integration of IoT with blockchain technology.

This thesis contribution studies the integration possibilities of IoT with blockchain technology. One of the main parts of the contributions is developing a model of dedicated low-power-consumption IoT hardware architecture that enables communication with multiple types of blockchains. The architecture model is composed of an ARM-based CPU emulated on QEMU and cryptographic hardware accelerator designs modeled in SystemC TLM high-level hardware description language. A Linux Operating System (OS) is executed on top of the architecture. The development of dedicated Linux Kernel device drivers was required because the AIPs executed on Linux can not directly access given hardware IPs (Intellectual Properties).

Dedicated device drivers and PwClkARCH SystemC TLM library were used to implement the architecture's power management to optimize the architecture's overall energy consumption when a given blockchain API is executed.

This work also proposes different blockchain APIs (Ethereum, Hyperledger Sawtooth) written in C++, including all the requirements of the given blockchain, e.g., ABI encoding, transaction structure, and cryptographic primitives.

The contribution results represent that a significant reduction of the overall energy consumption can be achieved when the elliptic curve point multiplication operation is hardware accelerated. The results also show that when the payload size of the transaction increases, it is worth using hash hardware accelerators to decrease the overall energy consumption and accelerate the given API's execution.

Keywords

Blockchain; IoT; Low-Power; High Level Modelling of SoCs

“ You see, one thing is, I can live with doubt and uncertainty and not knowing. I think it’s much more interesting to live not knowing than to have answers which might be wrong.”

Richard P. Feynman.

Acknowledgements

First of all, I would like to thank my thesis supervisor, Professor François Verdier, who always gave me suggestions, motivation and help to progress. Since my first year at the university, Professor Verdier has taught me, and he has also shown me the beauty of science. After two internships and three years of doctoral studies with him, I can proudly announce that I have found in his person a researcher, a teacher, a good friend and an honorable ”spiritual father”. I sincerely hope that our research collaboration and friendship will not end after I leave the laboratory.

Secondly, I would like to thank my family, especially my mother, who has always been by my side in good and bad times. Thank you, Mom and Charly, for believing in me and helping me get to this point. Without your care and love, it would have been impossible for me to complete my university education and have the opportunity to defend my doctorate. I would like to thank the Kromes family, on whom I can always count, thanks to uncle Gabor and his wife Ildi, and my cousins Klaudia and Gabor. And finally, I would like to thank my grandparents Mariann and Richard for their infinite love and trust. I will never forget you!

I would like to thank my thesis defense jury: Pascal Lafourcade, Jean-Christophe Prévotet, Abdoulaye Gamatié, Parisa Ghodous, Christine Hennebert, Xing Liu and Patricia Guitton-Ouhamou.

I would like to thank the EDGE team, and in particular the blockchain team Luc Gerrits, Cyril Naves, Edouard Kilimou, and François Verdier.

During my years of research in the LEAT lab, I found not only answers to scientific questions, but also good friends. Thanks guys for supporting me and sharing good moments and fun with each other. Lyes Khacef and his sincere friendship, I couldn’t be more grateful. Thank you, Luc Gerrits, program addict and great friend with whom I was able to publish several articles. Thanks to Marta Ballatore, who helped me understand the management and acceptability aspects of blockchain technology and who has become a good friend of mine. Thank you Marino Rasamuel, you are the worst and best colleague in the office. I had too much fun in our office during these years. I want to thank my friends who always made me happy in bad times, Attila Nagy, Mark Benjamin Thurzo, Szilvia Somogyi, Mircea Moscu, Diana Resmerita, Edgar Lemaire, Flora Zidane, Luca Santamaria, Ahmed Oualha, Yacine Khacef, Katarzyna Tomasiak, Rémi Garcia, Cyril Naves and Yassine Chouchane. I hope I haven’t forgotten anyone.

I would also like to thank all the teachers at the university, high school, and school for giving me a solid foundation of scientific knowledge. A special thanks to DS4H (Digital System For Humans) Graduate School and Research for funding my thesis.

Finally, I would like to thank the motivation provided by my favorite bands, Pink Floyd, the Rolling Stones, the Nigun Quartet, Hugh Laurie and Marcus Miller.

Author's Publication List

Related International conference papers

- **R. Kromes**, L. Gerrits and F. Verdier, *Adaptation of an embedded architecture to run Hyperledger Sawtooth Application*, 2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), Vancouver, BC, Canada, 2019, pp. 0409-0415.
- **R. Kromes**, F. Verdier, *IoT devices hardware modeling for executing Blockchain and Smart Contracts applications* 16th ACS/IEEE International Conference on Computer Systems and Applications AICCSA 2019, ACS/IEEE, Nov 2019, Abu Dhabi, United Arab Emirates.
- L. Gerrits, **R. Kromes**, F. Verdier, *A True Decentralized Implementation Based on IoT and Blockchain: a Vehicle Accident Use Case*, COINS 2020 - IEEE International Conference on Omni-layer Intelligent Systems, Sep 2020, Madrid, Spain.
- L. Gerrits, E. Kilimou **R. Kromes**, L. Faure, F. Verdier, *A Blockchain cloud architecture deployment for an industrial IoT use case*, COINS 2021 - IEEE International Conference on Omni-layer Intelligent Systems, Sep 2021, Madrid, Spain.

Related National conference papers

- **R. Kromes**, F. Verdier, *An IoT hardware modeling for using blockchain with Smart Contracts applications*, 13ème Colloque National du GDR SOC2 Montpellier, Jun 2019, Montpellier, France.
- L. Gerrits, **R. Kromes**, T. Kilimou, F. Verdier, *Hyperledger Sawtooth Blockchain for IoT-Blockchain Based Ecosystem*, 15ème Colloque National du GDR SOC2 Rennes, Jun 2021, Rennes, France.

Unrelated International conference papers

- **R. Kromes**, A. Russo, B. Miramond, F. Verdier, *Energy consumption minimization on LoRaWAN sensor network by using an Artificial Neural Network based application*, Sensors Applications Symposium 2019, Mar 2019, Sophia-Antipolis, France. pp.1-6

Contents

Bibliography **175**

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The Internet of Things (IoT) is becoming a key technology in people's daily lives. The best-known devices of IoT technology are smartphones, smartwatches, intelligent sensors, health-monitoring devices, and many others. Some of these devices are only gadgets, and it is possible to live without them, but many of them play an essential part in people's lives. This type of technology also has significant importance in the industrial domain. IoT devices started to be embedded in robots, in machines, in vehicles and in airplanes, for example.

Most of the IoT are small and constrained devices that can also be considered as embedded devices realized as Systems on Chip (SoC). IoT devices are used widely, and the demand for these devices is high but the time to market is short. Numerous new applications require faster execution, lower power consumption, and other specific needs. In order to meet all of the application-specific requirements in a relatively short time, the development phase of the new SoC devices starts to be divided into high- and low-level modeling phases. Dividing the modeling into two phases can accelerate the development and the architecture design verification and validation process. High-level modeling allows the parallel development of a basic functional or power-controlled architecture and the application-specific software that would run on the architecture.

High-level modeling can provide early results of the functional or power-controlled model of the given architecture. These results can also demonstrate that the low-level modeled designs need to be modified to meet the specific requirements. The most significant part of the contributions of this thesis work is based on the high-level modeling, which is realized by SystemC-TLM hardware description language.

Blockchain is a distributed ledger technology which also means that the data contained by such a ledger is decentralized. The decentralization also means that there is no need for a trusted third party that holds the overall system's data, it is distributed among the system's members (the copy of the blockchain is contained by each of the participants). It must be noted that this technology is not only used for cryptocurrency exchange but also for providing data traceability, integrity, and data immutability. Thanks to the features of blockchain technology, the combination of blockchain with IoT systems can solve many problems that are present in today's IoT-based systems. The most significant problems with IoT structures are device authentication, security of data sending and sharing, and trust in unfalsifiability of collected data.

It was mentioned above that, in general, new applications can require new specific hard-

ware designs, mainly when the application needs to be applied on IoT architectures. Blockchain and IoT technologies have different hardware and software requirements, and it can also be noted that these two technologies are entirely different. The integration of IoT with the blockchain domain is a challenging research and development work. Today it can be assumed that it does not make sense that an IoT device contains a copy of a blockchain. However, an IoT may make all of the essential operations enabling communication with a given blockchain.

In order to perform all of the essential operations more efficiently in terms of energy consumption and execution speed, a specific IoT hardware architecture design is needed. The first step in realizing such a design may be high-level modeling, that is exactly one of the aims of this thesis.

1.2 IoT an Blockchain technologies

This section provides a brief introduction to the definition of the IoT and blockchain technologies that form the fundamental part of this thesis. The section also demonstrates the challenges in blockchain-IoT based applications.

1.2.1 Definition of IoT

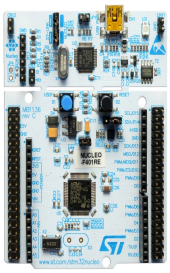
The acronym IoT means Internet of Things, which contains an enormous number of devices connected directly or via mediators (gateways) to the Internet. By the statistic estimations [20] [21] until 2025, the number of connected devices to the Internet would achieve 75 billion.

Today the IoT architecture contains several types of devices that can be more or less constrained in terms of computational power, memory place, and battery lifetime.

The figure 1.1a represents NUCLEO-F446RE development card which is based on an M4 STM32F446RE micro-controller which can be considered as a constrained IoT device. The ARM Cortex-M4F being the core of this micro-controller operates on 180MHz, which is far less important than the frequency used in modern PCs. The low frequency can be bi-rationally associated with less computational power. This architecture is equipped with 128 kB of SRAM, which also demonstrates that the data storage is limited, and in some use cases, external storage is required (e.g., SD cards). This device can be powered thanks to an external battery. However, the lifetime of the battery is not limitless.

A less constrained IoT device is presented in figure 1.1b, namely a Raspberry Pi 3B+, which is based on a BCM2837 64 bit architecture using 1.2 GHz. This architecture is equipped with 1GB RAM, which is far more significant than the memory of the mentioned Nucleo board but still less significant than the memory size of a modern PC. It is possible to connect a Raspberry to a battery, however, its battery lifetime is limited to a few minutes comparing to the Nucleo board that can be hours.

Today the IoT architecture is not only limited to connected devices to the Internet, but also it can contain Wireless Sensor Networks (WSN) using different types of communication protocols. It can also contain firmware, middle-ware, and also service levels. Whether it be a wireless or wired connection to the Internet, most IoT devices also contain radio modules for communication, which also increase the device's power consumption. In optimal



(a) NUCLEO-F446RE development card. Image is retrieved from [22]

(b) Raspberry Pi 3B+. Image is retrieved from [23]

Figure 1.1: Example of constrained IoT devices

cases, the communication and connectivity of the IoT devices should be limited in order to consume less energy and increase the battery lifetime. All of the limitations mentioned previously highlight that the choice of the IoT hardware architecture is application and use-case-dependent.

According to [24], the main objective of modern IoT architectures is to guarantee the operation of a given system based on events without human intervention. Generally, one of the main goal of the IoT architectures is the data collecting from the connected objects to a central entity (e.g., cloud application). The data flow from the connected devices to the central entity can be transmitted directly or via gateways or edge devices. Using a central entity for data storing in IoT networks can include several points of failure in terms of security, privacy, and data transparency. Replace the central unit with a distributed and highly secured system such as blockchain technology can solve the bottleneck mentioned earlier. However, integrating IoT with blockchain technology can be a challenge which is described later in this thesis work.

1.2.2 Definition of Blockchain

The history of blockchain technology has started with Nakamoto's Bitcoin implementation in 2008 [25]. This blockchain is better known because of its cryptocurrency and not because of its decentralized ledger nature. The basic idea of Bitcoin was to enable cryptocurrency transactions without the need of a trusted third party such as a bank. Avoiding the system's third party means that the members no longer need to rely on and trust a central entity.

The decentralization of the system can be achieved through the fact that the blockchain is a peer-to-peer network in which each member is connected to all members of the network, and the same data (database) is stored in every peer.

Blockchain technology provides several advantages, such as once data are added to the blockchain, they become immutable (it cannot be modified or removed anymore). In addition to data immutability, the technology also provides data traceability which means that every actions on the system is recorded forever. All of the recorded data and actions can be seen by every network participant; hence, data transparency is also provided.

Obviously, the distributed database and the knowledge of every network participant do not provide a completely secure environment. For this reason, blockchain technology applies essential cryptographic primitives such as hash functions which provide immutability of the

data stored in the blockchain. The hash value of a message is a unique representation (so called fingerprint) of the message.

The data is stored in form of transactions ordered in a block which contains the hash of the previously added block. The previous block's hash value contained by the newly added block can be considered as the link between the two blocks. The hash values of the blocks is link between them, hence the name blockchain (see figure 1.2).

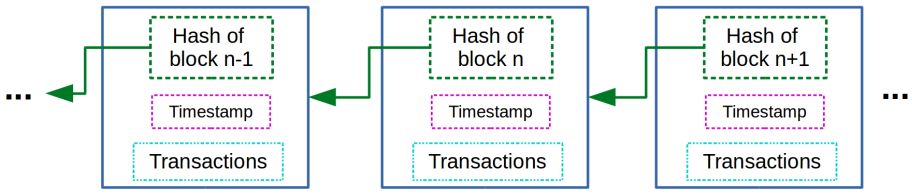


Figure 1.2: Examples of a blockchain

Data can be set to the blockchain in the form of transactions that generally contain a payload field (containing the raw data). The transactions are validated according to a common agreement of the network that is also known as the consensus rule (introduced below), which plays a significant role in every blockchain system. The technology also applies digital signature algorithms which provide data traceability (which participants sent which transactions) [14]. Every transaction sent to the blockchain is signed with the private key of the sender. Thanks to digital signature the data tracability is provided, every transaction sent to the blockchain can be associated with a participant of the blockchain network.

Avoiding a third party of a given system can be useful not only in cryptocurrency transactions but also in new types of eco-systems in which the participants do not want to rely on trusted third parties anymore. In centralized systems related or not to financial transactions, a central unit contains a certain business logic (e.g., a cloud collects IoT devices data analyzed afterward). The participants of the system have to trust this central unit. Blockchain technology can also contain such a business logic thanks to smart contracts. The new version of blockchains, also called blockchain 2.0, started with Ethereum blockchain's implementation [26]. These blockchains can allow smart contracts deployment. The blockchain's nature provides a secure environment for the execution of the smart contract (introduced below), because once the smart contract is deployed it becomes unchangeable, no manipulation of the business logic is possible.

Thanks to all of the advantages mentioned above that this technology provides, it can be concluded that blockchain technology can resolve several issues of centralized systems and also can provide a more secure system.

1.2.3 Definition of consensus rules

Consensus rules are one of the key-points of a blockchain [27] because these algorithms serve to achieve a common agreement on which blocks to add to the blockchain. The consensus rule also aims to achieve the common agreement in a secure way, in which the malicious participants can be filtered. It is also possible that the system contains faulty participants; however, the overall system can remain a trusted environment until (the system continues working as it was desired initially) the majority of the participants are honest.

Today, there exist many consensus rules (about 30 [28]). One of the most known is the Proof-of-Work [29] (PoW) consensus which is applied in Bitcoin [25] and Ethereum (main network) [26] blockchains. This consensus is based on a cryptographic challenge that has to be resolved by the members taking part in the consensus. This consensus rule provides a secure environment until 51% of the participants are honest. It can be noted that the PoW consensus is a hardware resource-intensive task that is computed among the participants of the consensus rule. The member who resolves a given PoW faster than the other participants gains an amount of cryptocurrency (e.g., Ether in Ethereum blockchain).

The Practical Byzantine Fault Tolerance [30] (PBFT) is one of the most used consensus rules among the private and consortium blockchains (e.g., Hyperledger Fabric [31]). This consensus is based on successive votes, which can achieve a secure environment until 2/3 of the consensus participants are honest. Contrarily to PoW, in PBFT the goal is not to make a contest between the participants for gaining cryptocurrency but to achieve an agreement more securely. Because there are no cryptographical challenges in this consensus, this consensus rule is less hardware resource intensive.

1.2.4 Definition of a Smart Contracts

The main idea of smart contracts was invented by Nick Szabo [32] almost eleven years before the first blockchain was implemented. A smart contract can be defined as a digital program written in a programming language that can be executed automatically according to a specific event. In general, a smart contract is used to automatize a process or realize a given business logic.

The application of smart contracts to blockchain technology allows blockchain members to read all smart contracts and call them if they wish. Using smart contracts can also replace the procedure that a third-party entity does in a centralized system. Thanks to the nature of blockchain technology, smart contracts can be executed in a cryptographically secured environment that also provides that the results performed by smart contracts cannot be falsified. The events that can launch the execution of a smart contract are the transactions, and the deployment of smart contract generally can be done by sending it via a transaction.

1.2.5 Challenges in Blockchain-IoT applications

It can be noted that in modern IoT architectures, the goal is the automation of the architecture to avoid human intervention. Most of today's architecture implement this automation by using a centralized entity (application or service level). This approach requires that the members of the systems trust this central unit. In order to remove this third party and ensure trust by the nature of the system, the mentioned central entity could be replaced by a blockchain structure. Since blockchain is decentralized and the smart contracts can describe a given business logic removing the human intervention, integrating the blockchain technology with IoT makes sense. However, it is necessary to keep in mind that blockchain and IoT domains are completely different; therefore, their integration together to obtain a combined architecture is challenging.

In the previous sections, it was already mentioned that blockchain technology uses consensus algorithms to achieve agreements on transaction validity and new blocks creation to the block-chain. The bottleneck of consensus algorithms is that they can require high com-

putational power or extensive communication between the participants. This technology was designed to be used in computers or servers with high enough computational power and large enough storage place. For example, Bitcoin takes more than 350 GBytes of data [33], and its size increases without stopping.

It is problematic to execute a blockchain in an IoT device because of its limited storage and computational power. In addition to these limits, another problem is the full connectivity of IoT devices. Providing fully connected IoT devices that change their position is also problematic. Another problem is that fully connected devices consume more energy than devices avoiding the connection as many times it is possible. Thus, the battery life can decrease radically.

One of the contributions of this thesis work is the study of how IoT devices can establish communication with a blockchain network and make part of it even if the IoT device does not contain a copy of the blockchain. The study also highlights the requirements of given blockchains (Ethereum, Hyperledger Sawtooth, EOS.IO, and Substrate) in order to be able to realize a blockchain-IoT network architecture. The thesis proposes basic and hybrid IoT-Blockchain architectures, which can be used in vehicle accident use cases and in use cases based on eco-systems in which the blockchain data size must be minimized.

This thesis also contributes in APIs written in C++, which meet all of the requirements for creating valid blockchain transactions. The transactions must meet specific requirements, and also essential cryptographic primitives must be applied.

The analysis of the APIs also demonstrates the importance of cryptographic primitives, which can eventually be hardware accelerated to obtain a faster execution and a minimized energy consumption. The hardware acceleration can be achieved by modeling a specific IoT hardware architecture, which is detailed below.

1.3 IoT architecture modeling for Blockchain applications

The previous sections mentioned that several IoT applications or IoT network structures require specific IoT architecture hardware to achieve a more efficient energy consumption and faster execution time. The study of multiple blockchains and deployment of IoT APIs for the given blockchain demonstrates that cryptographic hardware accelerators are required to achieve better performances.

1.3.1 Requirements for dedicated IoT architecture

This thesis work is based on a particular vehicle accident use case that is detailed in section 1.5. The main idea in the point of view of hardware-level is the equipment of the vehicles with dedicated IoT devices which allow sending transactions to the blockchain. The basic requirement of the IoT hardware architecture is to use an existing ARM-based CPU architecture as the core of the overall architecture. The overall architecture must allow running a Linux Operating System that can also execute the blockchain applications. In addition to these requirements, the overall architecture has to provide a low-power consumption, which can be achieved only by adding dedicated hardware accelerators.

The basic requirements can also be presented thanks to figure 1.3, which represents the layer of the IoT hardware model, the Linux OS layer, and finally, the layer of blockchain

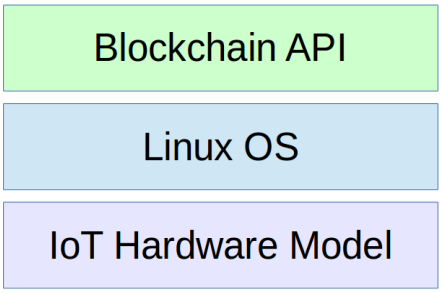


Figure 1.3: The basic requirements of the dedicated IoT architecture.

API launched by the OS.

Another important part of the contribution of this thesis work is the high-level modeling of a dedicated IoT hardware architecture meeting the given blockchains requirements. In particular, one of the main goals is to design a power-controlled hardware model which is optimized in terms of power consumption and execution time.

1.3.2 Methodology, modeling and simulation

The objective is to model a dedicated low-power-consumption IoT architecture operating with blockchain applications. It must be noted that this thesis work does not provide a final ASIC prototype of the architecture. However, it provides a high-level modeled power-managed IoT architecture. The model of this IoT architecture is based on two simulation components, the ARM-based CPU which is emulated by using QEMU [34] which is a tool that can emulate processors and complex architectures (e.g., ARM Versatile/AB (ARM926EJ-S) architecture). The Intellectual Properties (IPs) of the cryptographic primitives are modeled in SystemC-TLM [16] hardware description language (IEEE standardized C++ library). The reason for emulating and not modeling the CPU is that CPU architectures are relatively complex systems that would take years to model in any hardware description language.

It must be noted that the QEMU and SystemC-TLM are two completely different tools that have to be synchronized because they apply different notions of time. The thesis's contribution also demonstrates different co-simulation tools that can be used to synchronize QEMU with SystemC-TLM. The proposed IoT architecture model has been realized thanks to Xilinx Co-Simulation [35] [36] open-source tool.

It should be noted that the CPU executing the Linux OS cannot access the IPs directly. Therefore, dedicated Linux Kernel Device Drivers have also been deployed to allow the CPU to perform read/write access from/to the IPs.

The power management of the architecture is realized by a Unified Power Format (UPF) [37] like SystemC-TLM library called PwClkARCH [38] and by dedicated Kernel Device Drivers (also proposed by this thesis work).

1.4 Vital cryptographic primitives in IoT-Blockchain domain

Subsection 1.2.5 mentioned that this thesis work contributes to blockchain APIs, which can create valid blockchain transactions. The study about the APIs also identifies the essential cryptographic primitives needed to realize valid blockchain transactions. Another important contribution of this thesis work explains how these cryptographic primitives work, why these primitives are essentials, and also lists the existing hardware designs found in the scientific literature. The most efficient hardware designs in terms of power consumption and computing latency are modeled in SystemC TLM and used in the proposed IoT architecture model.

Thanks to the cryptographic primitives' identification, two basic primitives were found: the hash functions and the elliptic curve digital signature algorithms.

1.4.1 Cryptographic Hash

The theoretical definition of a hash function is that it takes an input message of arbitrary size and produces a unique hash value or hash digest of a fixed size. The emphasis is on the word "unique". The hash value can be seen as a fingerprint of the input message of the given hash function [14] [39]. The utility of hash functions is essential. For example, when a sender performs the hash of the data to send, the receiver can produce the hash of the received data. If the data was modified during the sending phase, it could be easily detected thanks to the uniqueness of the hash generation.

In blockchain-IoT architectures, hash algorithms are necessary for the digital signature process and when only the hash of the IoT data is stored in the blockchain. When only the hash of the data is stored in the blockchain, the data can be stored in another type of distributed ledger. The data then can be accessed thanks to its unique hash value. This method of storing data can avoid data falsification when storing it on a medium.

1.4.2 Elliptic Curve Digital Signature Algorithm

The other cryptographic primitives are the digital signature algorithms which are based on the private-public key signature schemes. It must be noted that the signature is not equivalent to encryption. The private key is used to sign the message to send. The public key can verify according to the signature and the data whether the data has been signed by the private key corresponding to the public key. The digital signature allows the verification of the data's provenance. In blockchain networks, each of the participants uses its private key to sign the transactions, and the corresponding public key is stored in the blockchain. If an unknown private key was used to sign a transaction, the transaction cannot be validated and added to the blockchain.

The particularity of these algorithms is that their operations are based on elliptic curve cryptography [40], which would be detailed in section 4.2 (p.84).

1.5 Smart IoT for Mobility Project

Professor François Verdier, a full-time professor at Université Côte d'Azur in the Department of Electronics, and an academic researcher in the LEAT laboratory, initiated the Smart IoT for Mobility project (multidisciplinary project). The first phase of the project started in 2017 with transdisciplinary objectives. In 2018 the project arrived in a second phase. It became a multidisciplinary project. From 2019 until 2021, the project is funded by the French National Research Agency (ANR) around 800.000 €. This thesis work was financed in the second phase of the project [41].

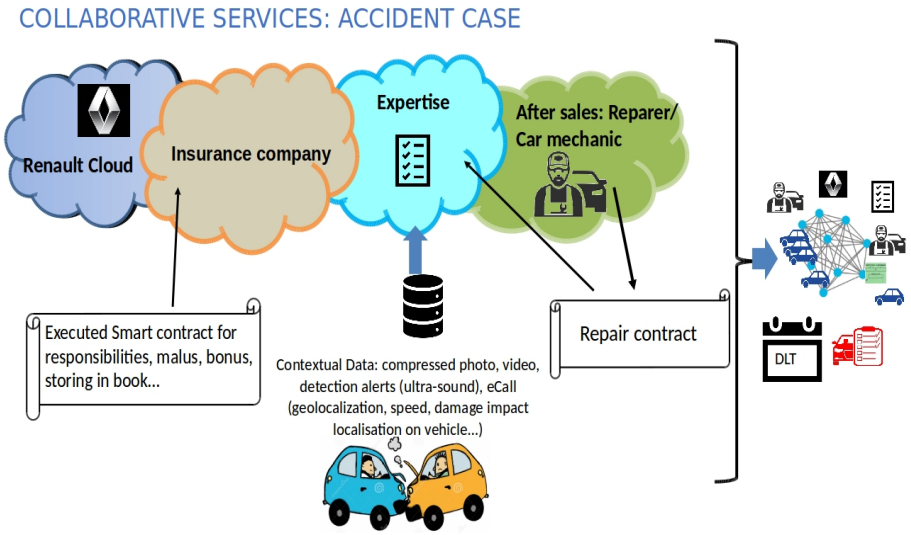


Figure 1.4: Renault's accident use case.

The basic idea of the project is a new ecosystem in which vehicles are connected to a blockchain-based network. Figure 1.4 represents the imagined ecosystem, in which multiple actors such as the car manufacturer (Renault), Insurance company, expertise, and car mechanics are present. It must be noted that the ecosystem members can have their own infrastructure (demonstrated by clouds); however, their infrastructures are connected by a blockchain-based system. Using a blockchain-based ecosystem can provide a transparent and trustworthy environment for data storing, and it also provides the safe execution of dedicated business logics thanks to smart contracts. The imagined ecosystem includes two main applications. The smart vehicle passport in which the maintenance history of the corresponding car is stored (cannot be modified thanks to blockchain features). The other is the car accident use case in which the vehicles are connected to a blockchain, and they send their recorded data about their environment when the accident occurred.

The project includes four disciplines. Economics are working on the acceptance of blockchain technology by the future end-users and industrial. Computer Scientists are working on natural language processing that allows the translation of any smart contracts to a usual language like English. Jurists of the project are examining how a smart contract can become juridically legal. Finally, researchers in electronics, studying the integration possibilities of blockchain technology with the IoT world.

It should be noted that this thesis work is based on a specific vehicle accident use case

in which the vehicles containing IoT devices record data about their environment before an accident happens. After a vehicle suffers an accident, the recorded data or its hash are transmitted to the blockchain. The interest in sending data about the car's environment or the driver's behavior can help experts or dedicated smart contracts determine the accident's faulty party and do the refund process faster and automatically.

Another interest in using blockchain is that the data about the accident is immutable, and the technology provides full traceability of the events in the system.

1.5.1 Management-Electronics multidisciplinary aspects

The previous section has described that the SIM is a multidisciplinary project which contains different disciplines. A part of this thesis contribution contains Management-Electronics research aspects. The management-electronic research aspect is based on interviews with enterprises such as Renault, Symag BNP Paris Bas, and Cardiff insurance company to better understand their needs and also anxieties about using distributed ledger technologies. According to this study based on prospective ergonomics [42], the goal is to propose technical solutions for future industrial real use cases taking part in blockchain-based ecosystems. Furthermore, to increase the acceptability and confidence in decentralized ledger technologies like blockchain. See more in Appendix A (p.165).

1.6 Objectives of the thesis

The contribution of this thesis work is based on two main branches. The first branch studies the integrations possibilities of IoT with blockchain technology. This branch also contains, but is not, limited to the development of blockchain APIs and the proposition of IoT-blockchain hybrid network architecture. The main goal of the contribution is to enable the execution of multiple types of blockchain APIs on the proposed IoT architecture. For that reason, this branch also provides a prosperous state of the art about blockchain, blockchain-IoT structures and also declares the specific requirements of the selected blockchains (Ethereum, Hyperledger Sawtooth, EOS.IO and Substrate).

The second branch of the thesis work identifies the most called cryptographic primitives that are required to create valid blockchain transactions. This part of the thesis also lists the existing hardware accelerator designs of the given cryptographic primitive. The most efficient designs are selected in terms of power consumption and computation latency to be applied in the proposed architecture model.

The third main objective of this thesis is first to develop a functional model of the proposed architecture, which also includes modeling the selected hardware accelerators and the development of dedicated Kernel device drivers. This part also highlights the challenges in the choice of the co-simulation tools. After the functional model is validated, the following objective is the power-managed architecture deployment.

The results consist of the overall energy consumption and total execution time of the power-managed architecture while running a given blockchain API. Accurate and realistic measurement of the power consumption of a given hardware architecture requires manufacturing details such as area, capacitance, and leakage resistance of the architecture depending on the CMOS technology used [43]. In general, the manufacturing details are not available

to the public but are treated as a trade secret by the circuit manufacturer.

It should be noted that the energy consumptions and execution times are retrieved according to the estimations of the manufacturing details mentioned previously. Hence, the values of the result can be different when implementing the proposed architecture on ASIC.

1.7 Thesis structure

Figure 1.5 represents the structure of this thesis work. Chapter 1 can be compared to a genesis block (first block) of a blockchain in which all of the required parameters are declared to initiate a blockchain system. This chapter describes the context of the thesis work and some of its basic contributions. It also explains the fundamental objectives, challenges, problematics, and questions that would be detailed, highlighted, and answered in the following structure.

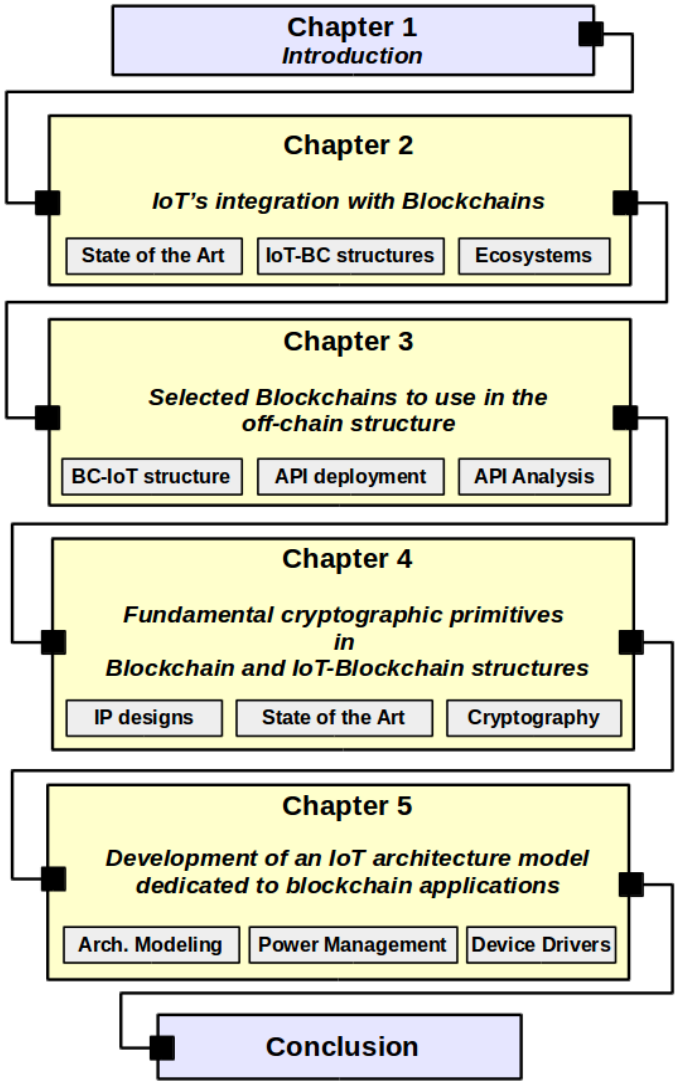


Figure 1.5: Thesis Structure

Chapter 2 (p.13) is like the first valid block of the blockchain, which is now accepted by all network participants and whose data cannot be deleted anymore. This chapter provides an essential state of the art about blockchain technology, and it also studies different approaches

to integrate IoT with blockchain. One of the main objectives of the thesis contribution is to allow the executions of different types of blockchain APIs on the proposed IoT architecture.

For that reason, the chapter 3 (p.45) highlights the specific requirements of Ethereum, Hyperledger Sawtooth, EOS.IO, and Substrate blockchains. Highlighting these requirements allows the deployment of APIs which can create valid blockchain transactions. The chapter also identifies the most called cryptographic functions, which play an essential role in the valid blockchain transaction creation. Thanks to the identification, it is also demonstrated in which use cases the given cryptographic primitives are used more or less. The chapter, finally, proposes a hybrid blockchain-IoT network architecture that can optimize the quickly growing size of the blockchain and the authentication of IoT devices in a given ecosystem.

Chapter 4 (p.83) explains the importance and why the given cryptographic primitives are used in blockchain technology. The chapter also gives a brief mathematical description of the cryptographic hash and digital signature algorithms. The main result of the chapter is the list of existing hardware accelerator designs (IPs) of the cryptographic functions found in the scientific literature. The chapter also highlights which designs should be selected to implement in the proposed IoT architecture model.

Chapter 5 (p.107) also provides a considerable part of the thesis contribution. The first part of this chapter discusses the selected tools for architecture modeling and the challenges in the co-simulation. This part also describes the utility of the PwClkARCH SystemC-TLM library, allowing the power consumption monitoring and management of the designed architecture. The chapter also demonstrates the proposed SystemC models of the cryptographic IPs selected in chapter 3 (p.45). The chapter also represents the deployment of the dedicated Kernel device drivers, which allow the communication between the CPU (emulated by QEMU) and the given hardware accelerator IP. The chapter takes emphasis on the functional and power-managed model of the architecture. For applying a power-managed model, a power management orchestration is required, deployed with the PwClkARCH tool and a dedicated Kernel device driver. This chapter also estimates the dynamic and static power consumption of the emulated ARM-based CPU (Cortex-A9) and the modeled cryptographic hardware accelerator designs. This estimation is needed because PwClkARCH uses parameters such as the capacitance and leakage resistance of ASIC designs. ***Thanks to these estimations, the proposed architecture's overall energy consumption and execution time can be retrieved when the architecture runs the blockchain APIs.***

All of the results are retrieved in order to be able to compare the overall energy consumption and total execution time of the APIs' execution, when power management was applied (the hardware accelerators are called) and when no power management is applied (hardware accelerators was not called).

This thesis fork finishes with a conclusion of the overall contribution and perspectives that should be done in future works.

Chapter 2

IoT's integration with Blockchains

2.1 Introduction

Nowadays the Internet of Things (IoT) domain has become more than "devices" or "things" that are connected to the Internet. The Internet of Things can be considered as a set of electronic devices (physical layer) establishing communication among them and between the Internet (communication layer). IoT domain includes a wild range of different devices, from small wearable device architectures to specific System on Chips (SoC) such as cell phones for example. The IoT has become an important set of the Wireless Sensors Networks (WSN) [2], and IoT devices can also be viewed as embedded systems with certain constraints as the limited battery life, limited memory footprint, limited size, and computational power that is significantly lower than today's computers. According to [44], the IoT can be considered as "an evolution of data communication" allowing direct device-to-device communication.

These connected devices encompass us, they participate in peoples' everyday life, and the number of these devices increases quickly. According to statistic estimations, the number of connected devices would reach more than 50 billion by 2025 [20] and the active device connections would be near to 30 billion [21]. The use cases and applications in the IoT domain are large, IoTs are used in many Smart applications forming Smart Cities, Smart Homes, Smart Farms, or even Smart Grids. IoTs are inevitable in the industry (e.g. M2M), in e-health (e.g. examine of a patient's being), in sensing applications (e.g. to control of machines' behavior), in monitoring (e.g. meteorology service), in Artificial Intelligence-based applications, in vehicular networks, and Intelligent Transportation Systems, etc. It can be assumed that these connected devices are used and can be used in almost every application domain. The common point of all of these IoT applications is the data. This huge amount of collected data is needed to facilitate peoples' daily life, improve productivity, or simply better understand certain domain-specific problems.

In general, the IoT data is stored in a traditional centralized manner (in servers of a centralized authority). After the data is stored, a specific data process can be done according to the given application. The data provenance and the trustworthiness of the data is still an important issue, because the IoT devices send data to a central authority such as a cloud. IoT data can be manipulated and falsified. In addition to data manipulation, the IoT devices of a given system are not necessarily authenticated, or if it is the case, they are authenticated by a central unit. In an ecosystem ideally containing several members and dedicated business logic, this central unit or authority can be considered as a trusted third party that has special control over the incoming data. This trusted party can be also considered as a weak point

of such a centralized system. [2] [44] highlight that the combination of IoT and blockchain technology can provide the data trustworthiness of the IoT network thanks to the nature of the blockchain technology, which is decentralized, distributed, and does not contain a central authority (or trusted third party). This data reliability can be achieved thanks to the main features of blockchain technology such as once data is stored in the blockchain it becomes immutable and every event that happens in a blockchain network is transparent for every blockchain network participant. Blockchain is also a Peer-to-Peer (P2P) network, every participant of the system is known for each other, thus the data provenance is also transparent. In a blockchain-IoT system the IoT devices could be authenticated and a distributed authorization logic could also be held, thus the need for a trusted third party can also be removed from such a system.

In theory, blockchain technology is scaled for computers with sufficient memory for storing a large amount of data, regardless the complexity and the power consumption of cryptographic primitives used to make this system architecture even more secure. The main challenge of integrating IoT with blockchain is the valid transaction creation in the constrained IoT devices, which allows the communication with the given blockchains, and therefore allows obtaining a more secure overall network system.

2.2 Blockchains

Most often, the blockchain is known as a distributed data storage. Unlike a traditional data storage system in which data is stored by a central entity (e.g. server, cloud), and managed by a central organism (or trusted third party), blockchain is considered as a distributed database, with each member of the network holding the same copy of the stored data, without the help of a central entity. The blockchain is not only a special data structure, it is also a P2P network, in which all peers are connected to each other (Figure 2.1).

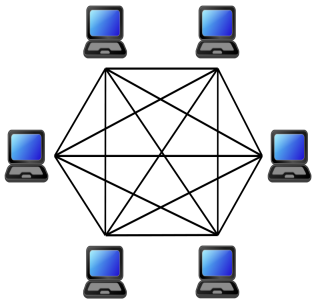


Figure 2.1: Examples of a Peer-to-Peer network

In a P2P network, there is no central entity that manages the identification of peers, this process takes place in a decentralized way by using private/public cryptographic key pairs. Each peer has its own unique private/public key pair. The private key is used to sign the transaction (containing a certain payload) that the peer wants to transmit to the blockchain. The public key of the peer who transmits the transaction is known to the other peers. Using this public key, the provenance of the transaction can be verified, proving that the transaction was sent by a given member of the blockchain. The use of digital signature techniques ensures the reliability of the data. Digital signature methods are one of the key points of IoT

integration with the blockchain. The subsection 2.2.1 provides more details on these digital signature algorithms and their use in IoT architectures.

The basic idea behind the blockchain is to remove the third party, and achieving a decentralized authority without depending on any central entity. In addition to this feature, blockchain provides data trustworthiness and data immutability by using several cryptographic primitives and techniques which would be discussed in subsection 2.2.1.

Before going into technical details of blockchain technology, this section helps to better understand the idea of the blockchain and its features. The first implementation of the blockchain was published by Satoshi Nakamoto in 2008[25]. Today, this blockchain is better known as Bitcoin. This system was created to make direct online payments between network members without going through a central trusted third party (or central entity). The Figure 2.2 represents the traditional transfer of currency between participants A and B.

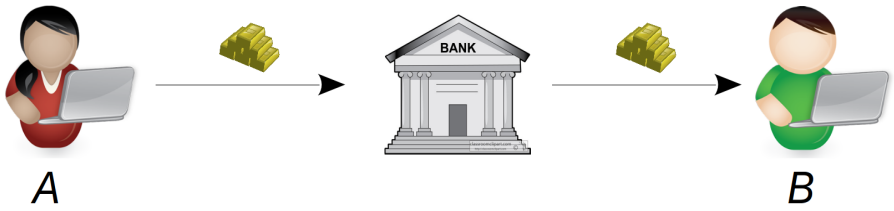


Figure 2.2: Examples of centralized systems

In this case, the transfer is performed through a financial institution (e.g., a bank) that acts as a trusted third party. The use of a trusted third party in a financial transaction system and in complex ecosystems can have several drawbacks. In systems related to finance, there are cases when the intermediate limits the minimum practical transaction size, it can also increase the transaction costs, it can also refuse the transaction because of other policies that the participants do not want to accept. Other disadvantages in ecosystems are, the intermediate can stop working or only work regularly, the third party can be attacked and the participants' data can be stolen by the attacker, or a successful attack can provoke the systems to behave incorrectly. For examples, an incorrect behavior of the centralized entity can occur in real-world situations, such as when banking applications are temporarily shut down or when person A wants to sell his house to person B, so they have to sign a contract that is held by a notary, but the process stops temporarily, until the notary returns from his vacation. These types of drawbacks can be critical in systems depending on a central entity, which leads us to the following question: Can the system/ecosystem members trust the central entity?

According to Nakamoto's Bitcoin the third trusted party can be replaced with the decentralized trust that the blockchain technology provides. The figure 2.3 represents how the central entity is replaced by using a blockchain. It can be noticed that all of the participants can send transactions directly to each other and that the participants have the same data (copy of the blockchain) that can ensure data integrity and transparency. These data include not only the transactions that are sent between members, but also the records of actions that occur on the blockchain.

One of the aspects that can replace the trusted third party is the traceability of actions taking place on the blockchain. The traces are visible to each member, which allows the participants to make decisions about the trustworthiness of the actions. It should be noted that the data transparency itself does not provide sufficient protection against faulty mem-

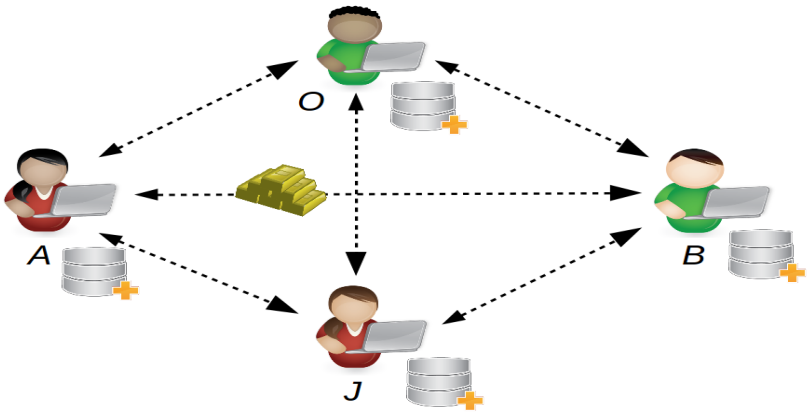


Figure 2.3: Examples of a decentralized systems. Each of the members owns the same database.

bers attempting to manipulate the data or to generate actions that cause the system to behave wrongly. In order to ensure that the data is the same for all members, a decentralized consensus mechanism (e.g., Proof-of-Work in Bitcoin) is used. This consensus mechanism solves the problem of the majority decision making, i.e., which data should be kept or added to the existing data by the members. For better understanding how the consensus mechanism is deployed and how blockchain provides immutable data storage to achieve the desired decentralized trust the reader is encouraged to continue reading the following subsection.

2.2.1 Blockchains' structures

This section describes how the blockchain structure is built and highlights how this technology provides permanent/immutable data storage, how data transparency is ensured by a consensus mechanism, and finally how the data trustworthiness is guaranteed.

New data can be added to the blockchain by transferring it in the form of a transaction. The size of the transaction depends on the implementation of the blockchain. In the case of bitcoin, this size is limited to 1MB [45]. The newly arrived transaction has to be validated by all of the participants. After the transaction validation, a set of validated transactions will be added to a new block of the blockchain.

The validation process consists of two basic key points, first, the transaction has to be signed digitally by the transmitter. Using this digital signature method allows other participants to verify if the sent transaction corresponds to the transmitter, thus identify the sender of the transaction. The transaction would be directly rejected if the signature does not correspond to the signer. The principle of digital signature and its verification is shown on figure 2.4. The principle of the digital signature and its verification is an essential procedure in IoT-Blockchain based application, and it is based on the following scenario: Alice wants to send a message m to Bob, and Bob wants to be sure that it was Alice who sent the message.

This scenario is also used in the mailboxes. Note that the purpose of the signature is to verify the origin or authentication of the message (i.e., to identify who the sender is, in this case, Alice). The message m will not be encrypted, it will only be signed. It was already mentioned earlier that each peer in the blockchain has its unique pairs of private and public keys, and each peer contains all the public keys of the other participants, the public key is also considered as the identifier of a peer in the network. The private key is known only to its owner,

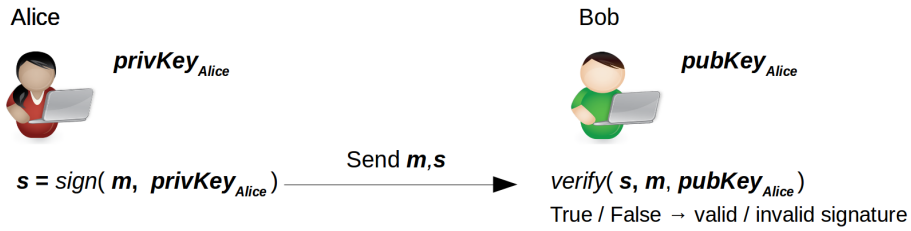


Figure 2.4: The principle of the digital signature and its verification

it is a secret key that should never be distributed. In this section, the signature/verification procedures are described simply, a more detailed description of cryptographic algorithms is given in subsection 4.2.3 (p.87) and 4.2.4 (p.88).

Alice has her private key ($privKey_{Alice}$) which is used to sign the message m. The signing algorithm uses Alice's private key ($privKey_{Alice}$) and the message (m) to generate the message signature (s). When Alice sends her message m, she sends it with the corresponding message signature s. Bob receives the message and the message signature. Bob knows Alice's public key ($pubKey_{Alice}$). Since the public keys are known to each participant, Bob performs a verification function with the message (m), the message signature (s), and Alice's public key ($pubKey_{Alice}$) as inputs. If the result of the function is true, it means that the message has been signed by Alice and the origin of the message is confirmed. Otherwise, the message was not sent by Alice and Bob must therefore reject the message, he cannot trust the content of the message and the identity of the sender of the message.

Transaction and transaction payloads are formed of binary-encoded data. Thus the second non-trivial key point is the respect of the transaction form that is related to the given blockchain implementation. The encoding method is used to serialize the transaction sent by the sender, which can be deserialized (i.e., "unpacked") in the blockchain, and can be used in smart contracts for example.

The data serialization of a transaction can be seen as a structure of information according to a certain logic that is given by the blockchain implementation. This encoded information is used to execute the desired actions on the blockchain according to the content of the transaction.

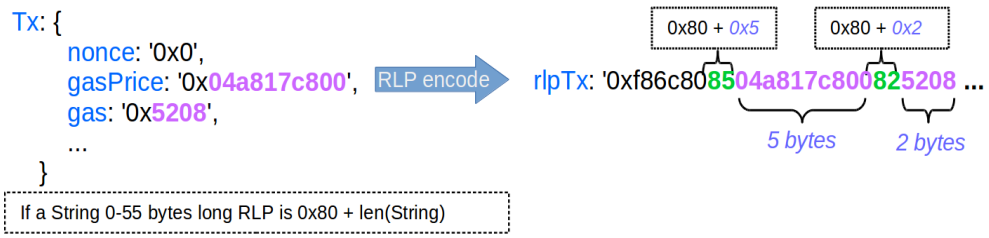


Figure 2.5: Examples of RLP encoding, used in Ethereum blockchain

In the case of Ethereum blockchain, RLP (Recursive Length Prefix) encoding is used, this encoding method encodes arrays of arbitrarily nested binary data. Figure 2.5 represents an example of RLP encoding of an Ethereum transaction. The "rlpTx" is equal to the encoded transaction "Tx", as it can be seen, when the length of a string digest is in the range of 0-55 bytes the fixed value is incremented with this string length, and this value is set before the string digest. Some blockchains support CBOR and Google Protocol Buffer format (e.g.,

Hyperledger Sawtooth). If the transaction's form is not respected the transaction could be rejected or simply the desired action encoded by the transaction would not be performed.

The validated transactions are added to a block, Figure 2.6 shows that transactions are present in the block body. A new block containing transactions has to be committed (validated) according to the consensus rule that is used in the blockchain. The block validation and adding is also an important mechanism of the blockchain. The participants have to agree on which block has to be added and which member would add this validated block.

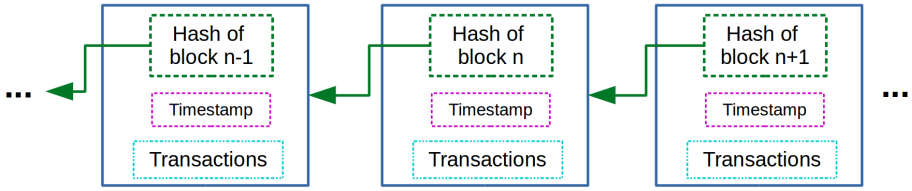


Figure 2.6: Examples of a blockchain

Thanks to the consensus mechanism, it is possible to deny faulty participants from committing blocks, and it is possible to obtain a trusted environment with the majority of honest members. In addition to filtering faulty members, the consensus mechanism also plays a synchronization role in block scheduling/adding [45] [46]. The goal is to obtain blocks that follow each other and finally form a chain of blocks, as shown in Figure 2.6. Each member is allowed to send transactions to be validated, however not every member is authorized to validate blocks. The nodes which are allowed to validate blocks are called validators or miners.

Each of the validators wants to create and add a new block to the existing ones. If no consensus role was applied, there would be no common agreement on which block can be added or not. A consensus rule also synchronizes which validator has the right to add a new block and when. Adding blocks without synchronization would also lead to trust issues, as it would be impossible to know whether or not the content of the block was manipulated by a given validator. It is worth noting that the consensus rule is used to order the process of adding blocks (when and which validator adds it to the chain). However, it often happens that a newly created block is added at the same time, which can create a fork or orphaned nodes in the blockchain. It should also be noted that the fork phenomenon also depends on the consensus rule used.

In general, consensuses designed to support large networks are not definitive and thus may contain forks. The figure 2.7 represents an example of a fork, as we can see the chain created by blue blocks is the longest chain (trusted chain) and the ”top” or last trusted block of the blockchain is block 9. The chain built by the orange blocks is the ”fork”. Which fork is selected as the main fork or the main chain depends on the consensus algorithm.

It was mentioned earlier that the blockchain should provide a trusted environment, all of the consensus rules are developed to provide a trusted environment by resolving a Byzantine faults [47]. Byzantine faults or Byzantine Generals problem is an example of a trustless environment. According to Lamport *et al.* [48], authors imagine a situation in which groups of Byzantine army commanded by their own generals encircle an enemy city. The generals must decide together (common decision) on a plan of action in order to succeed: attack or retreat. Generals can communicate with each other only via a dedicated end-to-end channel.

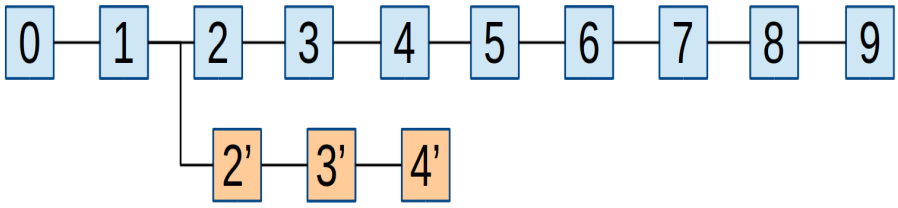


Figure 2.7: Examples of a blockchain with a fork in orange and the main chain in blue

In a network, the generals can be seen as network nodes. Among the generals, there might be traitors (malicious nodes in the network) who could send a different decision (attack or retreat) to different generals. The malicious behavior of traitorous generals prevents honest generals from reaching a common agreement on attack or retreat, which also prevents honest generals from succeeding in their actions.

All consensus implementations of blockchain and distributed ledger technologies are used to be able to reach a common agreement of the participants on the order of adding the new blocks in the blockchain [27].

Today there are a lot of existing consensus rules (about 30 [28]) used in blockchain and distributed ledger technologies. In this part of the section, an introduction is given about some of the most popular consensus rules which were used in the studied blockchains. However, it should be noted that this thesis contribution does not focus on the consensus algorithms adaptations for IoT.

1. *Proof-of-Work (PoW)*

   The idea of a consensus rule is older than the blockchain technology, and it was invented against email spamming [29]. The basic idea behind the PoW is that the sender has to compute a computationally hard mathematical task before sending his/her mail. The receiver accepts the received mail if and only if the mathematical task solution is correct. Nakamoto's Bitcoin [25] and V. Buterin's Ethereum [26] use the same concept, the miner nodes (participating in the block validation process) compute a resource-intensive task for solving a cryptographic puzzle. The base of this cryptographic problem is computing a hash value of a block header that changes regularly. The minor nodes are in contest with each other, each of them tries to solve the cryptographic hash as fast as possible. The fastest node which found the hash value informs the other miners, these last verify and confirm the correctness of the solution. When the solution is correct the miner node has a law to add the block to the blockchain and this last would be rewarded with a certain amount of cryptocurrency. This cryptographic puzzle-solving procedure is better known as crypto mining. In the case of PoW as long as the half of the network is composed of honest nodes, the blockchain is considered a trusted environment.

   The PoW is known as the most CPU-intensive algorithm, which consumes a high amount of energy (electricity). The yearly power consumption of Bitcoin and Ethereum is 40 and 10 TWh [49]. It can be summarized that mining in IoT devices is not possible and also does not make sense because these embedded devices would not be able to compete with the computers and dedicated ASICs that compute PoW. According to [50], in the Raspberry Pi 3 B+ model, which is a complex and powerful IoT device,

the Ethereum PoW caused the CPU to overheat, rendering the card unusable. For further technical details and implementations of the Proof-of-Work, the reader is kindly invited to read the following article [27].

2. *Proof-of-Stake (PoS)*

   The Proof-of-Stake algorithm is another alternative to PoW, however, this algorithm can be computed with less energy consumption. In this consensus rule the nodes do not resolve a cryptographic puzzle. Nodes can participate in block committing if they own a minimum value of cryptocurrency called a stake. Between stakeholders, the decision of which one has the right to commit a new block is determined randomly. When a malicious node is detected, it loses its stake and therefore cannot commit any more blocks. The idea behind this consensus is that stakeholders have an interest in remaining to be honest in order to gain more cryptocurrency [49] [45] [2]. This consensus is used in Peercoin for example, and Ethereum also wants to apply this consensus in the future.

3. *Practical Byzantine Fault Tolerance (PBFT)*

   Practical Byzantine Fault Tolerance is a consensus algorithm based on a replication algorithm, which is resilient against Byzantine faults [30]. To solve this problem a given system is considered as a black box in which replicas are executed and their results are compared, if 2/3 of the results are the same the system remains trusted. In the case of the Hypelrdger Fabric blockchain [31], the committing of the new block and the choice of the node that adds it to the blockchain are divided into several phases that are based on specific rules. In each phase, 2/3 of all nodes must vote (thus agree) to move on to the next phases until the block is added. This consensus is often used in blockchain networks with few nodes (in private blockchain, see on page 24). PBFT is based on voting to reach agreements; thanks to this feature, this consensus algorithm has a finality or determinism that could not cause forks and orphan blocks in the blockchain.

4. *Proof-of-Elapsed Time (PoET)*

   This consensus is notably used by the Hyperledge Sawtooth [51] blockchain. Proof-of-Elapsed Time consensus works as follows: validator nodes request a wait time which follows a probability distribution. The node with the shortest wait time becomes the leader, which can generate a new block. Before adding this newly created block, the other nodes verify if the leader node has not cheated with the wait time [52]. If it has cheated the block would not be added, and the node can be blacklisted or dropped form the network. PoET was created by Intel's Software Guard Extensions (SGX), a hardware design that provides a trusted environment for code execution and protection against data manipulation. However, the use of SGX is not a requirement, this environment can also be simulated, so PoET can be deployed on regular PCs [53].

5. *Delegated Proof of Stake (DPoS)*

   Delegated proof of stake was first implemented by Daniel Larimer [54]. This consensus is another Proof-of-Stake alternative in which stakeholders elect their delegates who are responsible for generating and validating the blocks. The voting power of

stakeholders is proportional to their wealth (stake). Contrary to PoS, in DPoS the number of validators, and thus of delegates, is limited (generally between 21 and 100). The malicious delegate can easily be forbidden to validate the blocks, by initiating a new election of delegates [55] [28]. This consensus is partially used in EOS.IO blockchain [56].

In figure 2.6 we can also observe that a block contains the hash of the content of the previous block, also called a parent block. A cryptographic hash function is used to obtain a message digest of fixed length as the output of the function from an input message of arbitrary length [57] [14]. The result of the hash function is called a hash, which is a unique representation of the input message. This function can also be considered as a compression of the input message to a fixed-length output message [39]. This cryptographic primitive is widely used in authentication systems thanks to the feature that the hash value is unique for the input message and the cryptographic hash functions are one way functions, meaning that the results are not reversible. To give an example, in order to prove that A's message was not altered when it was sent to B, A would also send the hash value of the message, B can calculate the same hash function on A's message. Modifying a single bit of the message results in a completely different hash value, thus B can recognize easily a message corruption.

In the structure of the blockchain, the newly added block contains the hash of its parent block, which contains the hash of its parent block, and so on until the first block of the blockchain called genesis. It should be noted that basically, each block of the blockchain depends on the hash of the previous block (a unique value). This dependent connection between the blocks creates the chain structure and the immutability of the blockchain. Block hash values are also used to detect data corruption. If a node attempts to alter the data in a block, the hash of that block will be different from the hash value computed in the other nodes. Using this hash technique makes easier filtering out the faulty nodes of the network.

2.2.2 Smart Contracts - Blockchain 2.0

In the previous sections, the reader could see the basic characteristics of blockchain technology, its structure, and an introduction to the consensus rules that can be used in this technology. The reader was also able to understand that the Bitcoin blockchain has revolutionized digital transactions of cryptocurrencies between network participants without the need for a trusted third party.

Bitcoin can be considered the father of other types of blockchains, as its main features are integrated into most blockchains used today. One of the limitations of Bitcoin is that the technology was developed for financial transactions and cryptocurrency trading, however blockchain can provide more, this technology provides a trusted environment, without centralization, and with full data transparency.

Vitalik Buterin, better known as the inventor of the Ethereum blockchain, came up with the idea of keeping the basic functions of the blockchain related to finance and cryptocurrencies, but in addition to these functions, he wanted to enable the deployment of smart contracts on the blockchain. With the first implementation of the Ethereum blockchain in 2013 [26], Ethereum enabled the deployment of smart contracts and this functionality gave birth to the era we call Blockchain 2.0.

The first idea for smart contracts was thought up by Nick Szabo in 1997 [32]. A smart contract is a digital program, a computer code that can be executed automatically. The goal

of a smart contract is to be able to realize the same contractual agreements as it is written in traditional paper format contracts. Figure 2.8 represents an illustration of a smart contract. Another important objective of smart contracts is the exclusion of trusted third parties (e.g., notary, lawyer) from the agreements, using cryptographic mechanisms.

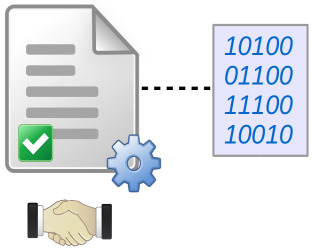


Figure 2.8: Example of an illustration of a Smart Contract

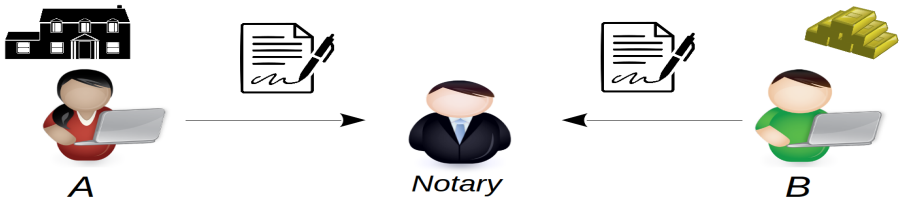
A digital program that can replace a traditional contract without including a trusted third party can accelerate the execution of a given agreement or business logic. There is no doubt that the automation of these agreement procedures could facilitate the work of many domains and people. However, how to ensure that the execution of these smart contracts is safe (i.e., that their results will not be manipulated)? Another problem is, how to be sure that a smart contract code that is visible to users will not be modified before its execution producing an unattended result?

The solution is the integration of smart contract with blockchain technology. Thanks to the basic features of blockchain technology (e.g., data immutability, data transparency), it can provide a trusted execution environment for smart contracts. Therefore, when the smart contract is being executed it cannot be manipulated. In addition to the trusted execution environment, once a smart contract is deployed to the blockchain it cannot be modified or deleted anymore, and the content of the smart contract is readable[1] for all participants. The smart contracts can be executed automatically thanks to blockchain events. A participant can execute the functions defined in the smart contracts by sending transactions pointing to the given smart contract's address. After a smart contract is executed, the blockchain would contain certain logs related to the execution, which also provides transparency. In special cases, a smart contract can even call other smart contracts, and thus establish a complex business logic.

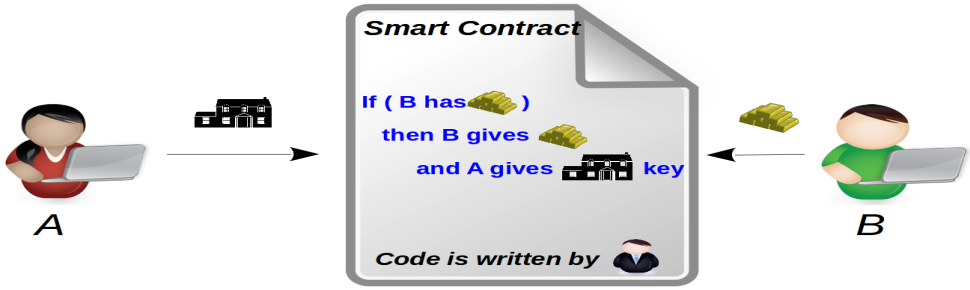
Section 2.2 earlier presented an example in which the notary can slow down the procedure of selling a house (shown on figure 2.9a), this inconvenience can be avoided and the whole procedure can be accelerated with the help of a smart contract, which replaces the notary and automates the sale procedure (represented in figure 2.9b). The content of the smart contract is highly simplified, this example has been given to show an idea of a possible realistic use case.

It should be noted that removing the notary from this process, would cause juridical problems, because today most smart contracts are not juridically legal. One of the main aims of Smart IoT for Mobility (SIM) project is to find a solution for the legalization of smart contracts. In this project, jurists and computer scientists are working on the translation of paper-based contracts into smart contracts. Different types of languages can be used to

[1]It is a general description of the integration of Smart Contract with Blockchain, private blockchains can restrict reading access of smart contracts, but it is not typical (See more in subsection 2.2.3)



(a) Centralized system, with the notary in the middle of member A and B



(b) Decentralized system, smart contract replaces the notary

Figure 2.9: Example of an application using Smart Contract

create a smart contract, for example, Solidity, Scilla, Java-Script, Python, C++. All these languages are programming languages, which is a disadvantage because non-programmers cannot understand the content of a given smart contract. In the SIM project, a team of computer scientists is working on the Natural Language process, which allows the interpretation of smart contracts in English.

It is worth noting that today, the Ricardian contract [58] can be considered as a tool to better understand the behavior of a smart contract according to its input parameters. The tool provides a translation of the behavior of smart contracts into basic English keywords, which makes it understandable also for non-programmers. In some U.S states, Ricardian Contracts are juridically legal, but they only describe basic financial agreement.

According to L. Gerrits [59], today the smart contracts or Ricardian Contracts are not contracts, they are agreements between the parties.

It can be concluded that even if smart contracts are not understandable for non-programmers today, there is no doubt that smart contracts deployed in blockchains technology could revolutionize many domains (not only the financial domain). Obviously, there are many applications from different domains, this subsection ends with some examples of possible applications from the leading domains.

- Blockchain with smart contracts can be used in traceability systems [60] that include multiple organizations and suppliers and retailers. In this application area, entire business processes can be described in smart contracts, with special agreements between suppliers and retailers for example.

- A supply chain is based on a consortium of companies, the use of blockchain and smart contracts can simplify business-to-business integration and also simplify machine-to-machine communication. Using blockchain can improve the visibility of deliveries

and leverage blockchain's tracking and tracing capabilities for the supply chain management. In food traceability applications, food contamination can be detected, and lives can be saved [2].

- In eHealth applications, blockchain and smart contracts can be used to remove intermediation in managing access to the electronic health record [61]. A patient's medical record may contain sensitive data, which can be accessed if and only if the patient authorizes it. Moreover, the patient does not necessarily trust an intermediary who can potentially manipulate or distribute their data.

- The government sector, especially e-governance, is also using blockchain technology and smart contracts. For example, the Estonian government is using blockchain to secure the nation's health, justice, legislative, security, and commercial code systems. The Estonian government has achieved 99% of public services being online 24/7 using blockchain technology [62].

- Artificial intelligence (AI) can also benefit from the features of blockchain and smart contracts. Authors of [63] provide an overview of AI integration with blockchain. The authors also points out that machine learning uses a large data set that is centralized and therefore not tamper-proof. Blockchain technology can provide tamper-proof data and it is also possible for the machine learning algorithm to be executed in a smart contract in a decentralized manner. AI-Blockchain also enables machine learning programs to be run as a decentralized application (dApp) [64] that are secured by the blockchain and smart contract features.

- Blockchain and smart contracts can also be used in data access management systems, reputation systems, and more. There are also a many IoT-Blockchain applications, these applications and in particular, the ones that inspired this thesis work or served as a promising idea are presented in section 2.3.

2.2.3 Blockchain types

Generally in blockchain technology, three groups of blockchains can be distinguished: public, private, and consortium.

In these three groups, the common feature is that these blockchains are decentralized peer-to-peer networks, and the data contained in these blockchains are distributed among the participants. All these blockchains use a certain consensus mechanism to be able to achieve the agreement on block adding. It can be noted that the basic technological features are the same for all of these three groups, however, because of some characteristics, it is important to distinguish them [65] [66].

- *Public Blockchain*

  A public blockchain is open to everyone, everyone can send transactions and read the state (the data of the blockchain, which is permanent) of the blockchain. In a public blockchain, all members are allowed to participate in the consensus. One of the main features of public blockchains is the transit of cryptocurrencies. Generally, PoW or PoS consensus algorithms are used, which makes sense because the miners (validators

calculating PoW or PoS) of the blockchain are rewarded after validating a block. Some of the most known public blockchains are Ethereum, Bitcoin, and EOS.IO.

- *Private Blockchain*

  Typically a private blockchain is deployed by a company or organization, which means that the organization can decide to restrict read and write rights for particular nodes. The organization can more easily change the rules or create business logic regarding the addition of new validators or participants to the blockchain, e.g., voting logic can be deployed using Smart Contracts, all validator nodes must agree on the addition of a new validator, or a new participant can be added to the blockchain after certain specific procedures that can be described in the given smart contract. In private blockchains, the presence of cryptocurrency is not a necessity, and thus there is no need for difficult and resource-intensive consensus rules like PoW. The main aim of these private blockchains is to obtain a trusted and secure environment in which an organization can use smart contracts to automate certain processes and business logic. This type of blockchain also contains fewer participants than the public blockchain. With fewer participants and lighter consensus rules, the private blockchain can produce faster transaction validation rate. An advantage of the private blockchain is that every validation node is known, making the risk of a 51% attack less likely (if the majority of validators control the computing power of the network). Some of the most known private blockchains are Hyperledger Fabric, Hyperledger Sawtooth, R3 Corda, and Quorum (merged from Ethereum).

- *Consortium Blockchain*

  A consortium blockchain is maintained by more than one organization and the group of nodes participating in the validation process is pre-defined. Reading and writing rights can be limited as in the case of private blockchains.

It should be noted that there are several public blockchains like Ethereum or EOS.IO that can also be developed as private blockchain.

2.2.4 Difference between blockchain and DAG technology

It is worth noting that blockchain is the best known of the Distributed Ledger Technologies (DLTs), but other technologies similar to blockchain are also part of DLTs. In these blockchain-like technologies, the data structure does not form a chain with blocks but they are based on Directed Acyclic Graph (DAG). The DAG is composed of edges and vertices. A directed graph means that there is no looping from a child vertex to a previous (parent or ancestor) vertex (see figure 2.10). In general, the vertices are blocks or transactions forming a graph, and the blocks or transactions are linked together by hashes, in the same way as in the blockchain [67].

In the literature, it is often mentioned that DAG data structure can be more scalable than blockchain technology with a higher transaction validation rate[2] and is easier to adapt for IoT technology. IOTA is one of the most popular DLTs using a DAG data structure that is called

[2] The transaction validation rate is one of the most important characteristics of a given blockchain, the section 2.3 describes the importance of this feature with more details.

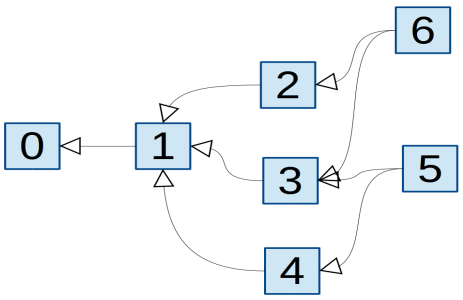


Figure 2.10: Example of a Directed Acyclic Graph

a Tangle. IOTA was created for the transfer of microtransactions at no cost between network participants, however, IOTA allows to include data in its transactions. In the Tangle of IOTA [68], a vertex of the graph is a transaction. When a new transaction arrives at the end of the Tangle, it is called a tip. This tip is selected using a tip selection algorithm, which performs a random walk from the genesis transaction to the last transaction to be proven by the tip. This algorithm uses Markov chain Monte Carlo method to ensure that each step in the walk does not depend on the previous one, but follows a decision rule of the next transaction. The tip must approve two unapproved transactions, which means that the node that sent the tip must perform a lightweight PoW (this is a less resource-intensive PoW that is used in Ethereum).

In the IOTA network structure, there are two types of nodes. Client nodes send their signed transactions to IRI nodes (usually installed on servers) that perform the tip selection and PoW, proving both transactions. The client node can also run a PoW – in the documentation, it is referred to be a local PoW – if it has sufficient computing power. It may be noted that the client has to request IRI to know which transactions it should approve and it has to call the IRI commands to broadcast its transaction that it wants to add to the Tangle. A client must make at least 6 HTTP requests when performing PoW locally and 4 in the other case[3].

In IoT applications using embedded architectures with limited resources, the number of message sending (in this case HTTP requests) should be reduced because sending messages can take a long time and can consume a large amount of energy (depending on the protocol). Another drawback highlighted by the literature [69] [70] [71] [72] [73] describing DLT applications with IOTA, is that the deployment of smart contracts is still not available on this platform. Until 2021, IOTA proposed to use Qubic networks that could execute smart contracts, but this proof-of-concept project is not maintained for two years. A GoShimmer project was launched in the same year, which would probably allow the execution of smart contracts soon.

Another DLT using DAG is called Aleph Zero [74] and uses a similar structure to IOTA, with a transaction validation rate of 100k transaction per second (theoretical value), but today the deployment of smart contracts is not yet possible on this platform.

Hashgraph [75] is another DLT that uses a gossip protocol to distribute information among participants, participants randomly choose other participants to communicate gossip to. The Hashgraph contains information about who communicated with whom. An event in this system, in addition to containing data, contains the hashes of its two parent events,

[3]These measurements were done by using iota.c library, that can be found on https://github.com/iotaledger/iota.c

forming a graph. To realize a BFT (Bizantyne Faulte Tolerance), the participants must vote to decide what should happen in the system.

In the case of the Hashgraph protocol, each participant contains the Hashgraph data structure, thanks to the parent hashes of a given event, a participant can estimate (virtual vote) the content of the data structure of the other participants, so the communication that is necessary for the voting process can be avoided. This feature allows reaching a transaction validation rate of about 20,000 transactions per second [76]. Smart contracts (Solidity language identical to Ethereum) can be deployed on Hashgraph. The disadvantage of Hashgraph is that it is only a partially open-source project, and Hashgraph can be used as a service provided by a company.

It can be summarized that today IOTA and Aleph Zero are still in an early phase in which they cannot provide deployment of smart contracts and this drawback does not allow to deploy real IoT-DLT applications. The deployment of smart contracts is enabled in Hashgraph, however, today Hashgraph is a service no real deployment is possible for the research community.

To conclude, this thesis focuses on blockchain, not on DAG and Hashgraph-DAG structures, because most DAG does not allow the smart contract deployment, and many of them are partially centralized.

2.3 On- and off-chain approaches to integrate IoT with Blockchains

It has already been described earlier (in section 2.1, p.13) that the domains of blockchain and IoT are different. Initially, most IoT devices were not designed to perform high computational tasks and store large amounts (hundreds of gigabytes) of data, which are the requirements of blockchain technology. It is worth noting that there are IoT devices with high computing power and adjustable memory, for example, the Raspberry Pi 3 B+ model[4] with the ARM Cortex-A53 processor or the Zynq-7000[5] with the ARM Cortex-A9 processor (these processors are also present in today's smartphones). These devices could contain a copy of the blockchain (until a certain amount of Giga Bytes) and compute most of the requirements of the blockchain. However, most IoT devices have limited hardware resources.

It is possible to distinguish two basic approaches: when the IoT contains the copy of the blockchain, and when it interacts with the blockchain without containing its copy.

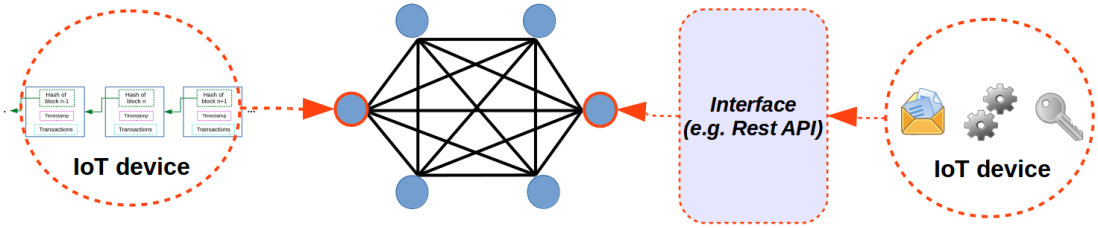


Figure 2.11: On-Off chain approach of IoT device integration with Blockchain

[4] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[5] https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

- *The "on-chain IoT approach"*

  On the left side of the figure 2.11, an IoT device can be seen that contains the copy of the blockchain; thus, the device takes part in the blockchain network as it is a peer of the blockchain network. This contribution called this the "on-chain" IoT approach, because the IoT device is part of the blockchain or is also "on the chain". In this case, the IoT device can benefit from all the advantages of the blockchain's features, however, the device must be synchronized with the other peers. This is a disadvantage of this approach, as the device needs to exchange the information with the other peers constantly, so the device's network connectivity must be maintained. Continuous connectivity costs too high energy consumption which could be critical in IoT applications. It should be noted that in some cases the device can remain offline, but it cannot send a new transaction until it has synchronized its database (which means it must download the missing blocks from its database).

  - ***Advantages:***

    The device contains the copy of the blockchain, so it fully participates in the blockchain network.

  - ***Disadvantages:***

    The size of the blockchain requires a huge memory for storing the copy of the blockchain. The device has to remain continuously connected to the blockchain network, which significantly increases its power consumption.

- *The "off-chain approach"*

  The right side of the figure 2.11 shows the "off-chain" IoT approach, in which the IoT device does not contain the copy of the blockchain and is therefore "off-chain." In the literature and developer documents, this approach is often referred to as offline transaction creation and signing. In this scenario, the IoT device can interact with the blockchain via an interface provided by the relevant blockchain. In this approach, constant connectivity of the device is not required, unlike in the "on-chain" approach. This solution is more optimal for IoT devices because the power consumption can be reduced due to the minimal communication required. However it is important to note that the IoT device must be able to use the given communication protocol and it must be able to compute a certain level of *cryptography* in order to identify itself to the blockchain. Moreover, the generation of the transactions must meet the requirements of the given blockchain. In the same figure, the key refers to the private key of the device, its public key must be registered on the blockchain. The digital signature process with the private/public key pair allows the identification of the device (see figure 2.4, p.17). This approach achieves IoT autonomy, but it requires more computing *complexity of the IoT hardware* or "*more sophisticated hardware*" [2].

  - ***Advantages:***

    The device does not have to remain continuously connected to the blockchain network, which is more optimal in terms of power consumption.

    The device can be authenticated in the blockchain network by using the digital signature process.

– ***Disadvantages:***

The device does not contain the copy of the blockchain, so it does not fully participate in the blockchain network.

This thesis also explores which cryptographic primitives are necessary to be able to establish an IoT interaction with a given blockchain, and which of these primitives could be addressed by *hardware* acceleration to speed up computation and achieve an optimal energy consumption.

2.3.1 On-chain approach - Related works in IoT domain

In this IoT-Blockchain approach[6], the IoT devices contain a copy of the blockchain or only the core of the database of the given blockchain.

The master thesis of N. Massiera [50], describes successful implementations of blockchain clients[7] on the Raspberry Pi 3 B+. The objective of this study was the analysis of implementation possibilities of Ethereum, Hyperledger Fabric, and R3 Corda blockchain node on IoT devices and in particular on Raspberry Pi 3 B+.

This work also demonstrated that this device can run a full client of Ethereum (Geth implementation). In particular, this Ethereum blockchain was deployed as a private blockchain, the author also mentioned that the mining process is infeasible in the Raspberry Pi (to solve the PoW problem, the CPU was overheating which physically broke the CPU). The infeasibility of mining on a Raspberry Pi was also demonstrated in [77]. The authors of paper [78] provide a description on how to deploy a full Ethereum node on Raspberry Pi 3 B model.

It should be noted that Ethereum provides two types of clients, full clients and light clients. The full client contains the entire database of the blockchain, it can initiate transactions and interactions with smart contracts, it can also participate in the consensus, thus participate in the mining. Unlike the full node, the light node does not contain the entire database of the blockchain, it must store only the block headers taking less space. The advantage of a lightweight node is that it can hold an account's wallet, making it easier to trade in cryptocurrencies.

According to the Ethereum documentation[8], this feature allows Ethereum to be used in more constrained devices such as smartphones for example. The disadvantage of a light node, is that it cannot initiate transactions or interactions with a smart contract itself, it has to request a full node. Light nodes play an important role in the Ethereum blockchain, these clients can easily verify the immutability of the database, verifying the hashes of the block headers form a Merkle tree (hash graph). After checking a certain depth of this hash tree, it is easy to determine if any of the blocks have been manipulated, allowing it to be reported to a full node.

According to [50] Hyperledger Fabric node (Docker implementation) can be run on a Raspberry Pi 3 B+. The comparison of Ethereum and Hyperledger Fabric nodes shows that the power consumption of the Hyperledger Fabric node is almost three times that of the Ethereum nodes. However, it should be noted that in this study, a private Ethereum

[6] In the blockchain literature, the "on-off chain" approach often refers to data storage. The data is stored on the chain or in another database that is not a blockchain.

[7] The literature also uses the term "node" as a synonym for "client.

[8] https://ethereum.org/en/developers/docs/nodes-and-clients/#light-node

blockchain was implemented (with 3 nodes) which is probably faster than a public Ethereum blockchain due to the number of participants which requires less communication between nodes.

The authors of [77] deployed an IoT-Blockchain solution in an electric vehicle battery refueling use case. They used a private Ethereum blockchain with special smart contracts that enable the management of a battery swapping system. One of the smart contracts allows storing pieces of information about the batteries (e.g., state of charge), these types of information are captured by an IoT device (Raspberry Pi 3 B+). Another smart contract was used to manage cryptocurrency transactions between accounts (battery swapping is a service that the car owner must therefore pay for). A third smart contract was also used to provide API interface for the station operator, the electric car owner, and the super account (admin). The authors used a blockchain to achieve a secure environment, traceability, and automation of the battery swap and the recharging of swapped batteries. In the experiments, the authors tested two scenarios, in which the refueling station and electronic vehicles are modeled by Raspberry Pis. In the first scenario, the refueling station and vehicles contain a full Ethereum node (Geth) without mining, a PC was equipped with a full node to mine. In the second scenario, only the PC contains a full node, the other entities are thin nodes, and they can interact with the blockchain and smart contracts by calling the full node with RPC calls. They found that using the full node increases the CPU usage almost 5 times and the memory usage of the Raspberry three times. The use of thin nodes in Raspberry Pis leads us to a more optimal use of the hardware.

In a smart home system [79], the authors also used the ”on-chain IoT approach”, this system uses a Raspberry Pi (called Sensor Manager) as an IoT gateway to collect environmental data provided by sensors. The data is used to determine emergencies (e.g., accidental injuries, crimes, etc.), if an emergency occurs, the Sensor Manager reports it to a dedicated smart contract that can log the emergency call. The smart contract informs the dApps that can call the Homes Service Provider (HSP), and distributes a one-time password QR code to them that allows the HSP to enter the homeowner’s home. In this system, the Raspberry contained a full Ethereum Geth node, and it also served as an IoT gateway.

Dalmasso *et al.* [80] propose a lightweight alternative to blockchain that can be implemented in IoT devices (”on-chain IoT approach”), this alternative takes into account memory, computational power and energy consumption constraints. The proposed protocol is called Wallance, and it is based on a monetary valuation. When a node shares its data, it earns DCoin, which is required to gain access to resources and/or services. Wallance uses a lattice wallet structure, so it does not store the full transaction history, unlike blockchains. A node has its wallet which contains the node ID, the DCoin, and the state which is an unique value, it is like a hash of a block in the blockchain, and therefore a new state depends on the previous state. Each node contains all the wallets in the network, however, a node is not required to store the full history of wallets, it only needs to store the latest version. Wallance provides integrity and no history. The transaction of a node that wants to access a service or resource must be validated according to the Wallance consensus. The Wallance protocol includes the following steps. First, the sending node must determine a nonce value, which is the result of a lightweight Proof-of-Work (the SHA-256 hash algorithm was used). The second phase is the voting process. The majority of the nodes, so 2/3 (to reach the BFT) must agree on the validity of the transaction (e.g., if the sending node does not have enough DCoin, the transaction will be rejected) by voting. Voting nodes are rewarded after this

process. This Proof-of-Concept (PoC) can be implemented on Raspberry Pis, and after estimations, the consensus could be reached within 20s when a network contains 10,000 nodes (Bitcoin needs 10mins to reach consensus). This system uses the characteristics of DLT, but it does not store the data, so it is questionable whether Wallance can be called a blockchain. But this approach can be used in some IoT applications where integrity and high transaction rate are the only requirements. For a more detailed version of this work, the reader can also consult the PhD thesis of L. Dalmasso [46].

In the related works above, the Etehereum Geth Go programming language implementation was used for Ethereum network deployment (this is the original implementation), in [19] (part of this thesis contribution), Ethereum Aleth, a C++ implementation of the Ethereum client, was analyzed to determine which cryptographic functions are most commonly used in the Ethereum client. The choice of Aleth is due to the fact that the C++ programming language is still the language that makes more optimal use of hardware resources than other programming languages (except C language). It could be expected that the Aleth implementation can run more optimally in more robust IoT architectures as a Raspberry Pi for example.

It can be noticed that blockchains are implemented using different programming languages, for example Go, Python, Java, Rust and C++. The choice of programming languages to be used was not significantly considered when deploying the blockchains, because blockchain technology was designed to be used in powerful hardware architectures such as PCs.

Computing the cryptographic functions required for blockchain technology may require high computing power or take a big amount of time and energy to execute. In IoT devices, execution time and energy consumption are still a challenge to solve. The objective of the analysis of one of our contributions [19] is to identify the most demanded cryptographic functions and their importance when using the Ethereum full client. This contribution can provide the idea for a new sophisticated model of IoT hardware architecture, which can be designed in SystemC-TLM for example. This contribution can also help to find a more optimal software implementation of cryptographic algorithms, which are dedicated to IoT architectures. The analysis was performed with the call graph execution profiler gprof [81]. This tool returns the call tree of functions (parent-child representation) that are used in the program, it also returns the percentage of the total time the program spent in the given function and the number of times it was called.

The measured program initializes a node (Client) that joins the Ethereum network containing three other nodes. After this peering procedure, the node must synchronize, so download the blocks that are present in the blockchain (at least 2 blocks, a genesis block, and a block because a smart contract has been deployed). After the synchronization, the client creates a simple transaction, which allows to increment a value in a smart contract. Finally, the program ends. The table 2.1 shows the results of gprof, thus the most called cryptographic functions.

The SHA-256 hash function is called when initializing the node and pairing with other nodes in the network. The keccakf1600 function with special parameters to obtain a 256-bit hash value (keccak-256) is called when a transaction is created. The pseudo-random function salsa20_8 (8 refers to the 8 rounds) is used to mix the function input pseudo-randomly. This function is called after the transaction is sent.

%	cumulative	self		
Total time	**seconds**	**seconds**	**n° calls**	**cryptographic function**
49.33	1.84	1.84	7339905	SHA-256
17.43	2.49	0.65	1074092	keccakf1600 (keccak-256)
13.40	2.99	0.50	8387735	salsa20_8

Table 2.1: Results by *gprof*, results are retrieved from our contribution [19]

It can be concluded that the on-chain IoT approach is not necessarily the most optimal solution due to drawbacks such as the large amount of data to be stored (and it grows infinitely), and it was also pointed out above that the node needs to stay synchronized with the network, so near-constant Internet connectivity is required. In applications where mobility of IoT devices (or dynamic devices) is a necessity (e.g., vehicular networks, smartphone applications, wearable devices, healthcare), this approach could not be used properly because connectivity cannot be provided, and even if it is, continuous communication leads to higher energy consumption. Most IoT devices are powered by batteries, and the continuous communication would quickly reduce battery life.

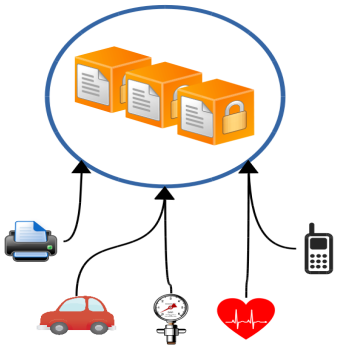
The other issue is the requirements of full and lightweight nodes, these clients require an operating system, so the IoT hardware should logically have more computing resources. However, in some IoT applications, where devices can be continuously powered and may have external data storage capabilities (e.g., smart homes, power grid, electric vehicle charging stations etc.), this approach can be a good solution and probably more secure than the ”off-chain IoT solution”. In IoT gateways this approach could also be an optimal solution.

2.3.2 Off-chain approach - Related works in IoT domain

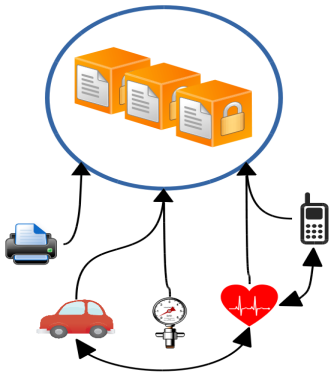
Before exploring the related works of this approach, it is worth noting that according to the literature there are 4 basic schemes/system architectures in which IoT devices can be integrated into a blockchain, these schemes are shown in figure 2.12.

In figure 2.12a, IoT devices send their transactions directly to the blockchain, so all interactions with the blockchain are recorded in the blockchain database. It is also possible that IoT devices exchange information with each other (see figure 2.12b), and that these IoT-IoT interactions or the logs of these interactions are not necessarily published on the blockchain. In some cases, for example in wireless sensor network applications, the use of a gateway is necessary, this scenario is illustrated in Figure 2.12c. Identification of IoT devices is still possible in a secure mode because the gateway can be seen as a bridge to the blockchain; it will not modify the IoT transaction. Or even if a malicious gateway alters the IoT transaction, it can be easily recognized through digital signatures. The last scheme is called the ”hybrid” cloud/blockchain structure 2.12d, in which some IoT data can be sent to a cloud, which can be done faster than to the blockchain, due to the block addition latency blockchain is slower than a cloud, and other types of data can be sent to the blockchain directly or via a gateway. This scheme can also ensure data integrity, the cloud or another type of Distributed Ledger Technology (e.g., IPFS[9], Swarm[10] can contain the raw IoT data, and the hash of that raw data can be recorded by the blockchain. When the raw data is accessed in the cloud, its

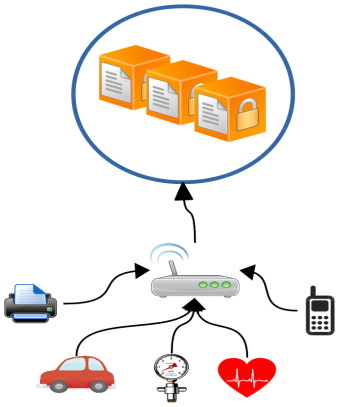
[9]IPFS: InterPlanetary File System - https://ipfs.io/
[10]Swarm: - https://swarm.ethereum.org/



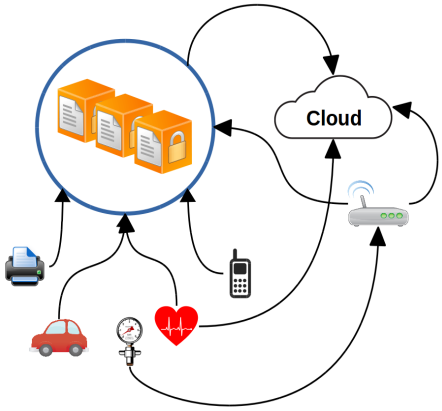
(a) IoT devices interacting with the blockchain directly



(b) IoT devices interacting with each other and with the blockchain directly



(c) IoT devices communicate with the blockchain via a gateway



(d) Complex "hybrid" network architecture containing BC, IoT, gateway and cloud

Figure 2.12: The 4 basic IoT-Blockchain schemes/system architectures, the figures are adapted from [1] and [2]

hash can be calculated and compared to the hash stored in the blockchain. If the hashes do not match, the raw data has been modified in the cloud.

These schemes also show that the IoT device must include some communication protocol. If the communication is direct between the IoT and the blockchain, 4G and 5G could be used for example, if the IoT is a more constrained device, it would probably use LoRaWAN, Bluetooth or Wi-Fi, in that case a gateway would forward the transaction to the blockchain.

In related works, there are a significant number of applications and use cases in which authors have integrated IoT with blockchain. This part of the subsection describes some of these applications and their implementations.

Oscar Novo aimed to realize a distributed access control management for IoT devices [3], which is driven by an Ethereum blockchain. Access management is needed when one IoT device wants to access another's resource. The author proposes an architecture in which sensor networks are geographically distributed and access control is decentralized. In this system, all IoT devices are uniquely identified in the blockchain network, thanks to their public keys that are stored in the blockchain. In this architecture (see figure 2.13), there are managers, which manage the access control of a set of IoT devices. The manager is a

device that can interact with the blockchain, but it does not contain the copy of the blockchain. System access management is implemented in a smart contract, managers can only access the smart contract when defining new policies. The communication between an IoT device and the blockchain (one of the nodes of the blockchain) is provided by an innovative solution, which the author calls a management hub. This is an interface that can translate CoAP-encoded messages from IoT devices (this encoding is used in most constrained IoT devices) into a JSON-RPC message that is a remote procedure call encoded in JSON. The interface calls a miner node on an RPC port with the translated message. This interface is written in JavaScript using the web3 JavaScript API allowing communication with the Ethereum blockchain. The RPC protocol is used in many blockchains to be able to interact with a given blockchain node directly, another popular protocol is REST.

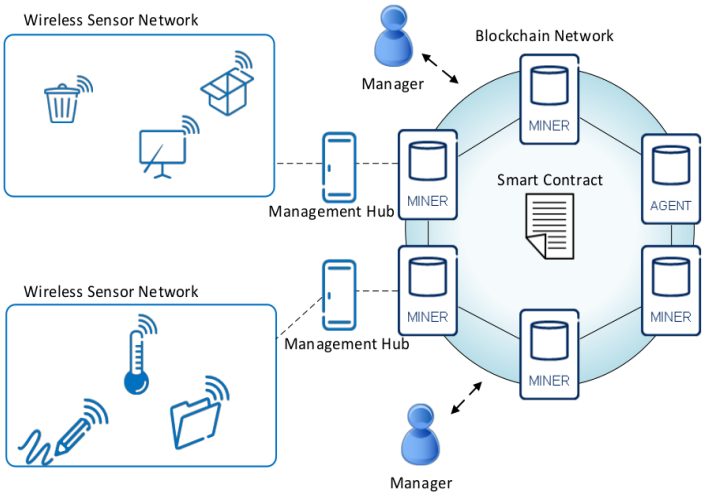


Figure 2.13: The blockchain-IoT network proposed in [3]. The figure is retrieved from [3]

It should be noted that this interface (management hub) is deployed next to a miner node (on the same hardware) in a private blockchain (in future work, this interface can be a standalone entity that can interact with the miners of the blockchain), so this system is a perfect example of a Proof-of-Concept (PoC), which means that the system is decentralized, but the user is required to trust the system provider. Another feature that leads this architecture to PoC is the use of an agent node that can deploy the smart contract for access control and so acts like an admin.

The realized architecture is similar to the one described in figure 2.12, and it can perform the following scenarios, for examples. An IoT device (heart monitor) can send a request to another device (smartphone) to get access to its resource (using CoAP protocol), before the smartphone gives its resources, it would inform the management hub which would request the access policy from the smart contract, that responds the policy to the smartphone through the management hub. If the response is positive, the smartphone can provide access to the heart monitor, to its resources.

In this architecture, manager nodes are used to register IoT devices and modify management policies, while agent nodes are used to deploy smart contracts. This agent can be considered as an admin entity, this entity with the functionality to register new IoT devices on the system is added to the implementations of this thesis contribution. The identification of the public key of all devices is also inherited in the implementations of this thesis work.

Hammi *et al.* [4] deployed a decentralized authentication system for IoT devices, the system is based on an Ethereum public blockchain. The authors call this decentralized system bubbles of trust, in which the bubbles refer to virtual zones in which trust and identification among the devices taking part in the given zone are provided (see figure 2.14). This system can be used in ecosystems using IoT to ensure ecosystem sustainability and resiliency. According to the authors, one of the requirements of any IoT application is the identification of all devices and mutual authentication, which means that each communicating party is known to the other.

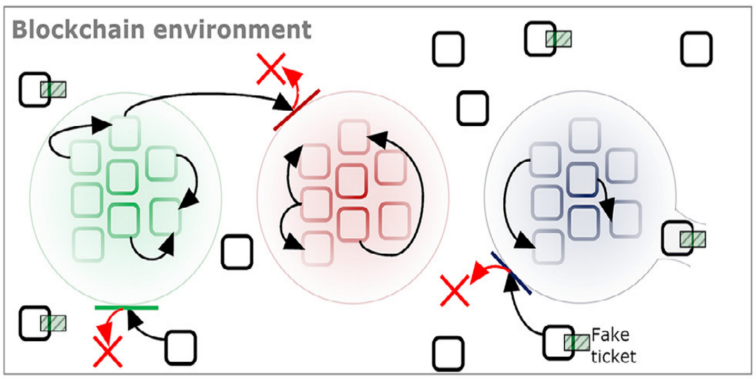


Figure 2.14: Bubble of trust in blockchain environment. The figure is retrieved from [4]

The goal of this research is to make an ecosystem using IoT devices more secure and to find a possible solution to the IoT paradigm by ensuring integrity, availability, scalability, non-repudiation, identification and authentication. In this system, there are two basic entities: masters and followers. The master is responsible for creating a bubble with a dedicated identity for the followers, and it also registers the followers and the identity of the bubble on the blockchain via a smart contract. The master also creates tickets that are distributed among the followers, which are participating in the same bubble. The ticket contains the identity of the given follower and the bubble. This ticket is signed (digitally) in the IoT device (master), by using the master's private key. When a follower wants to send a message to another follower, first it has to send the ticket to the smart contract. More precisely the follower sends the ticket and the corresponding signature of the ticket. First, the smart contract verifies whether the ticket has been signed by the master and it also checks whether the identities of the bubbles and followers contained in the ticket have already been stored in the blockchain. If the signature and the content of the ticket are correct, the follower can send data that can be exchanged with a specified follower of the same bubble. This exchanged data is stored in the smart contract and the targeted follower can read it. In this system, a follower cannot send or read information from other followers if it does not have the ticket that has been signed by the master.

In this implementation, the authors have developed a C++ interface that used for the encoding/decoding of data that needs to be sent to the Ethereum blockchain from the end device. RPC communication protocol and JSON format are used to ensure communication with the blockchain. The developed program uses User Interfaces (UIs) that are programmed with Qt C++ library. It should be noted that only the tickets are signed locally on the end

device, the transactions containing the given tickets or data to be exchanged are signed by a node of the Ethereum blockchain.

While the proposed implementation performs the essential encoding of the transaction in order to call the smart contract functions, this transaction can be viewed as a request to an Ethereum node to sign and broadcast the ”node signed” transaction.

It is worth noting that the verification of transactions is done by the blockchain network itself, without using a smart contract. In this work, it should not be confused that the transactions are signed by the blockchain node, so the transaction signature would be verified by the blockchain. However, the tickets contained in the transactions are signed by the end device locally (signed by the master's private key), so the ticket signatures are verified in the smart contract. This feature increases the security of followers identification.

The ticket signing and its verification in a smart contract are useful features, that can be improved and involved in other scenarios as well. For example, the device user's private key could be used to sign the transaction and the payload of the transaction could be signed by the device's private key (or inverse). This feature allows verifying if the device owner accepted to send the transaction from its device. The feature of signing the message that would be contained by the transaction, and afterward signing the transaction in the usual way are also used in this thesis contribution. More technical details are given on page 61.

In the proposed implementations of this thesis contribution, the transactions created by the IoT device (or end nodes) are signed locally in the device, this method is also called offline transaction signing. The signed transaction is then sent to a node of the blockchain that will broadcast this transaction in the network. Signing transactions locally in the IoT device can provide more secure identification of IoT members of the system. To accomplish this local signature when using Ethereum blockchain additional encoding and formatting is required (more than in the work described above). For more technical details on how the transaction should be formatted, encoded and signed (offline) the reader is invited to read the following chapter.

In [5] [82], the authors work on an implementation in which the IoT device functions as a direct data source actor in an Ethereum public blockchain infrastructure. The IoT devices in this system are used to measure water consumption, and the blockchain serves as a water management system. The direct interaction with the blockchain and smart contracts is provided by the digital signature of transactions that is given in the IoT device. In the same work, the authors implemented a C language tool that can be used in constrained devices like Arduino Uno, STM32L031K6T6, STM32L452 (see figure 2.15). The authors also measured the execution time and power consumption of these constrained devices, and they also described the different phases of program execution when creating and sending a transaction from the IoT to the Blockchain. In the next chapter on page 61 this work is compared with more details with the implementation that we developed in this thesis contribution.

In this thesis contribution, a C++ implementation is provided in order to create valid Ethereum transactions. The main difference between the proposed and the implementation above is that our contribution is open-source to help the IoT-blockchain community and the it also gives a possibility to create double signatures sign the payload of the transaction locally in the IoT as it was proposed in [4] and also sign the transaction locally in the IoT device. On the other hand, the developed tool is used in a different use case, used in a vehicle infras-

Device	Model	MCU	Architecture	Clock (MHz)	Prog. Mem (KB)	SRAM (KB)	Price (USD)
ATM	Arduino Uno	ATMega328P	8-bit AVR	16	32	2	18
M0+	STM32L031K6T6	Cortex M0+	32-bit ARM	32	32	8	10
M0	STM32F030R8T6	Cortex M0	32-bit ARM	48	64	8	10
PIC	ChipKit Lenny	PIC32MX270	32-bit MIPS32	40	256	64	23
M4L	STM32L452	Cortex M4	32-bit ARM	84	512	96	12
M4F	STM32F401RET6	Cortex M4	32-bit ARM	84	512	96	12

Figure 2.15: Studied devices and their characteristics. The figure is retrieved from [5]

tructure combined with blockchain.

The blockchain-based IoT systems can also be used in agriculture and food supply chain management [6], to provide traceability and transparency of every step, from agricultural production to consumption (from-farm-to-fork use case). In this use case, the future consumer can verify the whole history of the product before buying it. The history of the product (see figure 2.16) includes the raw materials purchasing (technical details about the products), planting (procedure of plants' planting), growing, farming (e.g., information about watering), harvesting, delivery to processor (GPS information can be used), processing (e.g., packaging), retailing (e.g., information about retail's environment) and finally the consuming when the consumer can verify every phase of the product that he/she would like to buy. The information of these phases in order to record the product history is provided by IoT sensor measurements these information are sent to specific smart contracts that can store the data and determine anomalies that would play an effect while the product would be sold to the client.

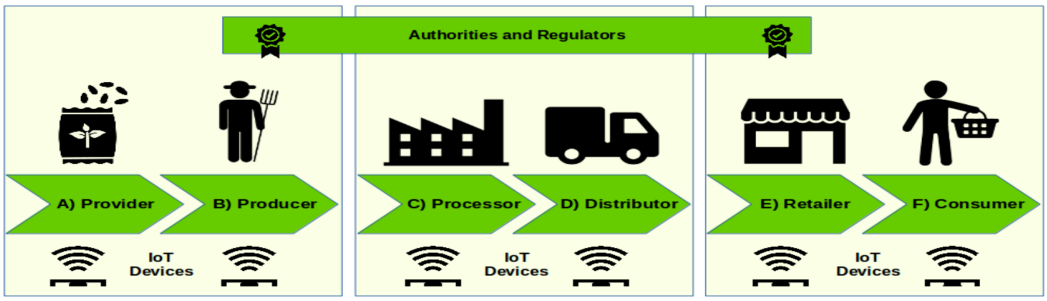


Figure 2.16: Simplified version of the Agri-Food supply chain management process. The figure is retrieved from [6]

In this work authors used Ethereum and Hyperledger Sawtooth blockchains. The authors highlighted that Hyperledger Sawtooth (see section 3.4, p.50) is a modular private blockchain that allows the deployment of smart contracts in different programming languages like Python, C++, for example. They also highlighted that the transaction validation rate of Hyperledger Sawtooth can be higher than Ethereum's. Transaction validation rate is an important factor in IoT-Blockchain systems, due to the large amount of IoT devices that can send information with high frequency. In this work the business logic that is described by smart contracts is not detailed. For an example of concrete business logic describing a supply chain management with Service Level Agreements (SLA) to ensure a secure and transparent delivery service of sensitive products like medicines, the reader is invited to read this article [83].

The latter characteristics and the fact that this thesis work looks for an ideal blockchain

to be used in an ecosystem, it encourages interest in studying the Hyperledger Sawtooth blockchain and the integration of IoT with this blockchain.

L. Gerrits' Master thesis [59] describes the utility of decentralized applications (dApps) for the EOS blockchain, which can be perfectly applied in ecosystem use cases. In this work, the author deployed a JavaScript WebApp as dApp[11] that allows the creation and signing of EOS transactions, and also provides Web user interfaces (UIs) to facilitate interaction with smart contracts. In this application, the smart contract is combined with the Ricardian contract, one of the UIs provides the Ricardian contract presentation to show the results of the smart contract according to inputs. These inputs are defined by the end-user through the dApp (the end-user defines the inputs through a User Interface). The author has also deployed a C++ Client tool[12] to interface with the EOS.IO blockchain. This tool allows creating and signing transactions offline, in an IoT device.

Since EOS can be used in ecosystems, and it also allows the deployment of Ricardian contracts and there is a C++ tool to interface with this blockchain, this thesis work will also explore the possibilities of IoT integration with this blockchain.

The related works described above give a few ideas about some of the existing use cases, the basic requirements, ideas, and the need for future improvements that allows an efficient IoT integration with blockchain technologies. As previously mentioned, one of the fundamental goals of this thesis work is to use IoT devices in vehicles that are connected to a blockchain-based ecosystem. In the following subsection, the reader can learn more about the related works on blockchain-based vehicle infrastructure.

2.3.3 Off-chain approach - Related works in automotive domain

This section describes related work on blockchain-based automotive infrastructure, which includes useful ideas and implementations. The desire of using blockchain technology shows a high interest in the automotive sector. To give a few examples of car manufacturers, Renault and BMW want to create a digital car pass that can contain for example car maintenance and millage information, this information would be accessible for potential car buyers [84] [85]. BMW also mentions that blockchain has the potential to be applied to future self-charging processes of autonomous cars [85]. Porsche has already tested blockchain applications that can lock and unlock the vehicle [86].

Michelin *et al.* [7] developed a Speedy Chain blockchain framework that can be used in smart city scenarios and Intelligent Transportation Systems (see figure 2.17). In this working proposal, roadside units (RSUs), e.g., traffic lights, contain the copy of the blockchain, while the vehicles that send their sensor data to the RSUs contain only the Merkle tree of the blockchain header hashes. The data collected by the RSUs can be used to help in case of traffic incidents for example. It should be noted that in this work, the authors, unlike using a standard blockchain (e.g., Ethereum), have developed their own blockchain framework. It works as follows: when a new car is added to the network, a new block is added according to that car. When a car sends a transaction, it is stored in the block that was created for that car.

[11] This work is available at https://github.com/lucgerrits/EOS.IO-sensor-dapp
[12] This work is available at https://github.com/lucgerrits/EOS.IO-cpp-client

This feature slows down the rapid increase in the number of blocks. Vehicles in the network are identified by their public key and transactions are signed by their private key.

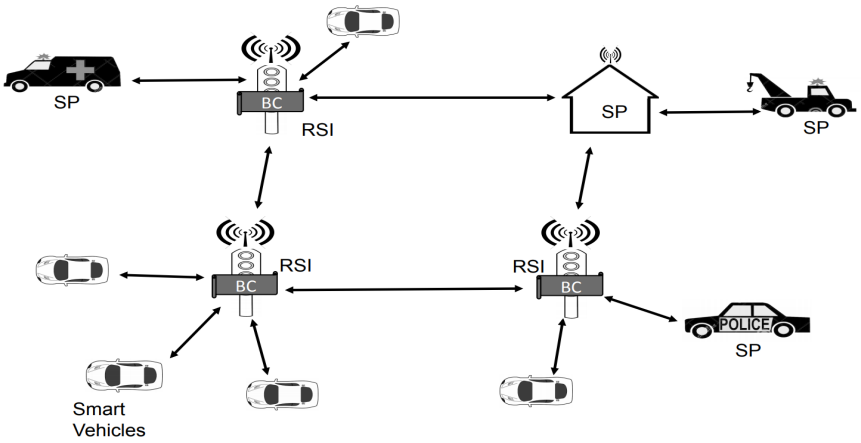


Figure 2.17: SpeedyChain in a Smart City scenario. The figure is retrieved from [7]

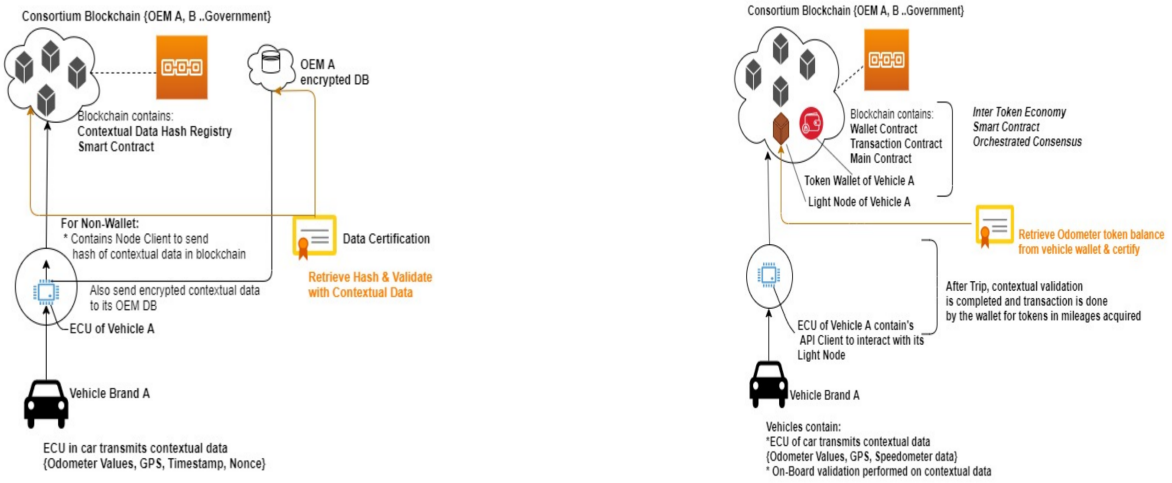
To achieve a higher level of security, private/public key pairs can be changed at any time, reducing the risk of an attack that could unveil the private key of a given car. The results of experiments show a promising transaction validation latency of 1.69 ms for 650 vehicles sending 1000 transactions, this result is significantly less than in the case of Ethereum public blockchain with 7 transaction/s for example.

Likewise, in the work described above, the authors of [87] have implemented their own blockchain for vehicle networks. In this system, vehicles can send information to each other to prevent each other from traffic, accidents, and outher events on the road. Vehicles evaluate the information sent by others (+1/-1 valid/invalid), this calculation is based on a complex probabilistic equation. The vehicles can report their ratings to the RSUs containing the blockchain. Each RSU calculates the trust value offset based on the ratings given by the vehicles. Cars can request this trust value to be able to filter an eventual malicious vehicle. This blockchain uses a joint PoW and PoS consensus algorithm, in which RSUs computes the PoW, which is easier to achieve if the trust value offset variation is large. This feature allows a faster block adding if there is a large variation among the trust values. The sum of absolute offsets is equivalent to the stake in the PoS. This "own" blockchain implementation can also provide a slight latency in transaction validation.

These two works described above do not use a standard blockchain, which can also be seen as a drawback, as there are no technical and research contributions on their implementations.

Samuel *et al.* used a consortium Ethereum blockchain to solve the odometer fraud problem in the vehicle industry [8]. The authors propose two implementations on the Ethereum blockchain, a wallet and a non-wallet-based system. In both solutions, a certification of the data is recorded on the blockchain, to guarantee the value of the odometer. The choice of the consortium blockchain is because according to the authors the inclusion of members is easier and the ecosystem is easier to achieve. In this ecosystem government, agencies and

car-sellers could participate.



(a) Non-Wallet use case

(b) Wallet use case

Figure 2.18: Wallet and Non-Wallet use case, the figures are adapted from [8]

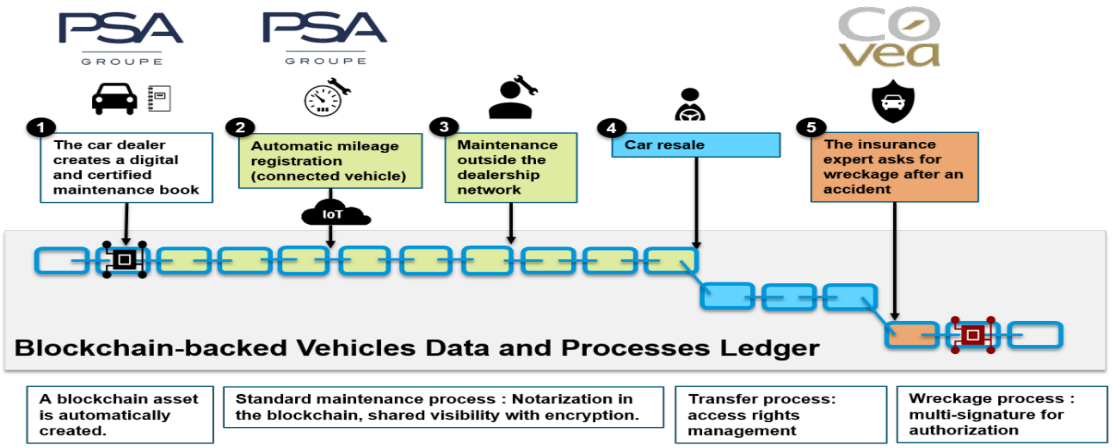
In the non-wallet architecture 2.18a, the vehicle contains a private-public key pair, the vehicle encrypts its identity, odometer data, and GPS coordinates, this encoded data is sent to the manufacturer's local database (contextual data). The car computes the hash of this data and its result is sent to the blockchain. When an ecosystem member needs access to the data it would first request the hash of the data stored in the smart contract, the hash can prove that the data was not modified in local storage. To compare the hash stored in the blockchain with the hash of the contextual data, a certification component is used. This entity, therefore, does not take place in the blockchain. This certification component uses GPS information in order to verify the correctness of the mileage information reported by the car.

In the Wallet solution 2.18b, each vehicle owns a Wallet in which the Tokens represent the mileage information (the way that the vehicle has done in km). Ethereum light nodes have been added to facilitate the deployment of Wallets and to facilitate communication between the car and the smart contracts. When the vehicle completes its trip, the vehicle's ECU (Electronic Control Unit) certifies the number of kilometers traveled, after a request would be sent with the certified mileage information to the smart contract. The Proof-of-Authority is used as a consensus of the blockchain, a second consensus is also applied to manage the Token economy. If this second consensus is reached, the vehicle's wallet will be enriched according to the miles the vehicle has been driven. These architectures are PoC-based, using admin nodes to achieve a more secure process for adding new ecosystem members.

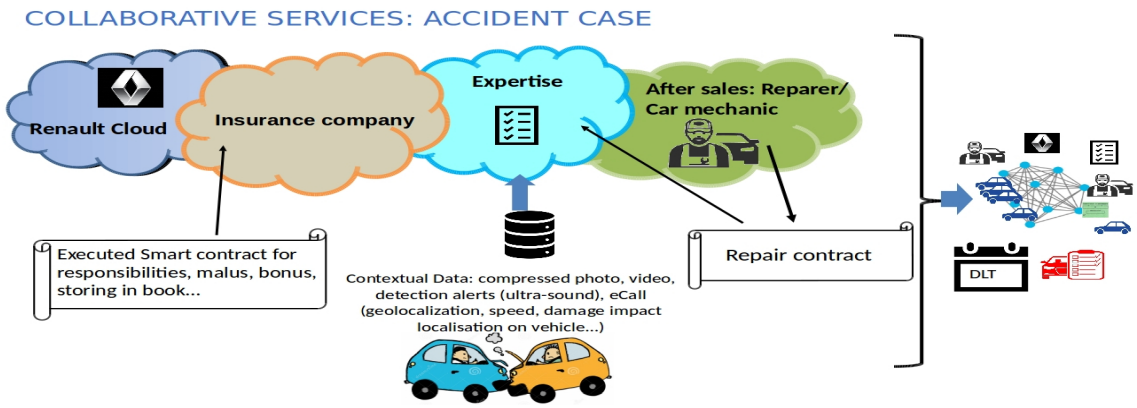
The authors conclude that this implementation can be applied to a fleet of 11 million vehicles if the vehicles make an average of 4 trips per day. In the experiments, IoT devices were simulated and the client that communicates with the Ethereum light node is developed in JavaScript programming languages. This client tool could be converted into other programming languages to be more optimal for constrained devices.

Likewise the work of Samuel *et al.* [8], the work proposed in [88] also encourages the use of consortium blockchain for the implementations of vehicular ecosystems. In this work, blockchain and decentralized technology are used to manage the vehicle life-cycle and to encourage collaboration of the actors involved in vehicle infrastructure. The authors propose

to apply digitized and certified service records and automatic mileage reporting to prevent vehicle-related fraud. This ecosystem architecture includes several actors, such as a car manufacturer, an insurance company (see figure 2.19a).



(a) Ecosystem proposed in [88]. The figure is retrieved from [88]



(b) Ecosystem used in SIM project

Figure 2.19: Similar ecosystems in vehicular use cases

This ecosystem has similarities with the architecture that is described by this thesis work being part of the Smart IoT for Mobility (SIM) project (see SIM project's ecosystem in figure 2.19b). The proposed architecture contains a Quorum-based blockchain that is derived from Ethereum. This blockchain is used to store information about vehicles' maintenance and to manage the access of nodes. According to the authors, every member of the ecosystem needs a Quorum node to be able to communicate, share information and manage services among them. The car manufacturer provides data access management for car owners, car dealers, and car repair shops. Certain types of vehicle data are stored locally in the automaker's servers and the pointer to that data is stored on the blockchain. Vehicles cannot communicate directly with the blockchain, the mileage information is reported to the automaker's cloud which contains a service that forwards this information to the blockchain. Because vehicles cannot communicate directly with the blockchain, car owners must trust that mileage information will not be altered in the manufacturer's cloud service. This architecture is a PoC, and the cloud service mentioned before can be seen as a drawback of the system because it

can be considered as a trusted third party.

In Smart IoT for Mobility, the vehicles' direct connection to the blockchain and their identification is an essential characteristic to be realized in order to achieve a decentralized vehicle infrastructure.

This system proposed in [88] also includes an insurance service, which can provide insurance smart contracts for car owners. After suffering an accident an insurance expert can have access to the car service book. The insurance access management and refound procedures by smart contract would also be studied in SIM project. The study also proposes to store raw data (e.g., invoices) in servers and their hashes on the blockchain. In SIM project there are implementations in which the data recorded by vehicle's sensors (e.g., 360° cameras, odometer, accelerometer) and sent to IPFS a (distributed ledger), the hash of the data is stored in the blockchain, and it serves as a reference to the data stored in IPFS. Storing raw data in a distributed ledger keeps the data more decentralized than storing it locally. It should be noted that in some cases an ecosystem member does not necessarily want to share a piece of sensible data, in that case, the data effectively can be stored locally and the hash of the data in the blockchain.

The authors also introduced a smart contract that helps two participants exchange encrypted information using synchronous (AES) and asynchronous (RSA) cryptographic models. The smart contract stores the encrypted data and the encrypted session key for AES, the encryptions are performed by the applications of the members participating in the data exchange. This idea could enrich the SIM project, however, this implementation is not deployed in this thesis work.

Since the SIM project and also this thesis work deal with the use case of vehicle accidents, it is relevant to describe the following work. In [89], the authors propose a framework for forensics applications of connected vehicles called Block4Forensic. The authors imagined an ecosystem and accident use case quite similar to the SIM project's ecosystem, and to Renault's use case. The proposed framework can be applied in an ecosystem including a car manufacturer, an insurance company, law enforcement (police), maintenance service providers, vehicles, and roadside units. The main goal of this work is similar to the accident use case of Renault, the vehicles On Board Units (OBUs) can collect data from the car's sensors (e.g., brakes, lidar for measuring breaking/security distance). This unit can also retrieve environmental data from road side units (e.g., traffic lights), the idea is to record as much data as possible before the incident/accident occurs. These data can help an insurance expert to determine more accurately and faster the faulty part of the given accident. The OBU contains a "Forensic Daemon" that is an application, that retrieves all of the information about the car and its environment, and it also automatizes the information sending procedure. The hash of the information is sent to the authorized blockchain using the PBFT consensus rule, the information is sent to the member who has an interest in the information. For example, data that is of interest to the insurance company may not be of interest to the car manufacturer. The data can also be sent to the local server of the car owner, to keep a backup. However, if one-day, member A needs data that is contained by member B, they can exchange it. Storing only the hashes of the data in the blockchain is an optimal solution to avoid the rapid increase of data that the blockchain needs to store, this technique is also used in one of the implementations of this thesis work (see page 76), however, this implementation encourages storing the raw data in the IPFS distributed ledger and not in a traditional way (servers or

clouds).

In forensic applications, the size of the data recorded by the vehicle can increase rapidly, as the vehicle contains many sensors (lidar, radar, accelerometer) and may also contain cameras to record the vehicle environment and driver behaviors. AIG a Singaporean insurance company provides a service in which the driver's driving behavior is recorded by an in-camera [90]. This service rates the driver's driving skills, who can obtain a premium (15% discount) if he/she drives according to the traffic rules. Authors of [91] determined that the size of the data can be near 1GB when a car is equipped with these devices and records only the last 10s of an accident. In a vehicle network that includes thousands of cars, this amount of data will not be possible to store in the blockchain.

2.4 Conclusion

This chapter gave extensive background about the blockchain and DLT technology, especially about existing IoT-blockchain structures and their applications. It can be noted that using blockchain with dedicated smart contracts can provide data integrity, distribution, and management for IoT applications. One of the main objectives of this chapter was to study the integration possibilities of IoT with blockchain.

The reader has been introduced to on- and off-chain approaches of IoT integration with blockchain technology. Blockchain-IoT use cases and vehicle infrastructure applications were also mentioned and studied on order to help in this thesis contribution.

It can be concluded that the on-chain approach (see page 29) which implements the blockchain on the IoT devices does not make sense because of the hardware limitations of the IoT devices. First of all, IoT devices need enough computational power to be able to run an OS (most of the blockchain client application's requirements). Secondly, the blockchain can take hundreds of gigabytes, which means that the IoT device must be equipped with large memory.

Using this approach only makes sense for IoT devices, which are less constrained, and in scenarios where the devices can be powered continuously, and the Internet connection is nearly continuous. This approach can be ideally used for example in IoT gateways to ensure a high-security level of the application. The reader also viewed an example of how blockchain can be formed for IoTs [80] by allowing devices to not hold the entire copy of the blockchain, but to ensure data integrity. This method is still in its early stages and it is questionable whether it can still be called a blockchain or whether it is rather a distributed ledger with specific characteristics and applicability.

In contrary to the on-chain approach, in the off-chain approach the IoT does not contain the copy of the blockchain, however, it must contain the essentials in order to interface with the blockchain. The IoT must be able to create a transaction with the encoding format (e.g., RLP encoding of Ethereum, see figure 2.5, p.16) required by the given blockchain, to be able to interact with a smart contract. It was also discussed that IoT devices must be identified by the blockchain, in order to be able to verify the provenance of the data and thus ensure the trustworthiness of the blockchain data. To identify IoT devices to the blockchain, the IoT device must be able to perform digital signature cryptographic primitives, the principle of

digital signature can be observed in figure 2.4 (p.17).

The literature often calls ”client, SDK or client API”, the tool that allows creating valid transactions. This thesis work encourages the development of C or C++ ”SDKs, client API” for IoT devices because these programming languages work more optimally on constrained/embedded hardware architectures. It must be noted that there exist only a few blockchains that provide official C or C++ client APIs. In the majority of blockchains, script programming languages such as JavaScript and Python are used to implement official SDKs. This can be considered normal because blockchain was invented to be used on PCs, the use of script languages is more comfortable for developers and the hardware usage of these languages is negligible in the case of PCs. The communication protocol can play an important role in the IoT device’s energy consumption, execution time, hardware (Bluetooth, Wi-Fi, LoRaWAN), and also the memory footprint requirements of the communication protocols.

The choice of the most optimal communication protocols is not within the scope of this thesis work, in the experiments and results, the time needed for communication is not measured or if it is, the Ethernet protocol has been used.

Chapter 3

Selected Blockchains to use in the off-chain structure

3.1 Introduction

This chapter is divided into three parts. The first part of the chapter highlights the basic requirements that a blockchain has to meet in order to be ideal for IoT applications. This part also describes the characteristics of the four blockchains which were selected to be used in this thesis contribution.

The second part of the chapter illustrates the blockchain APIs developed and found to enrich this contribution. The APIs were developed in C++ language, and they are contributed as open-source. All of the APIs meet the requirements of the given blockchain, which enables the creation of valid blockchain transactions. This part of the chapter also focuses on analyzing the given APIs to determine the most called functions. This part also highlights that cryptography plays an essential role in all of the APIs and that several cryptographic functions could be hardware accelerated.

The last part of the chapter provides a blockchain-IoT hybrid network structure, which is intended to resolve the quickly growing data size of the blockchain and control the member authentication. The smart contract and the blockchain handle the members' authentication and data storage on the ecosystem.

3.2 Requirements of an ideal Blockchain for IoT APIs

An ideal blockchain should provide a certain interface on the blockchain nodes that allows the client API of the IoT device to interact with one of the blockchain's nodes. This interface is often written in JSON-RPC (e.g., the related work [3] used this interface with Ethereum) or in REST API (e.g., Hyperledger Sawtooth).

This ideal blockchain should also provide the possibility of smart contract deployment. Ethereum blockchain was the first-ever blockchain really allowing smart contract deployment. The Solidity programming language has been invented for Ethereum's smart contract writing and Ethereum Virtual Machines (EVMs) are used to run the compiled contracts. Today, Solidity has become the most used programming language for smart contract deployment, thus if another blockchain also contains an implementation of an EVM, the Solidity language can also be used. Therefore, when a blockchain contains an EVM it can be considered an advantage.

Another interesting but less important requirement is the modularity of the blockchain,

which means that certain modules of the blockchain can be changed or modified, for example, changing or modifying the module that serves for running the consensus rule can be an advantage. By changing the consensus algorithm a better transaction validation rate can be achieved (this feature is implemented in Hyperledger Sawtooth and Substrate blockchain frameworks).

Blockchain smart contracts cannot themselves initiate their communication outside the block-chain, which also means that it cannot access an endpoint that is outside of the blockchain. To achieve this type of communication, dedicated modules can be added to the blockchain (in the case of Hyperledger Sawtooth and Substrate), so these modules are deployed next to the smart contract and they take part in the blockchain node. In this case, the smart contract can call these modules that have dedicated functionalities to communicate with the exterior (sending messages to end devices for example). In the case of Ethereum, it is not possible to add new modules, however, event logs can be generated in the smart contracts. End devices can subscribe to these event logs, thus the communication from smart contract to end-device (to the exterior) can be achieved. Adding this latter type of module is not a necessity, but it can be useful in IoT applications. For example, when an IoT sends data to the API of a local storage system, the API does not allow the data to be stored until the smart contract confirms that the IoT has been identified for the ecosystem. A such Blockchain-IoT-IPFS network architecture applying the logic of authentication and data storing is proposed by this thesis contribution (see page 76).

It should be noted that the transaction validation rate is also an important characteristic that should be taken into account while choosing a blockchain. The transaction validation is the time that passes while the transaction is accepted by all the members of the network and is added to the blockchain. The requirement of this rate is of course application-specific. In blockchain white papers, this transaction rate is rather a theoretical value than a realistic one. Several ongoing research works are known to determine the real transaction validation rate of blockchains [8] [92] [93] [94] [95] taking into account the number of nodes, the consensus algorithms, the blockchain parameters (e.g., maximum number of transactions in a block), as well as the infrastructure behind the blockchain implementation i.e., hardware resources and their management.

3.3 Ethereum

This thesis work studies Ethereum [26] because this blockchain is one of the most popular blockchains among developers, beta users, and crypto-experts. Other points of view of the choice are: Ethereum allows the communication for APIs through a JSON-RPC interface (exist in each node of the network). The most commonly used API among the users and developers is the web3.js[1] Ethereum JavaScript API, this API allows to interact with an Ethereum node using HTTP, IP, or WebSocket. In this thesis work a client API was developed for IoT devices in C++ language (see page 61). One more argument in favor of choosing Ethereum is that today this blockchain can be considered as the reference among the public blockchains, and most research works target this blockchain.

Ethereum was the first blockchain allowing the deployment of smart contracts, which

[1] Web3 is available on https://web3js.readthedocs.io/en/v1.3.4/

[2] The logo retrieved from https://ethereum.org/en/assets/

Figure 3.1: Logo of Ethereum[2]

revolutionized the blockchain domain (beginning of blockchain 2.0), and gave rise to more applications than just secure cryptocurrency transactions between blockchain participants. However, Ethereum public blockchain has a cryptocurrency called Ether, and it is possible to send Ether among the members in the traditional way as well as according to dedicated requirements outlined in smart contracts.

Ethereum was mainly invented to be used as a public blockchain, but it can also be deployed as a private blockchain. In both public and private networks, two types of nodes can join the network: miner and client nodes. Miner nodes participate in the consensus rule. When PoW consensus is used, miners are rewarded after adding a new block. It should be noted that the transaction sending and functions calling in the smart contracts are not free on Ethereum public blockchain. The transaction and smart contract pricing would be described later in this section. The reward of the miners is the value that was paid for sending a transaction (or smart contract interaction). Client nodes contain the copy of the blockchain, so they take part fully in the network, they can participate in the process of verifying transactions, sending transactions, and calling/deploying smart contracts. The main advantage of private Ethereum is that the cryptocurrency can be hidden from the users, and the users can benefit from all the features of Ethereum without paying a certain amount of Ether for a smart contract using and transaction sending.

When deploying an Ethereum private network, the developer can choose other consensus rules than PoW, such as Proof-of-Authority (PoA) or Proof-of-Stake (PoS). Using other consensus in Ethereum can increase the validation rate of transactions (e.g., by using PoA [8] higher transaction rate can be achieved).

3.3.1 Smart Contract

Ethereum smart contracts are written in Solidity programming language; after compiling the smart contracts into bytecodes, they can be deployed on the blockchain. The deployment is done by sending a transaction containing the byte code of the smart contract. The transaction sender is considered the owner of the smart contract. Once a smart contract has been deployed, it would have a unique address, which must be addressed when the users want to interact with this given smart contract.

The smart contracts are run by Ethereum Virtual Machine (EVM), a stack machine that executes instructions encoded in the bytecode. The bytecode can be translated into assembly code for the stack machine as shown in figure 3.2. The functions described in a smart contract can be called through a transaction. Thus the transaction contains the Application Binary Interface (ABI) encoded inputs of the function. The signed transaction is Recursive Length Prefix (RLP) encoded that allows Ethereum to decode the arriving transactions. An example of RLP is shown in figure 2.5 (p.17). This RLP-encoded transaction can be considered as a data structure, that can be read by the blockchain with the help of RLP decoding (more details on page 61). When using a transaction for smart contract interactions, the procedure would cost a certain amount of fee (called gas in Ethereum) according to the instructions of

the EVM machine has to run. It should be noted that the smart contract's variables are stored in the storage state of the blockchain. When the value of a variable should be set or changed a transaction would be used to do it, and thus it costs a gas.

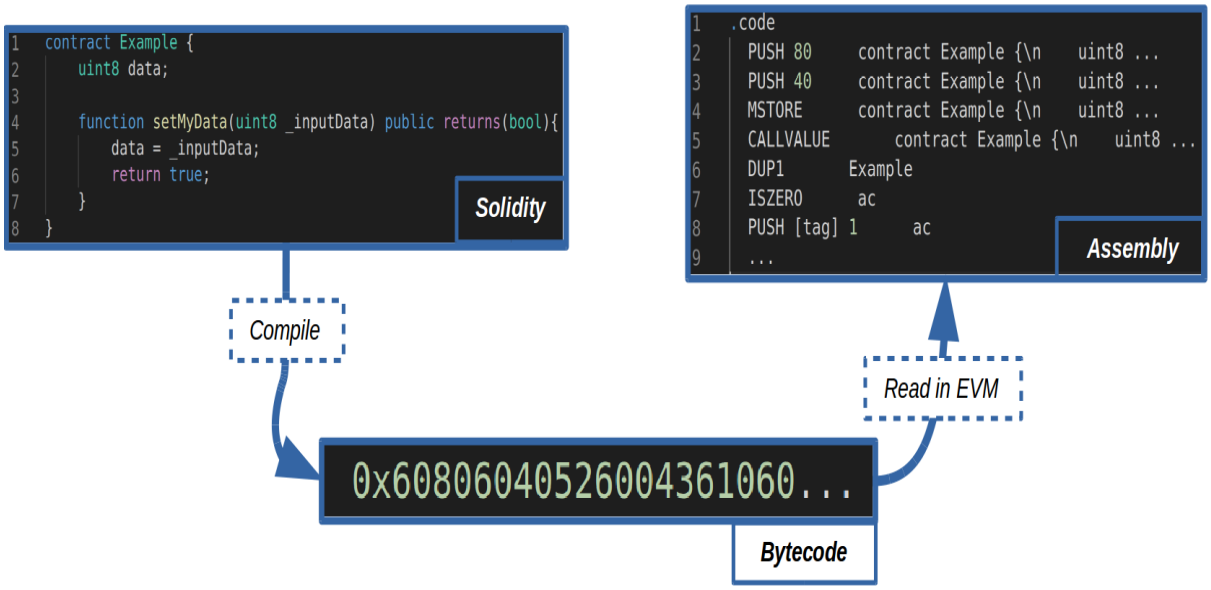


Figure 3.2: An example of a smart contract written in Solidity. E.g., bytecode 0x60 is equal to PUSH instruction.

On the contrary, when the blockchain state should be read, for example, get the value of a variable, this procedure is free of charge. In this latter case, there is no need for a transaction; it can be done with a call method. This call method is also called the verification tool of the behavior of the smart contract, as this call method does not make any change on the blockchain's state. Another exciting characteristic of Ethereum smart contracts is the event log. Smart contracts allow executing event logs that can contain all types of data, for example, the result of the called function. External applications can subscribe to these events and thus retrieve the data content of the given log. For example, an application will open the car doors only if the smart contract has identified the owner, so the log would contain a data about the authorization of door opening.

3.3.2 Transaction content and transaction flow

According to the information previously described, it can be noted that the transactions are the key point to be able to make cryptocurrency transactions and interact with smart contracts. According to G. Wood, Ethereum is a "*cryptographically secure, transaction-based state machine*" [26]. An Ethereum transaction contains more than a value of Ether that a member wants to send to another member or a command that enables to call a smart contract's function.

A transaction T is composed as follows:

$$T \equiv (T_n, T_p, T_g, T_t, T_v, T_d, T_w, T_r, T_s) \tag{3.1}$$

Where:

- T_n is the number of transactions that was sent by the sender, this value is also called the nonce.
- T_p is the gasPrice. Previously the reader may have become aware that the transaction sending and interactions with smart contracts are not free. For executing a smart contract, a certain amount of computation is required that costs a certain amount of gas (e.g., the ADD instruction in a smart contract costs three gases). The gasPrice is the amount of Wei (derived from Ether) that must be paid for each gas unit (e.g., the gasPrice is 2 Wei, then the computation of ADD costs 3*2 thus 6 Wei).
- T_g is the gasLimit that is the maximum quantity of gas available for the transaction execution; when the transaction execution requires a higher value of the gas than the gasPrice, the transaction would be rejected. This value can verify smart contract behavior and prevent spending too much gas. For example, if an infinite loop occurs in the smart contract, this could empty the transaction senders' wallet.
- T_t is an address of an account (member of the blockchain owning a wallet) or a smart contract that the transaction sender wants to call. The address is composed of 160bits. This variable is equal to zero when the transaction is sent for smart contract deployment. The smart contract's address will be automatically generated after it has been deployed.
- T_v is the value in Wei that the transaction sender wants to send to a participant by using the participant's address. This value can also be an endowment for smart contract deployment.
- T_d[3] this component is for the function call of the smart contract. This component must be formed to be understandable for a smart contract, because this entity contains the name of the smart contract's function and also the input data for the function.
- T_w , T_r, T_s The transaction is signed by the private key (32bytes) of an account. These three components (T_r and T_s forming the signature) are required to verify the transaction signature. The blockchain node that receives the transaction verifies if the transaction was signed by the transaction sender. It can be done with T_r, T_s, T_w, and the address of the sender's account is derived from the sender's public key. This address is equal to the first 20 bytes of the hash of the sender's public key. The signing and verifying process, as well as the importance of the components r, s w, are detailed in chapter 4.

It should be noted that all of these components are RLP encoded and concatenated to obtain the T. The sizes of the components can be summarized as follows:

$$T_n, T_p, T_g, T_t, T_v, T_r, T_s \in [\,0; 2^{256} - 1] \tag{3.2}$$

$$T_w \in [\,0; 2^5 - 1] \tag{3.3}$$

$$T_d, T_i \equiv \text{an unlimited size byte array} \tag{3.4}$$

[3] When the transaction deploys a new smart contract, this component is equal to the bytecode of the compiled smart contract. In that case, the component is called T_i.

It can be summarized that transactions can be considered the blockchain's inputs or inputs of the blockchain "state machine". The phases that occur with the transaction until it is validated and therefore added to the blockchain are the following:

1. Transaction creation with respect to the format requirements. RLP encoding of the transaction's components ($T_n, T_p, T_g, T_t, T_v, T_d$) and ABI encoding for smart contract calls (data component T_d, or smart contract deployment component T_i). After encoding, the components are concatenated, and thus they form the core of the transaction.

2. Transaction signing. This is done with the private key of the transaction sender (private key of the account that the sender has, each account has a wallet containing an amount of Ether). After signing, the transaction is completed with the components T_w, T_r and T_s, which are the result of the signature. These components are concatenated with the components of the first phase, so the transaction is now signed (see equation 3.1), and it can be sent to one of the blockchain's nodes related to the sender's account.

3. The singed transaction is sent to a local node that is related to the sender's account. This node verifies the signature and the format of the transaction. If both of them are correct, the transaction passes to the 4th phase; else, the transaction is rejected.

4. The transaction is broadcast among the members participating in the consensus (validators). The transaction is appended to miners' queues.

5. In the next step, validators apply the consensus algorithm in order to agree on the transactions that can be added to a block and on the validator that can add the new block to the blockchain.

It should be noted that a member is identified for the blockchain by its cryptographic public key that is derived from its private key. The public key is reported to the Ethereum network when a new account is created for the new member. The account is also associated with a wallet containing the Ethers.

The C++ implementation of the API proposed by this thesis work can be found on page 61. This implementation enables to sign and send the transaction to a local node (phases 1,2 and 3 described above).

3.4 Hyperledger Sawtooth

It's worth analyzing the Hyperledger Sawtooth [28] [96] because it seems a promising blockchain as it is based on a modular architecture (modules can be added and modified easily, this feature would be discussed later); it is also an open-source Linux foundation project that is always a prominent advantage in the academic research domain. Another reason is that this blockchain is equipped with a REST API interface allowing the interaction with end-users APIs (particularly with an IoT device). Finally, this blockchain allows deploying smart contracts in multiple languages such as Python, Rust, C++, and many others. Hyperledger Sawtooth provides several official SDKs (e.g., JavaScript, Python, Go, and others) for interaction with the blockchain, allowing transaction creations and signing. However, today,

Figure 3.3: Hyperledger Sawtooth's logo[4].

there is still no C/C++ implementation available. This thesis work proposes a C++ client API for IoT devices that enables the interaction with Hyperledger Sawtooth.

Hyperledger Sawtooth blockchain platform allows building distributed ledger applications and also networks. This platform is also considered an enterprise blockchain. As the Hyperledger Sawtooth targets enterprises and DLT applications, it can also be applied to build distributed ecosystems. This blockchain can be deployed as a public, private, or consortium network. The Hyperledger Sawtooth consortium network allows only specific nodes to join the network and participate in the consensus, and it allows everyone to send transactions to the blockchain network[5]. An ecosystem that is based on a consortium blockchain meets the requirements of the SIM project because, in a consortium, only particular organizations can participate, for example, Renault, an insurance company, and a car manufacturer. In addition to particular organization participation, every other actor, for example, vehicles, can send information to the ecosystem. These information are managed according to a business logic deployed in a smart contract. It should be noted that Hyperledger Sawtooth does not contain a cryptocurrency by default, but it can be implemented.

The Hyperledger Sawtooth network consists of nodes that are called validators. In practice, each validator participates in the consensus rule, but it is not an obligation. The network architecture of Hyperledger Sawtooth and the Validator node content are presented in figure 3.4.

The Validator node's core is the module called Validator that is written in Rust or Python programming languages depending on the version of the implementation. The Validator contains sub-modules, such as the P2P Network that manages the connection between the Validator nodes participating in the network, the State module that is the database of the blockchain (it has the similar meaning such as in the case of Ethereum, see section 3.3, batches containing the transactions are atomic units for state change), Block Management and Transaction Handling modules are used for the computing of the transactions and blocks, and finally, the Interconnect module serves to connect the modules that are next to the Validator. The architecture of Hyperledger Sawtooth is configurable, and new modules can be added next to the Validator, and existing modules can also be modified or changed. The Consensus Engine allows using the desired consensus rule, which can be chosen before the blockchain is built or can be changed dynamically when the blockchain is already in a running phase. The default consensus rule is the PoET (see page 16). However, Hyperledger Sawtooth also provides an official PBFT (see page 16) implementation. In this thesis work, the PBFT consensus was applied because the related works highlighted that PoET is not yet in a stable state. After a certain amount of transaction validations, the blockchain network halts when applying PoET

[4]The logo retrieved from: https://www.hyperledger.org/blog/2018/01/30/announcing-hyperledger-sawtooth-1-0

[5]The characteristics of public and private Hyperledger Sawtooth can be found on https://sawtooth.hyperledger.org/docs/core/releases/latest/architecture/permissioning_requirement.html

[6]The figure of the architecture is retrieved from: https://sawtooth.hyperledger.org/docs/core/releases/latest/architecture.html

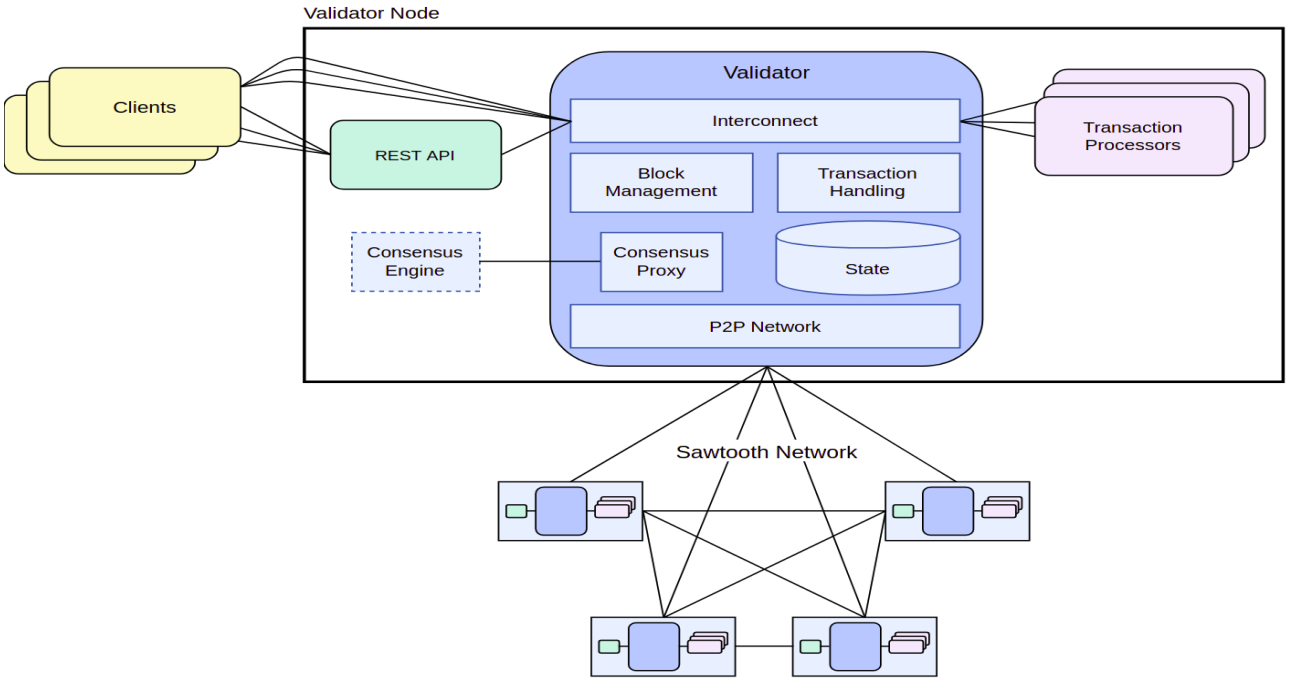


Figure 3.4: Hyperledger Sawtooth's network architecture and Validator node content[6].

[92] [93] [9].

The REST API module allows the Client (APIs) to communicate with the Validator through batches containing transactions. The Transaction Processors are the modules that describe the desired business logic, and they can be considered as smart contracts.

3.4.1 Smart Contract (Transaction Processor)

Hyperledger Sawtooth has two different types of business logic that can be applied, the native and the Ethereum smart contracts (in Solidity) executed on EVM.

The native business logic (transaction processor) can be deployed when the blockchain is built, and it can be written in Python, C, C++, Java, JavaScript, Rust, and Go programming languages. This feature facilitates business logic development by giving developers the freedom to choose the programming languages they want.

One disadvantage, is that, unlike Ethereum's smart contracts, this type of "smart contract" cannot be deployed by sending it in a transaction.

In practice, these smart contracts are written by the blockchain owner (admin), who built the blockchain network. There is no limit to the number of transaction processors. Each transaction processor has a family name that is equivalent to its ID. The transaction processor under the same family name can be cloned, and they can be executed in parallel. The parallel execution can increase the performance of the blockchain in terms of the transaction validation rate. The deployment of this native business logic is implemented not only in Hyperledger Sawtooth, but also in the Substrate blockchain framework. An example of a transaction processor (smart contract) written in python is shown in figure 3.5 (the Intkey transaction processor is a basic processor for incrementing a value).

```python
class IntkeyTransactionHandler(TransactionHandler):
    @property
    def family_name(self):
        return FAMILY_NAME

    @property
    def family_versions(self):
        return ['1.0']

    @property
    def namespaces(self):
        return [INTKEY_ADDRESS_PREFIX]

    def apply(self, transaction, context):
        verb, name, value = _unpack_transaction(transaction)

        state = _get_state_data(name, context)

        updated_state = _do_intkey(verb, name, value, state)

        _set_state_data(name, updated_state, context)
```

Figure 3.5: Example of Hyperledger Sawtooth transaction processor (a portion) written in Python.

The ”seth-tp”[7] transaction processor allows executing Ethereum smart contracts written in Solidity. This transaction processor is equipped with an EVM that runs the smart contract in the same way as Ethereum does. The deployment of such a smart contract is done by sending the smart contract’s bytecode in a Hyperledger Sawtooth transaction.

3.4.2 Transaction content and transaction flow

In Hyperledger Sawtooth, the transaction creation and its flow are different from Ethereum that was described previously (see section 3.3, p.46). In Ethereum, the sender signed the transaction to communicate with a smart contract or send Ether to another participant.

In the case of Hyperledger Sawtooth, in addition to creating one or more transactions, a batch must also be created in which the transaction(s) will take place. Using a batch for multiple transactions allows creating dependencies among the transactions. For example, transactions A, B, and C are dependent on each other, and it is supposed that transaction C could not exist until B was not created. By adding the transactions to the batch according to the order A, B, C, and not A, C, B, the dependency is respected. It is also important to note that if one transaction among the transactions contained by the batch is invalid, the batch could also not be validated.

In the following, it is considered that the batch contains only one transaction to simplify the descriptions. So the sender sends the batch to the blockchain, a component called ”Completer” verifies the batches and the blocks dependencies. The verified batches are sent to the BlockPubliser component that validates the batches according to the consensus rule, and it creates a block with the validated batches. This block is emitted to the ”Completer” that verifies the block’s dependencies. After verification, the ”ChainController” element receives the verified block added to the blockchain state. The ”ChainController” also decides the blockhead, which is the last confirmed block in the case of blockchain forks. When using PBFT in Hyperledger Sawtooth, there is no forks creation.

It must be noted that the batch and the transaction can be signed with the same private

[7]Introduction about Sawtooth Seth could be found here: https://sawtooth.hyperledger.org/docs/seth/releases/latest/introduction.html

key of the sender. However, in practice, it is better to use different keys to increase security. If one of the signatures does not valid, the batch and the transaction are immediately rejected. In IoT applications, this signature method can also be applied. An IoT device could sign its transaction by its private key and send the transaction to a gateway that would add the transaction to a batch, signed with the gateway's private key. In the vehicle accident use case, the car owner signs the transaction containing the data about the vehicle's environment. The car signs the batch with its private key. In this case, the security is improved because the data of the vehicle cannot be sent to the blockchain if the car owner does not exist and vice versa.

The figure 3.6 shows the contents of the batch and the transaction that are structured with google protocol buffers. The protocol buffers can serialize structured data. Protocol buffers are platform-natural, which means that the skeleton of the data structure (an XML-like language, but faster and smaller) can be transformed into several languages as C/C++, Python, Java, and many others. This characteristic allows using the defined protocol buffer data structure among multiple endpoints using different languages. Batch and transaction creation are done by two protocol buffers.

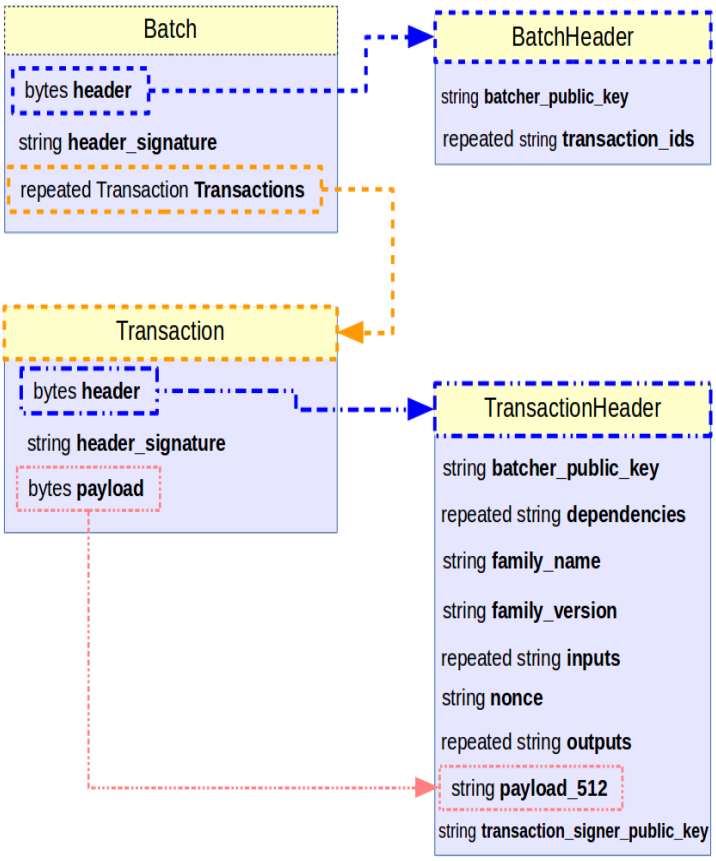


Figure 3.6: Hyperledger Sawtooth batch and transaction content.

- *Batch*
  The batch contains the serialized version of the $BatchHeader$, which means that the header's raw data is serialized into a hexadecimal string segment or bytes segment. The batch also contains the digital signature of the $BatchHeader$ ($header\_signature$) that is signed with the batch signer's (or "batcher") private key (e.g., gateway's or car's

private key). Finally, the batch contains the transaction. It can also be noticed that a transaction is an object here as the transaction is structured with another protocol buffer than the batch.

- *BatchHeader*
  This header contains the batcher's public key that is used for the header's signature verification when a batch arrives in the blockchain.

- *Transaction*
  The transaction contains the serialized version of the $TransactionHeader$, the digital signature of the $TransactionHeader$ ($header\_signature$) signed by the transaction signer's (e.g., car owner) private key. And finally, it contains the $payload$ (e.g., the data about the car's environment, recorded before the accident happens) that is by default stored in a CBOR (Concise Binary Object Representation) encoded data format. However, it is also possible to apply a protocol buffer, but in both cases, the transaction processor (smart contract) has to include tools for decoding the CBOR or protocol buffer formats.

- *TransactionHeader*
  This contains both the public key of the batch and the transaction signers, which are required for batch and transaction verification procedures when the batch arrives in the blockchain ($batcher\_public\_key$ and $transaction\_signer\_public\_key$). It also contains the $dependencies$ field, which is the information about the dependencies of the transactions of the given batch. The family name and version components ($family\_name$ and $family\_version$) refer to the family of the transaction processor and its version that the transaction intended to call. The transaction processors of Hyperledger Sawtooth use a state dictionary to store the variables data to the blockchain state. For doing so, addresses are used as pointers in order to access the variable's state. In the transactions, the variables that would be modified or read must be specified. The parameters $inputs$ and $outputs$ are used to specify the mentioned variables (this is also called the address creations of the variables that are the pointers to the variables in the blockchain state). The $nonce$ component has the same functionality as in the case of Ethereum. The nonce value must be different for each transaction sending; else, the transaction is rejected. This mechanism avoids the not-attended network overloads (DDoS attack). In Ethereum, this value was incremented after each transaction sent. In the case of Hyperledger Sawtooth, this value is a random number that is generated thanks to a seed and the Operation System's random number generator. Finally, the $payload\_512$ component is the hash of the payload taking place in the $Transaction$ described above. The hash is the result of SHA-512 (length of 512 bit) cryptographic hash function.

In the case of Ethereum, the public key of the transaction signer is stored in the blockchain, and it is used for signature verification. If the network member does not have an account, the member could not send signed transactions for interacting with smart contracts. In Hyperledger Sawtooth's case, at least two signatures must be done (batch and the transaction) at the endpoint (device). However, contrary to Ethereum, in Sawtooth, the public keys are not necessarily stored in the blockchain because they are sent in the $TransactionHeader$ and the $BatchHeader$. Thus, the blockchain can simply retrieve the public keys of the batch and the

transaction signer, which allows it to verify the signatures of the header ($TransactionHeader$ and $BatchHeader$). It must also be noted that Hyperledger Sawtooth can force the public key storage. If it is the case, existence of the public key found in headers would be verified before verifying the transaction signature. However, in consortium deployment, this latter characteristic is not necessary.

The description of the client API in C++ that we propose in this thesis work can be found on page 67. This implementation enables to sign and send a batch with a transaction to a Validator's REST API. This implementation is based on previous works[17] and [9] which ones provide a large base of this thesis contribution.

3.5 EOS.IO

The study of EOS.IO makes sense, because this open-source blockchain platform has been designed to be used in industrial and enterprise use cases [56] [97]. EOS.IO also encourages the creation of dApps (decentralized Applications) which number surpasses Ethereum's.

Likewise, in Ethereum, an account is required to be able to send valid transactions. The account also contains a crypto wallet. According to EOS.IO, an account can be associated with a group or an organization. This blockchain platform can therefore form the core of an ecosystem. The main net of EOS.IO is a public blockchain, and its cryptocurrency is called EOS. Unlike Ethereum, in EOS.IO, the transaction sending is free, but new contract creation costs a certain amount of RAM that is equal to a certain amount of EOS. In this blockchain, the members can buy "resources" as RAM, CPU, and NET (network). The CPU and NET can be bought for obtaining stakes that are used in DPoS consensus (see page 16). The wealthy members have more significant weight in the voting process of block producer nodes. The block producer nodes validate new blocks according to the consensus rule, and the winner of voting has the right to publish the validated block. The producers are rewarded with inflation of the sum of CPU and NET. This blockchain contains producing and non-producing nodes. The non-producing nodes cannot validate and publish blocks. However, they are used to validate transactions, relay API calls, and broadcasting information. In the default implementation of EOS.IO, the number of producer nodes is limited to 21, and they are revoted for a predetermined interval.

EOS.IO can also be implemented as a private blockchain, and its core can be modified that opens possibilities to developers to achieve better performances.

3.5.1 Smart Contract

The smart contracts are written in C++ programming language, and it is compiled into a WebAssembly (WASM) bytecode. It should be noted that EOS.IO provides its official library containing EOS.IO specific classes to allow developers writing their smart contracts. EOS.IO uses a WebAssembly Virtual Machine (WASM VM) for executing the compiled smart contracts. This VM should not be confused with Ethereum's VM. Likewise, in Ethereum, a transaction can call a specific function or functions (in EOS.IO, they are actions) of the given smart contract. The compiled smart contracts can also be represented in ABI form, which helps to convert actions in JSON to binary representation. The JSON format of actions is used when creating the transaction with the corresponding transactions.

One of the main advantages of EOS.IO's smart contracts is the deployment possibilities of Ricardian contracts, which were described in a previous section (see page 21). As a reminder, Ricardian contracts provide a comprehensible representation of the results of smart contracts in terms of possible inputs.

3.5.2 Transaction content and flow

There are two types of transactions, the signed transaction, and the packed transaction instance. The singed transaction's components and their description are given in the table 3.1.

Component	Description
expiration	This component is optional. This amount of time indicates that the transaction must be validated before the given time, else the transaction must be rejected. This option can increase the security while crypto-transfers, for example.
ref_block_num	The block number of the last blocks, this number can be retrieved from the blockchain with the help of the inbuilt "get" function.
ref_block_prefix	This is the identifier of the block that is referred to "ref_block_num". This prefix can be retrieved from the blockchain with the help of the inbuilt "get" function.
max_net_usage_words	This component is related to the previously mentioned NET value. If the value of this component is greater than zero, the transaction would be billed.
max_cpu_usage_ms	This component is measured in ms according to the execution time of the transaction in the CPU (which was also mentioned earlier). Likewise, in the "max_net_usage_words" component, if the value is greater than zero, the transaction would be billed.
delay_sec	Component to delay the transaction (in seconds).
context_free_actions	The computation of these actions depends only on the transaction data, which means that the blockchain state is not accessed.
actions	This list describes the actions that the transaction calls in the smart contract (actions are the functions described in the smart contract).
transaction_extensions	This component serves for support additional features.
signature(s)	The digital signature of the serialized transaction (hexadecimal string format).
context_free_data	This data can be stored temporarily on the blockchain because it is not stored in the blockchain state.

Table 3.1: Content of the EOS.IO transaction.

The packed transaction is a compressed signed transaction, allowing faster verification on the blockchain side. The packed transaction's components and their description are given in the table 3.2.

Component	*Description*
signature(s)	The digital signature of the signed transaction.
compression	Indicates which compression method was used.
packed_context_free_data	The compressed context free data (see tab. 3.1).
packed_trx	The compressed version of the transaction.
unpacked_trx	The transaction before compression.
trx_id	The unique identifier of the transaction.

Table 3.2: Content of the EOS.IO packed transaction.

A transaction can be signed in an external API (e.g., with eosjs JavaScript library) or by using "cleos" client interface allowing the communication with a node. A transaction is signed with the account's private key that is retrieved from the corresponding wallet. After signing, the signed transaction (transaction with the signature) can be sent to the local node (producer or not). The local node relay this validated transaction to an active producer node that will verify the signature, execute and validate the transaction. The transaction is then

added to a block, which must be validated by the other producers. When the supermajority of validation is achieved, the block remains immutable.

On page 69, this thesis describes the first open-source C++ API that allows to create and sign EOS.IO transactions for interacting with smart contracts. In addition to describing the API, improvements for more optimal execution are also proposed.

3.6 Substrate

Substrate is an open-source blockchain framework that allows developers to modify the blockchain architecture to meet their specific requirements [98].

Today a given use case has to study multiple blockchains to decide if it can meet the use case requirements. And often, the requirements have to be modified to be able to use with a blockchain. Maybe this framework can be a new approach that allows developers to first study the given use case and then develop a blockchain that meets the use case needs.

The developers can use Substrate in three different ways. Using Substrate node means that the default node architecture would be used, but it is possible to configure the genesis block's parameters allowing to obtain different running characteristics. In the second mode is called the Substrate FRAME mode, it is possible to change the modules (so-called "pallets") of the Substrate node. In this mode the developer does not have to write brand new modules, there are also pre-existing modules provided by the Substrate libraries. The third way is called Substrate Core, in which the core elements can also be modified, for example, block header's structure or block serialization formats.

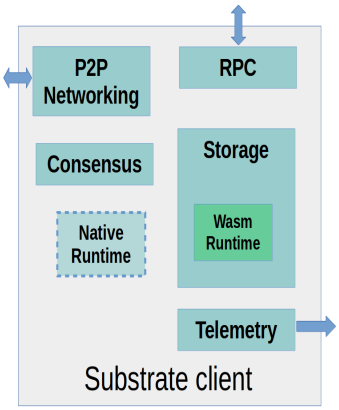


Figure 3.7: Substrate modular architecture[8]

It was described previously that the Substrate client node's modules could be changed. The modules which form the Substrate node are depicted in figure 3.7. The Substrate node consists of a Storage module that handles the blockchain state. The Runtime module determines the block processing mode and the logic of the state transaction. This module is compiled into Wasm. The Native Runtime module can also be used for the same reason as the basic Wasm Runtime module. The Native Runtime module can be more efficient than the basic Wasm Runtime module, depending on its implementation. The P2P network is the module for establishing the connection and communication among the nodes of the

[8]The figure is derived from https://substrate.dev/docs/en/knowledgebase/getting-started/architecture

blockchain network. Likewise, in Hyperledger Sawtooth, Substrate also allows choosing the desired consensus rule by changing the Consensus node. The development of own consensus rules is also possible However, there are pre-existing ones, such as Aura, BABE, PoW, and GRANDPA.

An RPC module provides the users' interactions with a node. The last module to announce is the Telemetry that retrieves the metrics when the blockchain already runs. For example, Grafana can visualize the Telemetry's data (CPU, memory usage, etc.).

By default, the transaction sending is not free in Substrate. Using a transaction fee is encouraged in order to prevent eventual DDOS attacks. The transaction fee is calculated in terms of the byte length of the transaction multiplied by a fixed weight, and another fixed weight is added to this value. It should be noted that the zero-fee transaction can be implemented by changing a few parameters of the blockchain

3.6.1 Smart Contract

Substrate allows the deployment of two types of smart contract virtual machines. It can be implemented with an EVM module that can execute smart contracts in the same way as Ethereum does. The second possibility is the use of a contract module that can execute WebAssembly contracts written in "ink!" programming language. It should be noted that a Runtime module can also implement business logic, this module is similar to Hyperledger Sawtooth's transaction processor. This module has more possibilities to modify the state of the blockchain and interact with other modules intrinsically.

3.6.2 Transaction content and flow

After signing the transaction with the account's private key, the signed transaction is sent to a node. The transaction verification phase is equivalent to the verification of the correct nonce, the verification if the transaction signer has enough funds for the transaction sending, and finally, the transaction signature verification has to be done. After this phase, the transactions are set into the transaction pool. The pool is broadcast among the peers. The execution of the transaction results in changes in the state. These statements are stored in the local memory before the transaction is set into a new block. When a block is filled, it is broadcast and executed in each of the peers.

In Substrate, the transactions are also called extrinsic, which are encoded with SCALE codec. The components of a transaction are the follows:

- Nonce indicates the number of transaction that was sent by the account (the transaction emitter).
- Era indicates the mortality of the transaction. In the era, a period can be given within the transaction can be validated. This period is counted from a checkpoint (hash of a block) to a certain number of blocks.
- The call component contains the module ($call\_module$) that is equivalent to a smart contract that the transaction wants to interact with. This component also contains the function and its parameters that the transaction aims to call in the smart contract ($call\_function$, $call\_params$).

- The transaction also contains the signer public key ($account\_id$).
- A tip parameter can also be set, which allows getting priority in case of network traffic saturation.
- Finally, the transaction signature and its version are set to the transaction ($signature$, $signature\_version$).

It can be summarized that Substrate is a promising framework to build and test new blockchain architectures. Another great advantage of Substrate is that, during the execution of the architecture, the user has a web interface to facilitate the understanding of the system operation. In other blockchain platforms, the logs of system operations are logs in the terminal that is hard to understand. However, the Substrate platform is an academic work that does not have enormous industrial users at this time.

On page 71 the requirements to build a C++ interface allowing the communication between an IoT and a Substrate node is described.

3.7 Blockchain APIs in C++ for the off-chain approach

This section describes the necessary operations that are required to establish the communication between an IoT device and the given blockchain. The section also presents the developed C++ tools and slightly modified open-source tools that allow the creation of valid transactions for a given blockchain.

The proposed C++ tools (APIs) are analyzed to determine their behaviors and call graph analysis is also done in order to identify the most significantly called functions of the given program. This contribution can also show which parts of the given program can be optimized by software or hardware modifications.

It can be noted that this thesis work focuses on the possibilities of hardware optimization to achieve a better performance in terms of execution time and energy consumption. The most of the proposed C++ APIs were tested on a Raspberry Pi 3 B+ IoT device or tested on QEMU emulated ARM-based architectures.

It should be noted that the C++ APIs that we developed could also be run on other more constrained IoT devices. However, it is possible that some architecture specific libraries must be included, to achieve a correct run on the given platform.

3.7.1 Simple transaction sending

In the following parts of this subsection, the proposed APIs create transactions according to the requirements of the given blockchains. In this subsections, more details are given on formatting requirements and on the APIs characteristics. The transaction that the APIs create is called a simple transaction because it contains only a few bytes of information (the payload size is 32 bytes long) that would be sent to the given smart contract.

32 bytes long payload size was chosen because this is the typical size of hash value that could be sent to the blockchain. Previously it was already mentioned that in several IoT-blockchain applications it makes more sense to send only the hash of the raw data to the blockchain, and raw data to another storage system. Section 3.8 (p.74) provides more details

about why hash sending could be a better choice in IoT application in which the data size to send is significant. This section also describes a proposed blockchain-IoT-IPFS network architecture, in which the hash of the raw data is sent to Hyperledger Sawtooth blockchain and the raw data to IPFS. The results also demonstrate how the hash creation influences the API's execution time.

It can be noted that the data size that is sent to the blockchain is application-specific, and it can cause different behaviors in the API execution. For example, the API would behave differently when an IoT sends a temperature measurement (light data size around 32 bits) or when it sends a significant amount of data about a vehicle's environment (hundreds of MBytes). In each section, a result of the experiment (call graph) is given to show the given API's behavior. It should be noted that each of the APIs creates a final valid transaction that would be able to interact with a smart contract. The experiments measure these phases of valid transaction creation, but the sending phase is not measured. The sending phase depends on the communication protocol used (Wi-Fi, Bluetooth, LoRaWAN, etc.), the bandwidth of the channel, and also on the response time of the given blockchain (this response time can also vary when a blockchain is deployed with different parameters).

The call graphs or call trees are obtained by Callgrind that is a tool of Valgrind dynamic analysis tool used to generate call trees and to debug and profile Linux programs. The results can be visualized with KCachegrind[9] providing an interactive user interface. In contrary to gprof profiler, the analyzed program should not be compiled with the "-pg" option. The results of gprof are difficult to understand, and the visualization tools do not provide an interactive user interface, unlike Callgrind. It can be easily called with the command line *valgrind –tool=callgrind -v executable*. The result can then be visualized with KCachegrind.

3.7.1.1 Ethereum

The proposed tool enables the interactions with Ethereum smart contracts, implemented in C++. It should be noted that there is no official Ethereum C++ library that allows transaction creation and offline signature for interacting with a smart contract. Web3 Ethereum JavaScript API (web3.js) is a collection of libraries that allows these functionalities. However, JavaScript was not primarily invented to use in IoT devices, it is more useful in dApps. Some web3.js logic is implemented in the proposed tool, and some of its features are based on open-source initiatives.

The aim of the tool is to communicate with an Ethereum smart contract written in Solidity language. For doing so, the transaction has to be formed according to the requirements of the Ethereum blockchain and smart contracts. It must be noted that the transaction contents representation is hexadecimal (Hexa string) encoded. For example, the value 0x96 logically can be set into an *uint8_t type* (1byte), however in this environment, for representing 9 and 6, two bytes are needed as it is Hexa string encoded information. This representation eases the understanding of the contents for users. However, in the point of view of embedded systems is a drawback because each of the data takes twice more places, and also, the data size of the transaction that is sent to the blockchain is doubled. The Hexa string representation is used in all of the studied blockchains.

The function described in the smart contract can be called as follows. The name of the

[9]KCachegrind documentation is available at: https://kcachegrind.github.io/html/Documentation.html

function and the types of its inputs (e.g., *setMyData(uint256,uint8)*) must be hashed by the Keccak-256 function, the first 4bytes of the Hexa string form (the first eight characters of this representation) identifies the function (also called Method ID). The hash of Keccak-256 hash of *setMyData(uint256,uint8)* is ***0xcd96e8fc****f08108dcb15eb3a3aa...*, the function's identity is thus ***0xcd96e8fc***. This component is followed by the content of the two variables (uint256,uint8). For encoding the values of these two variables, Ethereum uses an ABI encoding format. Solidity language allows using several types like uint8, uint16, uint32, uint256, bytes, bytes32, string, bool. Every type has its specificity according to the ABI encoding format. For example, the value 12 stored in a variable of type ***uint8_t*** in C++ corresponds to a ***uint8*** variable in the smart contract. 12's hexadecimal representation is 0xC. The ABI encoding of any ***uint*** variable is represented by 32bytes Hex string form, thus ***12*** is finally encoded as ***0x0000000000000000000000000000000...000000000000000000000000C***.

It should be noted that the uint8_t, uint16_t, uint32_t C++ types correspond to uint8, uint16, uint32 in Solidity, however, there is no uint256 type in C++. In practice, any uint array that does not exceed the total size of 256 bits (e.g., uint8_t [32], unit32_t [8]) can be serialized to a Hexa string representation that would have exactly 32bytes length that can be used in the smart contract as the type uint256. This operation can also be done to represent values in bytes32 type. It is also possible that the smart contract accepts an array as an input (e.g., uint8 [3]). In that case, the C++ uint8_t [3] elements are encoded one by one, (uint8_t [3] would be represented inside 3x32bytes Hexa string ABI encoded format). A smart contract can also have a string input. However, the C++ string type must also be ABI encoded. First, the utf-8 encoded characters hexadecimal codes are retrieved one by one and represented as a Hexa string. The concatenated Hexa string is zero-padded to the right to achieve a 32bytes Hexa string. Another 32bytes Hexa string field indicates the length of the string that is ABI encoded (e.g., Hello is a length of 5). And a prefix is also added that is the value 0x20 in 32bytes Hexa string. The figure shows some examples of ABI conversions.

C++ type variable	ABI encoded variable
uint8_t var = 12	*0x000000000000000000000000000000000000000000000000000000000000000C* (uint8)
uint8_t var[3] = {12, 13, 14}	*0x000000000000000000000000000000000000000000000000000000000000000C* *000000000000000000000000000000000000000000000000000000000000000D* *000000000000000000000000000000000000000000000000000000000000000E* (uint8 [3])
string var = "Hello" (*len*(var) = 5)	0x0000000000000000000000000000000000000000000000000000000000000020 0000000000000000000000000000000000000000000000000000000000000005 48656c6c6f000000000000000000000000000000000000000000000000000000 (string)

Figure 3.8: Examples of ABI encoding

The transaction is composed, as the equation 3.5 shows. This was already mentioned in a previous section 3.3 (p.46).

$$T \equiv (T_n, T_p, T_g, T_t, T_v, T_d, T_w, T_r, T_s) \tag{3.5}$$

It must be noted that the nonce (T_n) value must be incremented after every transaction sending. The gas limit (T_g) and the gas price (T_p) can be estimated and fixed. The API does not need to send a request to get this value from the blockchain before every transaction sending. The address component (T_t) of the transaction is equal to the smart contract's address. The value component T_v is not set when a smart contract is called. T_d is the previously

mentioned ABI encoded information for smart contract calling. To complete the transaction, the signature components (T_w, T_r, T_s) must also be set.

The transaction of form (T_n, T_g, T_p, T_t, T_d), must be hashed by the Keccak-256 algorithm, and the hash result must be signed. It must be noted that the components that would be signed must be RLP encoded, and the final valid (signed) transaction has to be also RLP encoded. The Ethereum blockchain uses this RLP format to be able to distinguish the fields of the transactions. The principle of RLP encoding is based on the byte length of the data.

The principle of RLP encoding is as follows:

- If the data to encode is represented in a single byte, and its value is in the range [0x00, 0x7f], then nothing to do.
- If the data is a string, its characters must be converted to a hexadecimal representation (utf-8). If its length is less than 55 bytes, then the RLP encoding is equal to *0x80* plus the length of the string, followed by the hexadecimal representation of the string's characters. For example, the string *cat* is encoded as follows *0x83636174*. The length of *cat* is equal to 3, so the first value is *0x83*. The hexadecimal value of *c* is *0x63*, *0x61* represents *a*, and finally, *0x74* is the hexadecimal code of *t*.
- If a string is longer than 55 bytes, the length of the string can be represented as a hex value. The number of bytes that is required for this representation is added to *0xb7* prefix. Finally, the hexadecimal encoded string is appended. For example the length-1024 is encoded as \xb9\x04\x00. Where *0x0400* is the hexadecimal representation of 1024.
- After each component is already RLP encoded, they are concatenated, forming a payload. This payload has to be RLP encoded again. If the payload length is less than 55 bytes, the RLP code is equal to *0xC0* plus the payload's length. The payload follows this code.
- When the length is greater than 55 bytes, for encoding the payload size the same logic viewed in the case of string length higher than 55 bytes must be applied. However, in this case the prefix is equal to *0xf7*.

It should be noted that data storing on the blockchain means that blockchain state changes. The sender must cryptographically sign the transaction that calls a function which changes the state of the blockchain. This is a logical step because the participant who has instantiated the change of the state must be identified. The Elliptic Curve Digital Signature Algorithm (ECDSA) is used to create the signatures of the transaction. This algorithm can use different types of elliptic curves for signing. Ethereum uses the secp256k1 elliptic curve, likewise Bitcoin. According to Bitcoin's documentations the operations on this curve are more optimal than on other curves. In the proposed API, the ECDSA digital signature and the Keccak-256 hash functions are called from trezor-crypto[10] cryptographic open-source and open-licensed C library.

In the following results, the program creates valid (signed and correctly formed) transactions that can call a dedicated smart contract. The payload of the transaction is a 32bytes

[10]Trezor-crypto available on :https://github.com/trezor/trezor-crypto

uint8_t array. The size of the payload is light in this scenario. It can be imagined that this 32bytes payload is equal to a hash value of a given raw data.

Call graph of the program

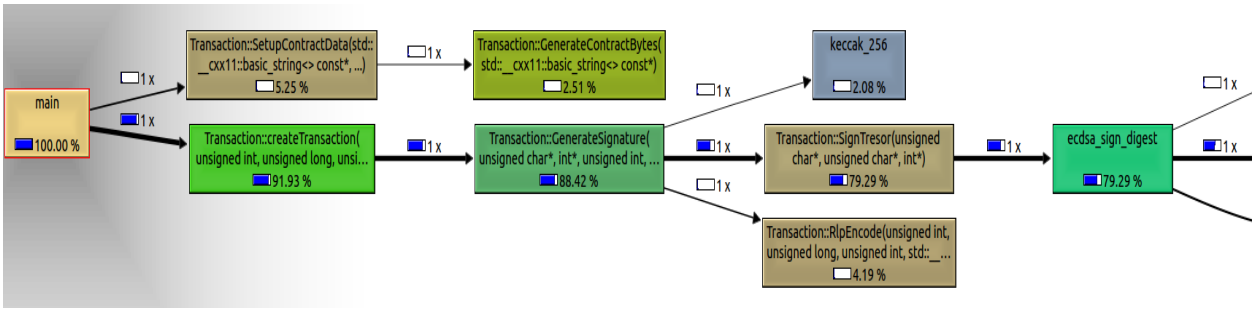


Figure 3.9: Call graph of Ethereum API, the *main* function is considered as the father function

Figure 3.9 represents the call graph of the API, in this analysis the main function is the "father" of the other functions (it takes the 100% of the execution time of the API). The only functions taking more than 2% of time occupations are represented.

The *SetupContractData* (5.25%) function process the ABI encoding of the transaction payload and create the smart contract's function identifier retrieved from a Keccak hash value. The *createTransaction* function occupies 91.93% of the program. To obtain the final transaction, the *GenerateSignature* function must be called first. This function takes 88.42% of the *createTransaction* function. This function is composed of *keccak_256*, *RLPEncode*, and *SignTrezor* functions. The *RLPEncode* function (4.19%) encodes the components $(T_n, T_p, T_g, T_t, T_v, T_d)$ of the transactions. Next, the *keccak_256* function (occupying 2.08%) calculates the hash value of the previously mentioned components. Finally, the transaction must be signed (*ecdsa_sign_digest* (79.29%)) by using the previously computed hash value.

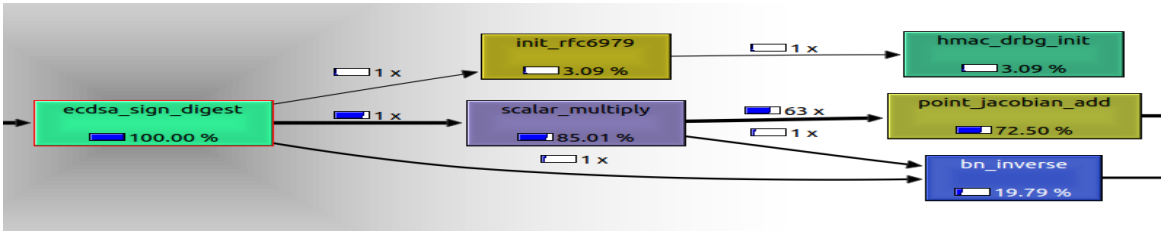


Figure 3.10: Call graph of Ethereum API, the *ecdsa_sign_digest* function is considered as the father function

In the figure 3.10 the *ecdsa_sign_digest* function is considered as the father function in order observe which other functions are called inside *ecdsa_sign_digest*.

It can be observed that the major part of the of *ecdsa_sign_digest* function is taken by the *scalar_multiply* function which takes a significant 85.01%, which is intended to do scalar point multiplication on the elliptic curve. The elliptic curve point multiplication is the core operation of ECDSA digital signature algorithm. It should be noted that this operation is computationally expensive and difficult to perform. The *init_rfc6979* function of *ecdsa_sign_digest* is used to create a nonce parameter required to the signature process. This

is an hmac algorithm using SHA-256 hash function.

In a related work of Pincheira *et al.*[5], the authors have proposed a C API that enables the Ethereum transaction creations and sending. In the experiments of the API, the authors measured the execution time of the API in different IoT architectures (Arduino Uno, STM32L031K6T6, STM32L452, etc.).

The ratio between the results of the experiments measured in [5] and the results of the profiler (call graph) proposed by this thesis work is almost identical. The two main differences between these works are that this thesis work contributes to the API's open-source code[11] to help IoT-Blockchain developer's work. The second difference is that this contribution contains a tool for double signature, which can be useful in IoT-Blockchain related use cases and applications. The double signature tool is explained later.

The authors of [5] have emphasized that the time required for the digital signature creation is significant, and it should be optimized to reduce the execution time. They also showed that in small architectures like an ATM device the signature process can take more than 4 seconds. This thesis work encourages the optimization of the signature process, which should preferably be done in hardware accelerators rather than in software. In subsection 5.3.4 (p.124) a SystemC-TLM module is proposed to model the hardware accelerator of the elliptic curve point multiplication (ECPM) operation of the digital signature process. Accelerating this operation makes sense because it takes 79.29% of the digital signature process.

It is important to note that the digital signature of the transaction is computed not on the transaction itself but on the hash value of the transaction, this operation is detailed later in section 4.3 (p.97). The hash value has a fixed size. Thus the digital signature is always performed on a fixed-length value, which means that the required time for its execution does not depend on the data size. In Ethereum, the hash value of the transaction is completed thanks to the Keccak cryptographic hash algorithm. The goal of any cryptographic hash algorithm is to realize a unique fixed-length hash value according to an arbitrary-length input. The previous result of the profiler is obtained according to small-size data (32 bytes). When the length of data increases, the Keccak hash function would take more time to compute the hash value.

Our previous works [17] [9] which also contributed to this thesis work by providing examples of Hyperledger Sawtooth API behavior when the data size to send to the blockchain increases. In the experiments, the execution time of the API was measured while running on a Raspberry Pi 3 B+. The results showed that the use of the hash function can be significant enough to be hardware accelerated. These results are in more detail later, in subsections 3.7.2 (p.72) and in section 3.8 (p.74).

In related work [4], the authors implemented a double signature tool in a C++ API. The double signature can be used in application-specific cases for increasing security. This feature allows signing the transaction's payload with a dedicated private key (for example, the IoT device owner's private key). This payload signature is done by the API (locally in the IoT device) using the ECDSA algorithm with the secp256k1 elliptic curve. This signature of

[11] Ethereum-web3-cpp project is available: https://github.com/KRolander/ethereum-web3-cpp

the payload is then verified in a dedicated smart contract. In the case of an invalid signature, the data sent to the smart contract cannot be stored. It is important to note that only the transaction's payload is signed locally in the IoT device. The API ABI encodes the payload's signature and the payload, which is concatenated with other transaction parameters forming an unsigned transaction (the unsigned transaction is not RLP encoded). This unsigned transaction is then sent to a local Ethereum node by using the Web3 command *sendTransaction*. Next, this local node signs the transaction and broadcast it to the network peers. The transaction is not signed by the IoT, which can be considered as a disadvantage because the blockchain cannot verify which transaction was signed by which IoT devices of the network.

Contrarily to this implementation, in this thesis contribution, we propose an implementation in which the transaction signature is also performed by the API locally in the IoT device (so it is not the local Ethereum node that signs). In our implementation, the payload and the transaction can be signed locally in the IoT device with two different private keys. This feature allows identifying the IoT device and better determine data provenance.

The following test analyzes the double signature implementation proposed in this thesis contribution. The data to send is 32 bytes long data (uint256) concatenated with a given identification number of the IoT device. The Keccak hash of this concatenated data is signed with the ECDSA algorithm using the secp256k1 elliptic curve. The payload of the transaction is composed of the 32 bytes data, the ID, and the signature (R, S components) of 64 bytes length. Next, the transaction is formed and digitally signed (ECDSA) with the IoT owner's private key. The call graph of the program execution is given below (figure 3.11).

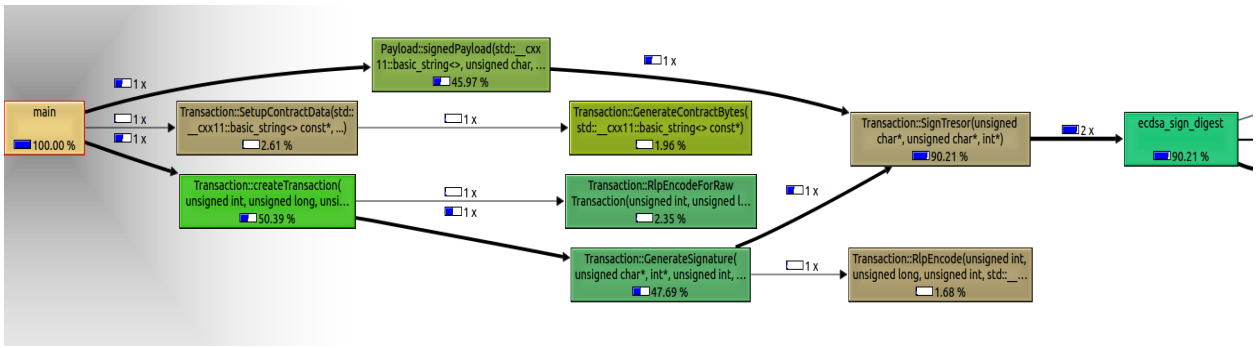


Figure 3.11: Call graph of Ethereum API when a double signature is performed, the *main* function is considered as the father function

It be observed on the results that the $SignTresor$ function is called twice, once for performing the payload signature and once for signing the transaction. In this case the signature creation takes more than the 90% of the total execution time, which also shows the importance of digital signature acceleration, eventually with a hardware module that can accelerate the elliptic curve point multiplication operation.

The following section discusses the C++ API implementation of Hyperledger Sawtooth blockchain.

3.7.1.2 Hyperledger Sawtooth

This API aims to enable interactions with transaction processors (smart contracts) of Hyperledger Sawtooth blockchain. The transaction processors can be written in different programming languages. In the experiments, Python programming language was used. Likewise Ethereum, Hyperledger Sawtooth does not provide an official C++ library for creating valid transactions.

Similarly, like Ethereum, Hyperledger Sawtooth also uses the hexadecimal representation (Hexa string) of the data it operates on. In transaction processors, actions are called and not necessarily the functions, contrarily to Ethereum smart contracts. The transaction's payload encoding (that would call an action) depends on how the transaction processor is implemented. By default, the transaction payloads are encoded by CBOR standard that uses "key:value" pairs that are binary encoded. The CBOR format is not explained here as RLP was explained previously because official C++ libraries exist for CBOR encoding. In the experiments of simple transaction sending, the *IntKey* transaction processor is used (officially provided by Hyperledger Sawtooth). This transaction processor can increment/decrement a variable with the value that is specified in the transaction. For doing so, the transaction's payload has to contain the name of the variable that would be modified, the action to do (increment or decrement), and the value which increments or decrements the variable in the transaction processor. The variable that can be incremented or decremented is stored in the state of the blockchain.

The data of a transaction processor is stored in the state dictionary, which uses addresses to access the data. When a transaction wants to modify the value of a given data, it has to specify the address of the variable, which is also used in the transaction processor. The address length must be a 70 character hexadecimal string (35 bytes). In the case of the *IntKey* transaction processor, the address of the variable which value can be modified is defined as the first 3 bytes of SHA-512 hash value of the transaction family name and the first 32 bytes of SHA-512 hash value of the name of the variable (it is the identical name that was specified in the payload).

It must be noted that the payload content and the address creation are implemented depending on how the transaction processor handles the addresses and the payload. The transaction processor decodes the payload to know which action has to be executed, and it creates the addresses to access the variables in the blockchain state. The address creation is done in the same manner on both sides, in the API and the transaction processor. This address is the input and output field in the transaction header (see figure 3.6, p.54). The addresses must be unique values thus the use of a hash algorithm is a requirement.

In figure 3.6, the batch and transaction content can also be seen. This structure is the requirement of Hyperledger Sawtooth (it is independent of the transaction processors), and the official implementation of batch and transaction structure is available in the form of a Google protocol buffer. This protocol buffer is a data structure that can be converted into several programming languages, like Python, JavaScript, and C++. The Google protocol buffer in C++ language corresponds to a C++ class. In C++, five classes are used to realize the batch and the transaction creations: *Batch*, *BatchHeader*, *BatchList*, *Transaction*, and *TransactionHeader*. The classes contain the same variable names as it is described in figure 3.6.

The following figure 3.12 shows the results of the call graph when the data size is 32

bytes. The default *IntKey* transaction processor was also used, which allows sending a 32 bits value which can increment or decrement the state value. With a slight modification, the implementation allows setting a value of 32 bytes to be able to visualize the differences between Hyperledger Sawtooth's and Ethereum's APIs. In this analysis, the 99.32% of the main function is occupied by the *build_signature* function. This function is then chosen as "father" of the other functions (it takes 100% of the execution time of the API). Only functions taking more than 1% of time occupations are represented.

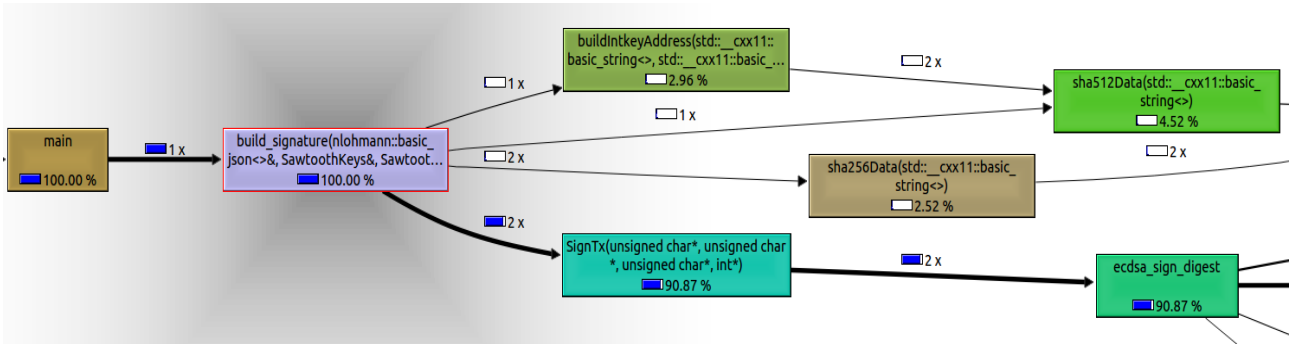


Figure 3.12: Call graph of Hyperledger Sawtooth API, the *build_signature* function is considered as the father function

The result shows that the digital signature creation (*SignTx*) takes more than 90% of the total time, which is evident because the API must compute two signature processes, the signature of the *BatchHeader* and the signature of the *TransactionHeader*. The signature is done by ECDSA algorithm using the secp256k1 elliptic curve, likewise in Ethereum. This optimized API is contributed in open-source[12] and uses the trezor-crypto library for computing ECDSA sign process (same in Ethereum API).

The *sha512Data* SHA-512 cryptographic hash function uses 4.52% of the total execution time, this function is called to create the hash of the payload. This hash value is called *payload_512* in the *TransactionHeader* (see figure 3.6, p.54). This hash function is also called for creating the address encoding, which allows access to the state variables declared in the transaction processors. The *sha256Data* hash function is used to hash the *TransactionHeader* and the *BatchHeader*. These hash values are required for the signature process, because signatures are done on the hashes of the *TransactionHeader* and the *BatchHeader*. It can be noticed that Hyperledger Sawtooth does not use the Keccak-256 hash algorithm, as is the case in Ethereum. The two SHA hash functions above are implemented in Crypto++[13] free and open-source C++ library. The size of the *BatchHeader* and *TransactionHeader* is quasi invariant, so the time required by the *sha256Data* function does not vary significantly. The SHA-256 function is more solicited when the number out/input variables of the transaction processor increase, which makes sense because, in this case, more addresses have to be generated.

The *sha512Data* function's computation time depends on the payload size, so the *sha512Data* requires more time to compute when the data size (payload) increases. Experimental results

[12]The Hyperledger Sawtooth optimized API project is available: https://github.com/KRolander/HyperledgerSawtooth-cpp-client-optimized The basic project can be found at: https://github.com/lucgerrits/HyperledgerSawtooth-cpp-client

[13]Crypto++ is available at: https://www.cryptopp.com/

about the importance of hash function computing are given in subsection 3.7.2 (p.72) and section 3.8 (p.74).

It should be noted that the payload creation is faster in Hyperledger Sawtooth than in Ethereum thanks to the CBOR encoding. It can also be noted that the transaction/batch creation is easier to do in Hyperleger Sawtooth, thanks to Google protocol buffers, which provide C++ classes to fill with the reliable data (see figure 3.6).

3.7.1.3 EOS.IO

Unlike in Hyperledger Sawtooth or Ethereum, in EOS.IO the transaction structure is given in JSON format that must be filled with values according to the keys. EOS.IO provides default functions to convert JSON format to ABI and ABI to Hexa string representation (serialized representation). The actions (functions) in JSON format can be converted to ABI and from ABI to a hexa string representation, which can be read and executed by the smart contract. The converted field is also called the data field of the actions. For example, the *hi* function can take an input that would be printed out. This input is the data field of the action *hi*.

The main drawback of EOS.IO is that in transaction creation, some types of information are required from the blockchain, such as the block number of the last blocks (*ref_block_num*) and the identifier of the block (*ref_block_prefix*) that is referred to *ref_block_num*. These pieces of information can be retrieved by calling the blockchain. At least two request messages have to be sent to the blockchain (*/get_info* and */get_block* requests).

The smart contract ABI code must also be known for transaction creation. ”The Application Binary Interface (ABI) is a JSON-based description on how to convert user actions between their JSON and Binary representations[14]”. This information can also be requested from the blockchain (*/get_abi* request). However, it can also be hardcoded in the given IoT device. The hardencoding avoids the request sending to the blockchain.

Previously it was mentioned that action calls (function) of the given smart contract can be performed by completing the ABI code of the smart contract with the inputs. The smart contract’s action contains JSON *key:value* pairs in which the value has to be used as the input.

The raw data is given in a JSON format that is ABI encoded and represented as a Hexa string. Under the actions parameter of the transaction structure, the data field should be filled with this ABI encoded Hexa string data value.

Likewise, Ehtereum and Hyperledger Sawtooth, EOS.IO uses ECDSA with secp256k1 curve to perform the transaction’s digital signature. In the API the transaction is compressed, and it is the compressed transaction that must be signed. The compression occurs when the transaction already contains all the information about the block number of the last blocks (*ref_block_num*), the block’s identifier (*ref_block_prefix*), and the ABI encoded Hexa string converted data of the action that the smart contract should execute. The uncompressed transaction is a JSON format structure. The compressed transaction is performed in the same way as the data encoding (ABI and Hexa string) for the smart contract.

Before signing the compressed transaction, it is concatenated with the chain’s ID and with 32 zeros because the transaction contains context-free data. This concatenated data is hashed, which is performed by SHA-256 cryptographic hash algorithm.

[14]Cited from: https://developers.eos.io/welcome/latest/smart-contract-guides/understanding-ABI-files

This hash is then signed thanks to the ECDSA primitive with secp256k1 curve. It should be noted that the signature must be canonical. A canonical signature is more resistant against a possible replay attack. However, it is possible that the signature process has to be repeated until obtaining the canonical signature (in a loop, a nonce data value is incremented). The final recover id is equal to the magic number 31 of EOS.IO plus the recovery id determined during the signature operation.

The signature is concatenated with ”K1” string. This value is hashed by the RIPEMD160 (160 bits length hash) cryptographic hash function.

The final signature of the transaction that blockchain would verify intrinsically is composed of the signature concatenated with a string *”SIG_K1_”* and the last 4 bytes of the checksum. It should also be noted that the signature and the checksum’s last bytes are base58 encoded Hexa strings.

In the experiments, the helloworld default smart contract is used, which enables to print out the input of the function ”hi”. The function prints a 32 bytes long input (same length data as in the experiments of Ethereum and Hyperledger Sawtooth). In the previous experiments, the latency spent on the communication was not taken into account. In the case of EOS.IO two requests are necessary to create a valid transaction (*ref_block_prefix* and *ref_block_num*). In the experiments, the values required to be requested are set as default values to avoid communication and to be able to compare the EOS.IO API’s behavior with the Ethereum’s and Hyperledger Sawtooth’s.

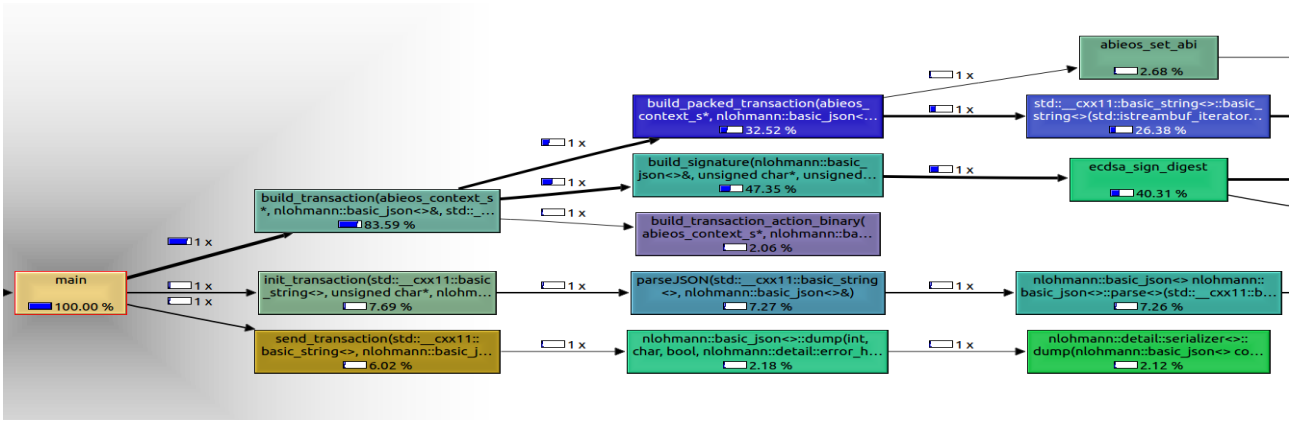


Figure 3.13: Call graph of EOS.IO API, the *main* function is considered as the father function

The following figure 3.13 represents the call-graph of EOS.IO API. It can be noted that the digital signature (*ecdsa_sign_digest)* plays an important part in the executions, such is the case in Ethereum and Hyperledger Sawtooth. It should also be noted that the *basic_string* function also takes an important part of the execution time, it is due to the construction of the packed transaction structure. This JSON structure is stored in a file that has to be read by the API. In this graph, the SHA-256 hash algorithm does not appear because the payload with the transaction structure does not take a significant size and other functions are more solicited than the SHA-256.

3.7.1.4 Substrate

The Substrate blockchain framework is entirely modular, it can be configured as the developer wants. This framework also provides already developed modules that can be considered as puzzle pieces. It has already been mentioned that the preexisting modules use certain cryptographic patterns. Substrate allows using the ECDSA digital signature scheme with secp256k1 curve (case in the blockchains described previously). However, Substrate by default uses the Ed25519 signature algorithm [99], which is a version of the EdDSA (Edwards-curve Digital Signature Algorithm). This signature uses a twisted Edwards curve (edwards25519) that is birationally equivalent to the curve Curve25519 [100].

Substrate also allows using sr25519 signature algorithm based on a Shnorr signature scheme [101], as is the case in Ed25519. Substrate implements this algorithm to ease the connection to the Polkadot blockchain [102]. The goal of Polkadot is to connect different blockchains together and facilitate communication between them. This blockchain would probably solve the problem of interoperability of different blockchain that aims to communicate together. It would probably be used in new ecosystems in which different entities own their specific blockchain, and they must be connected on a joint blockchain.

Substrate by default uses the BLAKE2b hash algorithm. However, other hash algorithms like SHA-256 and SHA-512 can also be used.

3.7.1.5 Conclusion

In these previous parts of the section, the proposed APIs have been analyzed to determine their behaviors and to identify which functions could be optimized. It could be resumed that the digital signature process takes the most of the execution time when the size of the data to be sent is small. It was also determined that the signature process could be optimized by accelerating the elliptic curve point multiplication. A significant acceleration can be achieved by a using specific hardware IP (Intellectual Property).

It has also been learned that several hash algorithm functions could be called more frequently according to the data size of the payload. The following sections would show how the data size causes a more significant use of hash functions. Moreover, these results would also demonstrate the importance of the need for hash hardware accelerators in the proposed IoT architecture.

It is worth noting that the previously presented APIs can create valid transactions according to the requirements of the given blockchain, and most of these APIs are well optimized at the software level. It can also be resumed that Ethereum and Hyperledger Sawtooth APIs could be sophisticated enough to be used in IoT-Blockchain network architectures. However, EOS.IO API requires to perform at least two communications ($getref\_block\_num$ and ref_block_prefix) with the blockchain before it can create a valid transaction. The communication of IoT devices is a crucial operation, and it has to be limited to obtain a better performance. Because EOS.IO has to do two additive communications to the blockchain to perform a valid transaction sending, using EOS.IO in blockchain-IoT systems is discouraged.

3.7.2 Car accident use-case

3.7.2.1 Description

The car accident use case takes part in the Smart IoT for Mobility project, and the main idea was imagined by vehicle manufacturer Renault, the industrial actor of the project. The idea is that the car manufacturer, insurance company, car repair company, the police form an ecosystem based on blockchain technology. Cars are the mobile members of the ecosystem and they are also connected to a blockchain. When an accident occurs, the car sends its data to the blockchain about the accident recorded just before the accident happened.

The vehicle's sensors providing the recorded data could be sensors such as lidars, radars, odometers, wheel pressure sensors, but also cameras that can record the car's environment and the driver's behavior. In order to record sensors data and create valid blockchain transactions an IoT device could be embedded into the vehicle.

The data stored on the blockchain is permanent, and it could be used by a dedicated smart contract of an insurance company, for example. With the help of this smart contract, the vulnerable party of the accident could be found more straightforward, and the refund procedure could be done faster.

Figure 3.14 depicts an example of the payload that can be sent when an accident occurs. This payload was used in the experiments of latency measurement of the API in previous works [17] and [9].

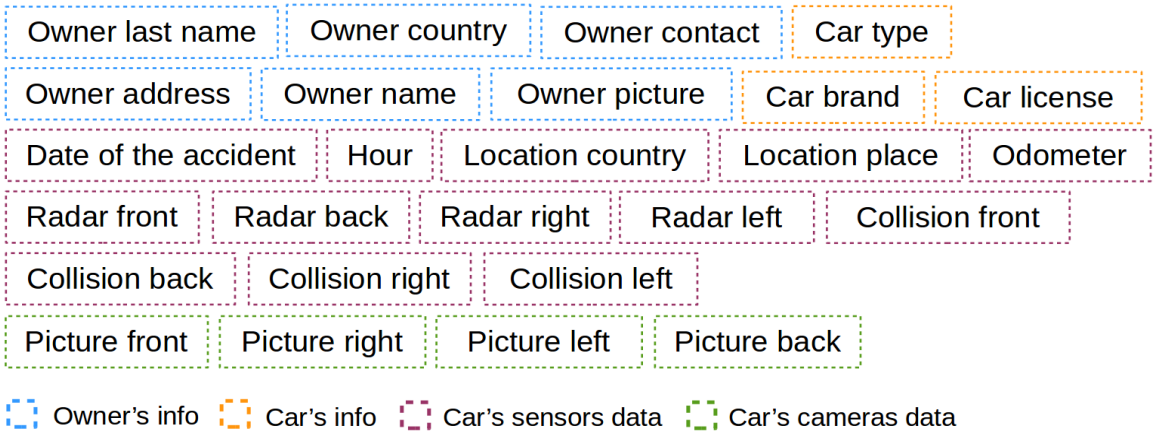


Figure 3.14: Example of the transaction payload that sent when an accident occurs

The payload contains diverse information about the car, and its owner. The vehicle's sensors provide detailed information about the accident, and camera pictures contribute about the car's environment. The report of Y. Khacef [91] provides a well-detailed list of sensors which are present in today's vehicles. The study also analyzes the size of the data recorded 10 seconds before the crash occurred using a buffer system. The events recorded 10 seconds before the accident occurred can open new research and analysis initiatives for experts and insurance companies to determine the faulty party of the accident. The report also contains a list of future automotive sensors, such as alcohol consumption and awake detection.

3.7.2.2 Importance of transaction size to send to the blockchain

Previously, it was mentioned that the data that are recorded by the vehicle is various, the data could be provided by radars and also by cameras. It should be noted that these different data can grow to a significant size. In a previous work [17] that provided an essential contribution for this thesis work, a Hyperledger Sawtooth blockchain was combined with an IoT network representing the vehicles. This Hyperledger Sawtooth blockchain network contained five validator nodes that are run on X86_64 laptop architectures. Raspberry Pi 3 B+ models sent data to the blockchain. The experiment consisted of measuring the API's execution time and the power consumption of the Raspberry Pi according to the increasing size of the data to be sent. It should be noted that these experiments included the time taken for the communication. The valid transaction were sent to the blockchain using the HTTP communication protocol.

In the first experiments, the size of the data varied from 1 MByte to 41 MBytes. Table 3.3 shows the total execution of the API and the percentage taken by the SHA-512 cryptographic hash function (this function computes the hash of the transaction payload). The average time that is required to compute one hash value is also given. The input of any hash function can have an arbitrary size.

The hash function produces chunks of the input. After a chunk is hashed, this hash value is reused to create the hash of the next chunk. Creating one SHA-512 in the table means the hash creation according to a chunk. This process is also called the core function of the given hash function. More detailed description is given on page 99 about how the previously seen cryptographic hash functions are computed exactly.

Data Size	Total exec. time	Time occupation of total time by SHA-512	Time of creation of one SHA-512 Hash
1 MByte	4.495 s	3.46 %	9.382 μs
2 MBytes	8.76 s	3.49 %	9.433 μs
41 MBytes	161.931 s	4.04 %	10.21 μs
Average :		3.66 %	9.675 μs

Table 3.3: Summarized execution time is reported as an average of 10 executions. The table with results is adapted from [17]

The results show that the hash creation procedure takes an average of 3.66% of the execution time even if data size increases. It is because the data that is sent to the blockchain has to be CBOR encoded (JSON to CBOR format conversion) and Hexa string converted, and also because when data size increases, the time due to the communication (data in the air) also increases. It can be summarized that in this context, all of the operations: encoding/conversion, communication, and hash creation are proportional to the increasing data size.

This work also mentioned that the SHA-256 function is used in the ECDSA signature process (nonce creation with hmac cryptographic patterns). Even if the use of the SHA-256 function is less significant than SHA-512, this hash creation can also be optimized by a hardware accelerator. Table 3.4 shows the speedup that can be achieved using a dedicated hardware accelerator [103] for SHA-256 and SHA-512 algorithms.

	Process time CPU	Process time ASIC	Gain
SHA-512	9832 ns	32 ns	**293.18**
SHA-256	2760 ns	24.48 ns	**112.75**

Table 3.4: Process time and gain, the ASIC estimated for 40 nm CMOS technology. The table with results is adapted from [17]

The API was executed on a Raspberry PI 3 B+ model in order to measure the APIs execution time. The SHA hardware accelerator and the Raspberry are implemented with different CMOS technology (accordingly 130 nm and 35 nm). In order to be able to calculate the gain that can be achieved using the SHA hardware accelerators, an advanced CMOS scaling method [104] can be used to estimate the execution time if the hardware accelerator were implemented with the same or near to the same CMOS technology.

3.7.2.3 Conclusion

It can be concluded that the proposed Hyperledger Sawtooth API's functions occupy a time in proportion with the increasing size of data. It can also be noted that the high and raw data sending to the blockchain is probably a naive solution because of two reasons. First, the increased data size requires more computation for transaction creation in the totality of the API. It should also be noted that the choice of the communication protocol can also play an important rule in the energy consumption of IoT devices. However, the choice of the communication protocol also affects the blockchain-IoT architecture (gateways are required for example).

Secondly, oversized transactions can make the blockchain's overall data growing too fast. Today Bitcoin's size is more than 333 GBytes [33]. Its data size grows since 2009, with an average transaction size of 500 bytes per transaction. Megabytes size transactions could "explode" the blockchain size.

In general, IoT applications are the source of a significant amount of data, and many applications send large amounts of data per payload. The following section proposes a solution for handling the quickly growing data size of blockchain but still allowing the inclusion of IoT devices and IoT applications.

3.8 Optimize the quickly growing data size in blockchains

In blockchain-IoT realistic industrial use cases, there is always a risk if the physical infrastructure can meet or not the use case requirements. Previously, it was mentioned that the quickly growing data size of the blockchain could be an essential issue in IoT-Blockchain applications. In certain cases, the infrastructure behind the blockchain network cannot handle the too quickly growing data size, which can cause a total crash of the overall system. On the other hand, creating/sending large transactions is also not optimal according to the latency and the energy consumption of IoT devices. In the following sections, the proposed solution can solve the quickly growing data size of the blockchain by combining it with the InterPlanetary File System (IPFS).

3.8.1 Introduction and challenges

Several studies can be found on blockchain structures in which IPFS distributed ledger technology is applied. The idea behind these structures is to store only the hash of the raw data on the blockchain and store the raw data in IPFS. The combination of blockchain with IPFS is used in data sharing with fine-grained access control [105], in healthcare applications to store medical data, there is also an use case in which COVID-19 digital medical passports containing vaccination and immunization records of the given person are stored in the IPFS and their corresponding hash in Ethereum blockchain [106].

Using data hash is not a new mechanism. The hash of the raw data is a unique fixed-size representation of the data. If the raw data were manipulated before it were recorded, the hash of these data will not correspond to the hash stored on the blockchain. Thus, data manipulation can be easily verified. Storing only the hash of raw data decreases the data size growth of the given blockchain, as the hash data size is fixed (256 - 512 bits in general). In certain blockchain-IoT architectures, the hash storing method was already implemented. However, the raw data are stored on a local database or cloud storage. This method can be reviewed because using a local database or cloud storage makes the system partially centralized, which contradicts the idea of decentralized data storage.

In IPFS, the data (or file) can be accessed according to the hash of the data, which can be considered as a pointer to the data. Applying IPFS for data storing meets the basic requirements of decentralized data storing systems because IPFS is a decentralized ledger technology. However, combining IPFS with blockchain technology can provide a decentralized system and a more secure, transparent, and partially immutable data storage.

IPFS

IPFS is a peer-to-peer network in which, likewise in BitTorrent file system, the shared data can be copied from every network member [107]. The data storage is based on content-addressed hyperlinks forming a Merkle DAG. The hash of the data serves as the pointer to the data stored in IPFS.

The main differences between the blockchain technology and the IPFS system are that in IPFS, not every peer is obliged to contain the same data, and IPFS does not contain a consensus rule. Because there is no consensus rule, every peer can add data to the system without control. In a blockchain-IPFS system, controlling the writing of data to the IPFS system by the blockchain is an essential operation. Without this control, malicious participants can fill both systems with unnecessary data.

In related work [108], the authors proposed a consortium network that contains validators of multiple sidechains. The sidechains are considered as separated groups. IoT devices can take part in one of the specific groups, and they can send their data to the given sidechain. Dedicated smart contracts are implemented in each sidechain to verify if it takes part in the sidechain (in the group). The IoT device's public key is used for the verification process. When an authorized IoT device sends its data to the Validator node, it updates it to the IPFS system. The hash of the data stored in the sidechain, and the Validator makes a log about the IoT device data sending.

In another related work [109] the authors developed middleware between the IoT devices Ethereum and IPFS network. In this architecture, the devices can be registered to the mid-

dleware. When an IoT device sends data to middleware, the middleware stores its data in the IPFS network and the hash of the data in the blockchain. The IoT client applications can listen to the Ethereum smart contracts notification about the success of the hash storage. When the hash is stored, the IoT Client can access the data stored in IPFS using the hash stored in the blockchain.

This system allows storing IoT data in a distributed manner automatically thanks to the middleware. Unlike the implementation proposed by this thesis work, IoT devices are not identified, and the blockchain smart contract does not control data storage.

The proposed implementation of this thesis work controls which device takes part in the ecosystem and is authorized to store data. The IoT device sends the raw data to the IPFS system and the hash of the data to the blockchain.

The data sent to the IPFS system remains in a queue until the dedicated business logic (smart contract), and particular module implemented in the Validator node of Hyperledger Sawtooth blockchain notifies IPFS about the authorization (see page 76 for more details). Controlling the data storage on the IPFS makes the system more secure and filters malicious members storing any types of data. This last control feature can be considered as an improvement comparing to [109].

3.8.2 Proposed solution by merging Hyperledger Sawtooth with IPFS distributed ledger technology and IoT

The proposed architecture was used in the car accident use case in which every IoT devices embedded in cars have their private/public key pair for identification in the blockchain. Hyperledger Sawtooth was used as a blockchain that is combined with a private IPFS network. In the private IPFS network, every peer has the same swarm (secret) key, which allows participation in the network. The complete architecture is depicted in figure 3.15. This architecture was also contributed in one of our previous works [9].

It should be specified that this architecture was imagined to be implemented in industrial infrastructure. The IoT devices do not send the data directly to one of the blockchain or IPFS nodes but through a *Load Balancer* entity. This unit contains an IP address that the IoT devices can target. When a message arrives in the *Load Balancer*, it forwards the message to one of the nodes (a round-robin algorithm is used).

The Hyperledger Sawtooth blockchain is highly modular. Thanks to this feature in every Validator node, an *IPFS module* is added. This module is connected to the transaction processor (smart contract), and it can also communicate with the transaction processor using Python sockets. This *IPFS module* then can send notifications (in fact, any types of data) to the *IPFS REST API* that was also added to the top of the *IPFS Deamon*. According to the message from *IPFS module*, the data waiting in the IPFS queue would be added to IPFS or rejected.

One of the main benefits of Hyperledger Sawtooth's modularity is that the transaction processor can communicate securely inside the Validator node with an additional module that can communicate externally (interest of *IPFS module*).

In the case of Ethereum, it is not possible to add such a module communicating externally. However, the smart contract can create a so-called "Ethererum notification" that can contain

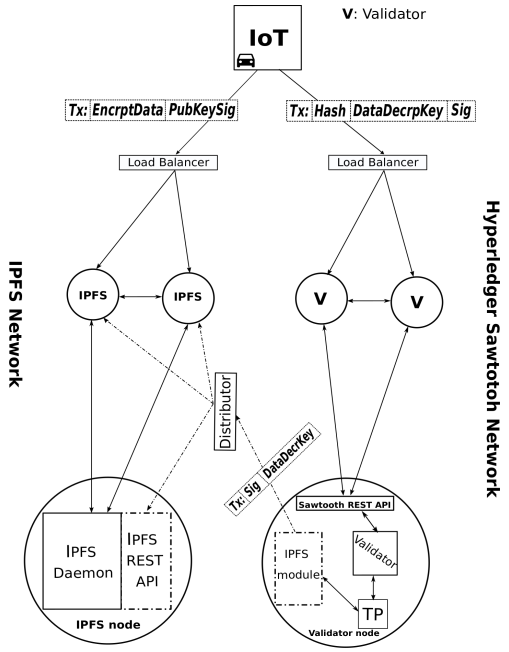


Figure 3.15: Implementation containing Hyperledger Sawtooth and IPFS.

some data. External applications can subscribe to these notifications, and by listening to the blockchain logs, the notification can be found.

With the proposed protocol, IoT device data storing on the IPFS and the blockchain is authorized if the device takes part in the ecosystem. Thus, the protocol can identify each device, and it also controls malicious data storing on the IPFS system. The protocol is depicted in figure 3.16.

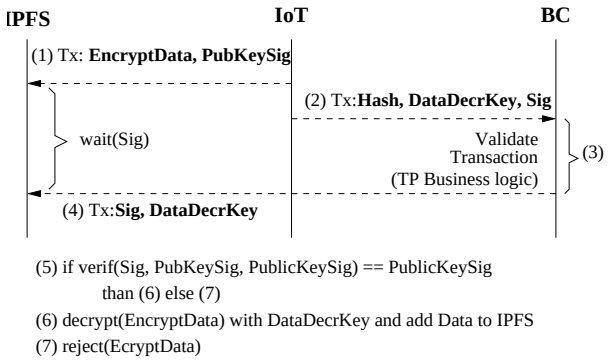


Figure 3.16: Message sending protocol in the proposed architecture. The figure is retrieved from [9]

The IoT device generates a public/private key pair (*PubKeySig.* / *PrivKeySig.*). The next step is to create a signature with the *PrivKeySig.* on a message to be signed. The message can be any data, but in this case, the public key was chosen as a message to be signed (*PubKeySig.*). The signature (*Sig.*) generation is described by the equation below (equation 3.6).

$$Sig. = sign(PrivKeySig., PubKeySig., PubKeySig.) \tag{3.6}$$

This signature is needed to identify the IoT device not only to the blockchain but to the ecosystem. When the signature is done, the IoT device encrypts the raw data with a

symmetric cryptographic encryption pattern. For symmetric encryption, a secret key is used that is called *DataDecrKey*. The ciphertext (encoded data) is called *EncrpData*. The choice of data encryption pattern is out of the scope of this work; for example, AES [110] algorithm could be used. The data has to be UnixFS formatted to be stored on the IPFS system. UnixFS protocol buffer can be used to achieve this UnixFS format. It must be noted that the standard UnixFS protocol buffer is not up to date. After a slight modifications proposed by this work, the protocol buffer is operational.

Now, the encrypted data (*EncrpData*) and the public key (*PubKeySig.*) are sent to the *IPFS REST API* (step (1)) of the protocol. These values are queued, and *IPFS REST API* waits for the signature (*Sig.*) that would be sent from the blockchain's *IPFS module*.

After the encrypted data is sent to the *IPFS REST API*, the data has to be hashed before sending it to the blockchain. Several types of cryptographic hash algorithms can be applied in IPFS networks, such as SHA-256, SHA-512, Keccak-256, or Blake-2. The SHA-256 is used as the default cryptographic algorithm. The hash of the data serves as the pointer to the data in IPFS.

The hash (Hash), the secret key (*DataDecrKey.*) for decrypting the encrypted data (*EncrpData*), and the signature (*Sig.*) are set to the transaction's payload. The transaction and the batches are structured and signed according to the requirements of the Hyperledger Sawtooth blockchain (see p.50 and p.67). In this particular vehicle use case, the batch is signed with the car owner's private key, and the IoT device's private key signs the transaction.

The batch contacting the transaction with the payload of the triplet *Hash*, *DataDecrKey.* and *Sig.* is sent to the blockchain (step (2) of the protocol). Step (3) of the protocol consists of the security level to pass is the verification of the batch and the transaction validity. It is an intrinsic operation of the Hyperledger Sawtooth blockchain. The second verification process is due to the transaction processor (smart contract), which describes a dedicated business logic. The transaction processor verifies if the vehicle has already been registered in the blockchain using its public key. The second step is to verify if the owner has already been registered and he/she corresponds to the car. If these two steps are not valid, the transaction is rejected. When the transaction meets the validity requirements, the transaction processor parses the payload and saves the hash. The signature (*Sig.*) and the decryption key of the data (*DataDecrKey.*) are forwarded to the *IPFS module* that sends these two values to the IPFS network using WebSocket (step (4) of the protocol).

Step (5) of the protocol: *IPFS REST API* verifies the signature sent by the *IPFS module*.

The IoT device signed *PubKeySig.* by using *PrivKeySig.*. *PubKeySig.* was already sent to the IPFS REST API in step (1) of the protocol, so the signature (*Sig.*) can be verified by *PubKeySig.*. If the signature is valid, the step (6) of the protocol follows. The encrypted data (*EncryptData*) can be decrypted by the secret key (*DataDecrKey*), and the decrypted data can now be added to IPFS. An invalid signature means that a malicious entity sent data to the IPFS system intending to spam. The data according to an invalid signature will be rejected (step (7) of the protocol).

A five-node network was implemented in the experiments in an Intel Xeon CPU D-1528 @ 1.90GHz server, with 12 CPUs and 128.8 GB RAM. The results showed that the IPFS module does not influence the transaction validation time significantly. The presence of the module increases the transaction execution time by approximately 10-100ms. The transaction validation rate is 2.7 transactions per second.

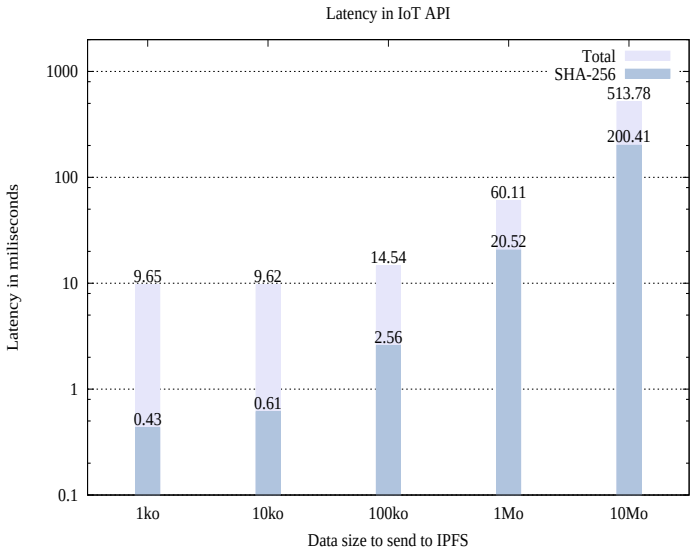


Figure 3.17: Latency in IoT device. This figure represents the total execution time and the time that is occupied by the hash creation. Figure retrieved from [9]

Another experiment measured the latency of the transaction creation in an IoT device. The IoT device of the experiment was a Raspberry Pi 3 B+ model. It must be noted that the latency due to the communication was not measured as it is dependent according to the communication protocol. In the testing phase of the overall architecture, the HTTP protocol was used. In the experiments, the size of the data to store on the IPFS system increases from 1k octets to 10M octets. Figure 3.17 depicts the total latency of the API execution time and the time taken by the SHA-256 hash algorithm required for IPFS hash creation (SHA-256 is the default hash algorithm used by IPFS). It can be observed that when the size of data increases, the time taken by the hash algorithm is more significant. In the case of 1M octet size data (corresponds to an image about the vehicle's environment, for example), the hash calculation takes 34.1% of the total execution time of the API.

3.8.2.1 Conclusion

It can be concluded that thanks to the proposed architecture, the blockchain contains only the hash of the data and some other pieces of information that can control the data storage in the ecosystem. The data to be stored on the blockchain is less than 96 bytes in size. This size is much less than the megabytes of data to be stored per transaction.

This architecture handles the quickly growing data size of the blockchain. In addition to this feature, it also controls the data storing securely and limits data storing only for ecosystem members.

In similar IoT-blockchain use cases, in which the size of the data can be significant, and the data must be decentralized, it is essential to use a DLTs such as IPFS. It is also important to note that in the architectures combined with blockchain-IoT and IPFS the hash creation operation can be optimized using dedicated hardware accelerators for the given hash algorithm. In the proposed model of IoT architecture (detailed later), modules for hash creation are added.

3.9 Conclusion

This chapter demonstrated three blockchains (Ethereum, Hyperledger Sawtooth, and EOS.IO) integrated into IoT network architectures. Substrate blockchain framework is not yet operational, however, it is also a promising candidate for future IoT-blockchain applications.

Another objective of this chapter was to determine the basic requirements of the three studied blockchains to establish interactions between the IoT devices and the given blockchain. In this thesis work, APIs were implemented in C++ language to enable these interactions. All of these APIs are contributed in open-source codes.

The Callgrind profiler analyzed the APIs to obtain a view of the program's most called and used functions. Studies were also done on the APIs behavior when the data size to send to the blockchain is limited and also when it is important. The API has also been analysed when using it in an IoT-Blockchain-IPFS network structure. According to the profiler's analysis and the measurements of the physical device (Raspberry Pi) running the API, it can be concluded that the digital signature process and the hash operation should be be optimized in terms of execution time and energy consumption. It can also be noted that the majority of the studied blockchains uses the ECDSA algorithm with secp256k1 curve to perform the digital signatures of transactions. Using ECDSA with secp256k1 curve can be considered as a Bitcoin heritage as it is used in Hyperledger Sawtooth, Ethereum, and also in EOS.IO. Substrate framework implements EdDSA and other Schnorr-like signatures because these signatures can be computed faster, and they can be more resilient to several attacks.

Related work [5] analyzed embedded devices interaction with Ethereum blockchain, and also mentioned that the digital signature process should be done more optimally. In the blockchain-IPFS combination, the hash process becomes significant when the data increases. This fact also highlights that the hash creation in IoT-blockchain application should also be optimized.

In use cases in which the data size matters, it is essential to store only the hash of the data in the blockchain. According to these observations, this thesis work encourages optimizing the signature and the hash creation process. Several hash algorithms are used in the studied blockchains such as SHA-256, SHA-512, Keccak-256 and Blake2b.

One of the main objectives of this thesis work is developing an IoT hardware model that can interact with different blockchains in an energy-optimized way. This work proposes in Chapter 5 an IoT hardware model in which the hardware designs accelerating the signature and hash creation for different blockchains are implemented in SystemC-TLM language.

Chapter 4, in addition to providing a useful background of the cryptographic primitives identified previously (hash and signature), also lists the existing hardware accelerator designs found in the literature. In addition to listing the designs, the chapter represents the selected designs that were implemented in the proposed IoT architecture.

The power management of this model is also taken into account by using the PwClkARCH library and Linux power management tool.

Tab 3.5 illustrates the brief conclusion of this chapter. The tab includes the blockchain and DLT technologies that were studied and analyzed, the C++ API implementations that were proposed by this thesis and retrieved from related works, hardware acceleration possibilities for the identified cryptographic primitives, and finally the utility of the given blockchain in the Smart IoT for Mobility project.

Studied BCs & DLTs	Ethereum	Hyperledger Sawtooth	IOTA	EOS.IO	Substrate
Available SC	✓	✓	✗	✓	✓
Available API (C++)	✓ *	✓ *	✓	✓ **	✗
Hardware acceleration	- SHA-256 - Keccak-256 - ECC Point Mult → secp256k1	- SHA-256 - SHA-512 - ECC Point Mult → secp256k1	- Curl Hash	- SHA-256 - ECC Point Mult → secp256k1	- BLAKE2b - ECC Point Mult → Ed25519
Useful for SIM ?	✓	✓	✗	✓	↗

Table 3.5: Conclusion of Chapter 2. BC: Blockchain, DLT: Decentralized Ledger Technology, SC: Smart Contract, SIM: Smart IoT for Mobility project, * proposed in this contribution, ** proposed in related work, ↗ promising candidate

The following chapter (chapter 4) is about the fundamental cryptographic primitives in blockchain and IoT-blockchain structures. These primitives are essential to secure communication between the IoT device and the blockchain and the blockchain alone. The chapter also provides interesting details for better understanding how elliptic cryptography and hash algorithms work. Moreover, the chapter lists the existing hardware accelerators for cryptographic primitives, which were identified previously. It also gives the criteria for choosing the hardware accelerators that were added to the IoT architecture model. The chosen hardware accelerators are modeled in SystemC TLM and used in the final proposed IoT architecture model, described in chapter 5.

Chapter 4

Fundamental cryptographic primitives in Blockchain and IoT-Blockchain structures

4.1 Introduction

Chapter 3 identified and introduced the cryptographic primitives that are essential for IoT devices' interactions with the given blockchain. Hash and digital signature primitives are necessary to be able to create valid transactions and establish interactions between an IoT device and the blockchain. Section 3.7 (p.60) analyzed the proposed APIs that allow performing valid transactions. The analysis results show that the digital signature operation and the hash value creation can take a significant amount the APIs' total execution time.

This chapter describes the elliptic curve signature algorithms, that must be performed in the IoT devices. The Elliptic Curve Digital Signature Algorithm (ECDSA), Edwards-curve Digital Signature Algorithm (EdDSA), and the Schnorr-like signature algorithms are described because these are the signature algorithms used in the studied blockchains. Moreover, it can be noticed that most of the blockchains related to the top-30 mainstream cryptocurrencies (including Ethereum and Bitcoin) also implement these algorithms [111]. Twenty-one of the top-30 blockchains implement ECDSA, six uses EdDSA, and one of the 30 uses a Schnorr-like signature.

The chapter also gives a brief description of the hash algorithms identified in section 3.7 (p.60), SHA-256, SHA-512, Keccak-256, and BLAKE2b. It can also be noted that twenty-four of the top-30 uses the SHA-256 algorithm. The chapter also studies the basic operation of the cryptographic hash algorithm and its utility (especially in the digital signature algorithms).

It should be noted that the details that are given in this chapter about the cryptographic primitives do not go into the mathematical and security details. The objective is to understand how these primitives work. In this chapter, the hardware implementations of these primitives are also described. The presented hardware designs are retrieved from related research works. These designs are mostly intended for accelerating the totality or a part of the primitives. Some of the designs are available as open-source code (VHDL/Verilog language). These codes can be implemented on FPGA boards or synthesized to obtain an ASIC.

4.2 Elliptic-Curve Cryptography and Digital Signatures

This section describes the Elliptic Curve Digital Signature schemes that are widely used in blockchain and distributed ledger technologies. This section also gives an essential but simplified mathematical background of these schemes. The signature that has to be done in the IoT device and the verification required on the blockchain side are also discussed.

4.2.1 Introduction: "Classical" Discrete Logarithm Problem

The modern cryptographic schemes are based on the discrete logarithm problem. It is used in the digital signature and verification primitives (RSA, ECDSA, for example) and also in Diffie-Hellman Key Exchange (DHKE) methods [112]. The Generalized Discrete Logarithm Problem (GDLP) can be described as follows (and it is adopted from [14]):

G is a finite cyclic group with an operation $\circ$ and a cardinality n. Given the primitive element or also called generator $\alpha \in G$, and another element $\beta \in G$, the discrete logarithm problem consist of finding the integer x, where $1 \leq x \leq n$ in such a way that:

$$\beta = \alpha \circ \alpha \circ \cdots \circ \alpha = \alpha^x \tag{4.1}$$

A generator α is an element that can generate all of the elements of the cyclic group G by repeating the operation $\circ$. In the case of Discrete Logarithm Problem (DLP) the cyclic group is equal to $\mathbb{Z}_p^*$ of order $p-1$ ($n = p-1$) and x is an integer. The equation of DLP can be described as:

$$\beta \equiv \alpha^x \pmod{p} \tag{4.2}$$

It can be noted that the operation $\circ$ is a multiplication. The integer x is the discrete logarithm of β, and it can be defined as:

$$x = \log_\alpha \beta \pmod{p} \tag{4.3}$$

Finding the integer x when the generator α and the other integer β are known for a high p value becomes a challenging task. This property is essential in modern cryptography. For example in the case of DHKE [112]. Alice wants to exchange a session key S_k with Bob. Alice choses an integer $x \in \{1, \ldots, p-1\}$, x is actually the private key of Alice. The public key (β) of Alice can be generated as described in equation 4.2. The value of the generator (α), the cyclic group G and its order p are publicly known. Alice sends her public key (β) to Bob. Bob now can generate a session key S_k using his private key ($y \in \{1, \ldots, p-1\}$), $S_k = \beta^y \pmod{p}$, where $\beta = \alpha^x$, so $S_k = (\alpha^x)^y \pmod{p}$. Then Bob creates his public key $\gamma = \alpha^y \pmod{p}$. He sends his public key (γ) to Alice, who can generate the same session key that Bob created. Proof:

$$S_k = \gamma^x \pmod{p},\ where\ \gamma = \alpha^y \Rightarrow S_k = (\alpha^y)^x \iff \beta^y = (\alpha^x)^y \pmod{p} \tag{4.4}$$

If Oscar wants to generate the session key used by Alice and Bob, he has to find one of the private keys (x or y). Oscar, in fact, can try to make a brute-force attack on equation 4.3 and create every possible value for x that matches β however, when the p is in the order of 2^{80}, Oscar would need hundreds of years to find the private key x.

The property of the DLP is also used in the signature primitives that is explained in the following sections.

4.2.2 Elliptic Curve Cryptography

Today the Internet protocols are based on RSA digital signature schemes. It is an excellent question to ask why Elliptic Curve Cryptography (ECC) is used in blockchain technology and not RSA. The most significant advantage of ECC against RSA is the length of the operands. The RSA applies operands of length 1024-3072 bits while ECC operates on 160-256 bits. Even if ECC's operands are shorter than RSA's, the same security level is provided.

The ECC is also based on the Digital Logarithm Problem (see equation 4.1) as it was described before, however, there are some specifications that are worthy to note about the ECC. N. Koblitz and V. Miller had the idea to apply DL (Discrete Logarithm) in ECC. Unlike in discrete logarithm, in which the $\mathbb{Z}_p^*$ is used as a cyclic group, the cyclic group in ECC is a group of points on an elliptic curve over a finite field $\mathbb{F}_p$ [40]. The finite field is also called prime field that is composed by a set of integers $\{0, 1, 2 \ldots, p-1\}$. The elliptic curves over a finite field can be represented as follows.

Elliptic curves over finite field $\mathbb{F}_p$

E, an elliptic curve (Koblitz or **Weierstrass** curve) can be defined over $\mathbb{F}_p$, with $p > 3$ and $E(\mathbb{F}_p)$ consists of the set of pairs $(x, y) \in \mathbb{F}_p$, and can be described by the following equation:

$$y^2 = x^3 + ax + b \pmod{p} \tag{4.5}$$

with an imaginary point of infinity ϑ, and with $a, b \in \mathbb{F}_p$. The condition $4a^3 + 27b^2 \neq 0 \pmod{p}$ must also be respected. The precedent definition is adopted from [14] and [40]. Figure 4.1 represents an example of an Koblitz curve $E : y^2 = x^3 - 2x + 2$ defined over $\mathbb{R}$.

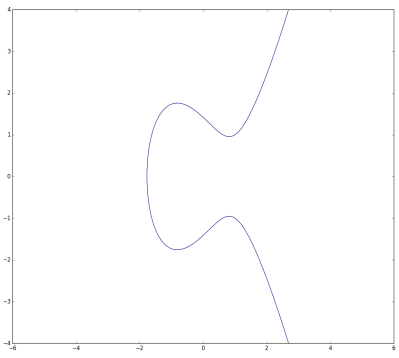


Figure 4.1: An example of an Koblitz curve over $\mathbb{R}$.

The Twisted Edwards curve [113] can be defined as:

$$E : ax^2 + y^2 = 1 + dx^2y^2 \tag{4.6}$$

An example of Edwards curve defined over $\mathbb{R}$ is given in figure 4.2.
$E : 9x^2 + y^2 = 1 + 7x^2y^2$ over $\mathbb{R}$.

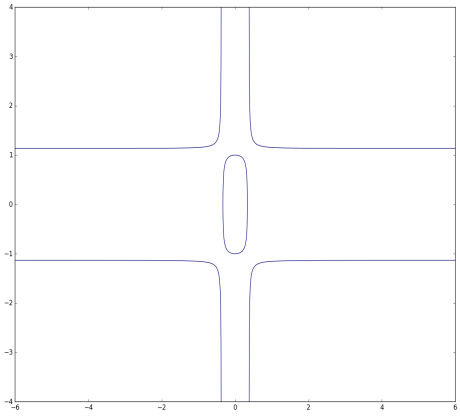


Figure 4.2: An example of an Edwards curve over $\mathbb{R}$.

4.2.2.1 Elliptic Curve Discrete Logarithm Problem (ECDLP)

The Elliptic Curve Discrete Logarithm Problem (ECDLP) can be defined similarly to the DL (see equation 4.1 and 4.2). However, in ECDLP the operation is not multiplication but addition. The definition of ECDLP is that given an elliptic curve E, P is a point (generator point) which is in the curve E. The Discrete Logarithm consists of finding an integer k, where $1 < k < \#E$ and computing the point $Q = kP$.

$$Q = P + P + \cdots + P = kP \tag{4.7}$$

It can be noted that the kP operation is equal to adding the point P to itself k times. Q can also be represented as a point on the curve $Q = (x_Q, y_Q)$.

$\#E$ is the number of points on the curve ($p + 1 - 2\sqrt{p} \leq \#E \leq p + 1 + 2\sqrt{p}$) and p is specified in 160-256 bits. The definition of the ECDLP is adopted from [14] [40] and [12].

This property can also be applied in DHKE (see equation 4.4). The integer k, in this case, can be considered the private key and Q the public key. It can be noted that this operation can take a significant amount of time to compute [12]. According to [14] in modern computer architectures the point multiplication takes around 2ms @3GHz to compute, and hardware accelerators can achieve around 100 μs of latency.

The result of the call graph (see figure 3.10 in page 64), also represents that elliptic curve point multiplication operation took more than 80% of the computation time of the signature

algorithms. Nevertheless, how does point multiplication appears in the signature algorithms and how can it be accelerated using dedicated hardware? These questions are answered in the following subsections.

4.2.3 ECDSA - Sign

ECDSA or Elliptic Curve signature Algorithm is an IEEE, NIST and ISO standard. The results of the previous chapter 3 represented the importance of this algorithm. The ECDSA is used in most blockchain technology with the curve secp256k1. It was also mentioned in the previous chapter that the signature of the transaction is an essential operation in IoT blockchain structures, in order to authenticate the IoT device and the provenance of the data.

The signature algorithm is defined as follows, adopted from [14], [40], [114], and RFC-6979 standard [115]. The message that has to be signed is denoted as msg :

Algorithm 1 ECDSA Sign Algorithm

1.) Calculate the public key $pubKey = privKey \times G$
2.) Calculate SHA256: h = hash(msg)
3.) Generate a secret integer k = hmac($h + privKey$)
4.) Calculate the random point: $R = k \times G \rightarrow r = R.x$
5.) Calculate the signature proof: $s = k^{-1} * (h + r * privKey) \pmod{n}$
6.) Return the signature: $\{r, s\}$

The private key ($privKey$) is generated as a random integer in the range $[0, \ldots, n-1]$ with n the order of the generator point (G) of the curve E. Each value and coordinate has the same bit length as the order's bit length. The symbol $\times$ is used to distinguish the point multiplication and the multiplication of integers. In the step 1.) there is no surprise, as it was already mentioned on page 86 that for public key generation a point multiplication is necessary. However, it must be noted that the public key generation is done only once (when new member is registered) in the blockchain networks and blockchain-IoT networks. Doing so makes sense because the network participants know each other thanks to their public keys (which are determined according to their private key and the fixed generator point G).

The 2nd step is the hash value computation of the message (msg) that would be signed.

In all of the studied blockchains (Ethereum p. 61, Hyperledger Sawtooth p.67, EOS.IO p.56 and Substrate p.71), signature techniques are presented using hash functions. In Ethereum, the transaction's components (T_n, T_g, T_p, T_t, T_d) were hashed before the signing procedure (p.61). In Hyperledger Sawtooth, the *TransactionHeader* and the *BatchHeader* were hashed before their signature (p.67). Actually, these hash values have to be calculated due to the ECDSA algorithm, as step 2 shows.

In step 3, a k random integer is generated. This integer is generated by calling a cryptographic hmac [116] algorithm. The inputs of this function are the hash of the message ($h = hash(msg)$) and the private key ($privKey$).

In the 4th step, a random point ($R \Rightarrow R(x_R, y_R)$) lying on the curve is generated. Here in this step, the elliptic curve point multiplication operation is required ($R = k \times G$). The answer to the question: "How does point multiplication appear in the signature algorithms

?" is then answered. The parameter r is equal to the x coordinate of R ($r = R.x$). The 5th step consists of calculating the signature proof (s).

Finally, the signature is composed of the random point r and the signature proof s as step 6.) presents.

4.2.3.1 secp256k1 elliptic curve

Previously on page 85 it was highlighted that elliptic curve cryptography is based on elliptic curves. Today, many different types of curves (e.g. Curve25519, NIST P-256, etc.) are standardized by IEEE, ANSI, and these curves are used in different application specific cases. The studied blockchains, Ethereum, Hyperledger Sawtooth, EOS.IO uses ECDSA sign algorithm with secp256k1 curve.

Bitcoin applies the secp256k1 curve in the ECDSA schemes. Since two years Bitcoin also provides Schnorr signature algorithms with the secp256k1 curve in certain multi-signature applications. Bitcoin can be viewed as the grandpa of most today's blockchain, and secp256k1 curve as the heritage of the grandpa. According to bitcoin's Wiki page [117], this curve is faster and more secure than other NIST standard curves.

This curve is defined over $\mathbb{Z}_p$, and it can be represented as a Koblitz curve (see equation 4.5, p.85) with the following parameters:

- $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
- $a = 0$
- $b = 7$
- $G_x =$ `79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798`
- $G_y =$ `483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8`
- $n =$ `FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141`

It should be noted that n is the order of G, and its bit length is 256 bits, that means that the values and coordinates in the ECDSA algorithm are expressed in 256 bits. The ECDSA signature $\{r, s\}$ is expressed in 64 bytes while using secp256k1. It can also be noted that the secp256k1 curve is defined over $\mathbb{Z}_p$, which is a finite field or $GF(p)$ Galois field. However, this curve cannot be expressed over $GF(2^n)$ binary Galois field. In elliptic curve cryptography, several curves are defined over the Galois field $GF(p)$ and not on binary Galois field $GF(2^n)$. These two different curve definitions have different mathematical properties, and their implementations are different on software and also on hardware levels.

4.2.4 EdDSA and other Schnorr-like algorithms

Substrate blockchain framework that can probably be the future of designing new blockchains and especially blockchains more adapted for IoT networks. Substrate can be deployed with different types of signature algorithms. However, the default ones are EdDSA and a Schnorr signature algorithms.

4.2.4.1 Schnorr - Sign

This algorithm [118] was invented before the elliptic curve exists, however, it can perfectly be applied with elliptic curves. The algorithm was called according to its inventor, the mathematician Clause Schnorr. This signature algorithm is similar to ECDSA. However, slight differences are making the computation easier than ECDSA.

The signature algorithm is defined as follows, adopted from [118], [119], [120]. The message that has to be signed is denoted as msg :

Algorithm 2 Schnorr Signature Algorithm

1.) Calculate the public key $pubKey = privKey \times G$
2.) Generate a random integer k: $\rightarrow$ RNG($\{1, 2, \ldots, n-1\}$)
3.) Calculate the random point: $R = k \times G$
4.) Calculate h = hash($R_x || R_y || msg$) $\pmod n$
5.) Calculate signature proof: $s = k + h * privKey \pmod n$
6.) Return the signature $\{h, s\}$

The public key ($pubKey$) generation (step 1) is done in the same way as in ECDSA algorithm (see algorithm 1). This step can be jumped for the same reasons as in ECDSA. The random integer k is generated according to a Random Number Generator function in the range that cannot go beyond the order (n) of the generator point G (step 2). In step 3, the point multiplication operation is used for the random point generation $R \rightarrow (R_x, R_y)$. The message (msg) to be signed is concatenated with the coordinates of the random point r. The concatenated value is then hashed to obtain h (step 4). Comparing the signature proof calculation with the ECDSA algorithm 1 step 5, the Schnorr algorithm does not inverse the secret integer (step 6). Avoiding this inversion, the computation of the signature proof is less computationally expensive. Finally, the signature is composed of the hash and the signature proof $\{h, s\}$.

The Schnorr algorithm was explained above because of two reasons. First, a Schnorrkel protocol of Substrate is based on the Schnorr algorithm. Secondly, EdDSA that is also used in Substrate, is also a variant of the Schnorr algorithm.

4.2.4.2 Ed25519 Signature Algorithm

Ed25519 is one of the schemes of EdDSA [99], that was invented to allow using twisted Edward curves in elliptic curve signature algorithm. Depending on the parameters, EdDSA can implement Ed25519 and Ed448 schemes.

The signature algorithm is defined as follows, adopted from [99], [114], and RFC-8083 standard [121]. The message that has to be signed is denoted as msg :

Unlike in ECDSA algorithm 1 and in Schnorr algorithm 2, here the hash of the private key $privKey$ is used to generate the public key (step 2). Likewise, in ECDSA the steps 1 and 2 can be jumped for the same reasons that were explained previously. The secret integer k is created according to the hash of the private key ($privKey$) that is concatenated with the message (msg), and the concatenated value is hashed again (step 3). In step 4 the random point (R) lying on the curve is calculated according to the point multiplication operation. In step 5, the hash function is computed on the concatenation of the compressed

Algorithm 3 EdDSA Signature Algorithm

1.) s = hash($privKey$)
2.) Calculate $pubKey = s \times G \quad \rightarrow pubKey_{comp.}$ = ENC($pubKey$)
3.) Generate a secret integer: k = hash(hash($privKey$) || msg) (mod n)
4.) Calculate the random point: $R = k \times G \quad \rightarrow R_{comp.}$ = ENC(R)
5.) Calculate h = hash($R_{comp.}$ || $pubKey_{comp.}$ || msg) (mod n)
6.) Calculate signature proof: $S = k + h * s$ (mod n)
7.) Return the signature $\{R_{comp.}, S\}$

form of the random point ($R_{comp.}$), the private key ($pubKey_{comp.}$) and the message (msg). The compression of R and the $pubKey$ is necessary because they are points lying on the curve. The total length of a point is twice longer than the order (n) of the generator point's (G) bit length. The function (ENC) compresses the input length to the same length as the generator point's order. The secret integer k, the hash h and the hash of the private key s is used to determine the signature proof S. The signature is composed of $\{R_{comp.}, S\}$.

4.2.4.3 edwards25519 elliptic curve

Substrate blockchain framework applies edwards25519 curve, which one is used in Ed25519. The edwards25519 curve can be defined with the previously presented equation 4.6 (p.86). The parameters of this curve are:

- $p = 2^{255} - 19$
- a = -1
- d = `52036CEE 2B6FFE73 8CC74079 7779E898 00700A4D 4141D8AB 75EB4DCA 135978A3`
- G_x = `141DA6F0 01AE0297 0D02D370 29CECCF8 D3038A00 D61B2C95 62D608F2 5D51A`
- G_y = `66666666 66666666 66666666 66666666 66666666 66666666 66666666 66666658`
- $n = 2^{252}$ + `14DEF9DEA2F79CD65812631A5CF5D3ED`

The bit length of EdDAS of values and coordinates when edwards25519 curve is used is 256 bits. In addition to bit length, the hash function ($hash$) used in EdDSA produces a 512 bits length hash digest. In RFC-8082, the SHA-512 function is used. Substrate applies BLAKE2b cryptographic hash function for this reason.

It can be concluded that all of the previously presented signature algorithms operate the elliptic curve point multiplication in order to achieve the discrete logarithm problem. Because these algorithms are based on point multiplication, the computation of these algorithms is expensive, as was mentioned in previous sections.

The following section describes some of the existing hardware accelerators for Elliptic Curve Point Multiplication (ECPM), which can be used in ECDSA with secp256k1 and in EdDSA with edwards25519 curves.

4.2.5 Hardware Implementations for ECPM

This subsection describes elliptic point curve multiplication hardware accelerators that can be deployed in the proposed IoT hardware model to accelerate the point multiplication in ECDSA and Ed25519 algorithms. It is necessary to keep in mind that the hardware accelerator must enable point multiplication on secp256k1 and edwards25519 curves. Therefore, this subsection aims to find hardware accelerator designs that can work with secp256k1 and edwards25519 curves.

It must be noted that most ECC hardware accelerators are designed to operate over prime field $\mathbb{F}_p$ and over binary prime field $\mathbb{F}_{2^n}$. First of all, the designs working over the binary prime field $\mathbb{F}_{2^n}$ cannot be used because secp256k1 and edwards25519 curves are defined over a 256-bit prime field $\mathbb{F}_p$ ($p \leq 256$). Therefore, the prime field length is also an essential requirement that must be taken into account. Hence, designs over a prime field of less than 256 bit cannot be used.

A recent work [10] compares and provides hardware implementations (ASIC and FPGA) of ECC processors over bit field $\mathbb{F}_{256}$ and $\mathbb{F}_{224}$. Even if this work proposes a good throughput, which is better than the compared designs' it cannot be used with secp256k1 curve. The reason is that even if a design was implemented for bit field 256, in most cases, designs support only on NIST-recommended elliptic curves. These curves are based on prime p, however this prime is a Generalized Mersenne Prime which makes that curves over this prime field have a specific form [122]. As a result, NIST-recommended curves can be executed faster due to their specific form, but the hardware designs supporting only NIST could not execute other curves defined over the general primes. This is because some inversion operations are harder to do over general primes. NIST recommends the edwards25519 curve. However, the hardware design proposed in [10] is specific for Weierstrass form curves, which is different from Twisted Edwards curves form. This accelerator design (see on figure 4.3) cannot neither work with edwards25519 curve. The design's implementation on Virtex-7 FPGA provides a latency of 2.44 ms @ 122.8 MHz.

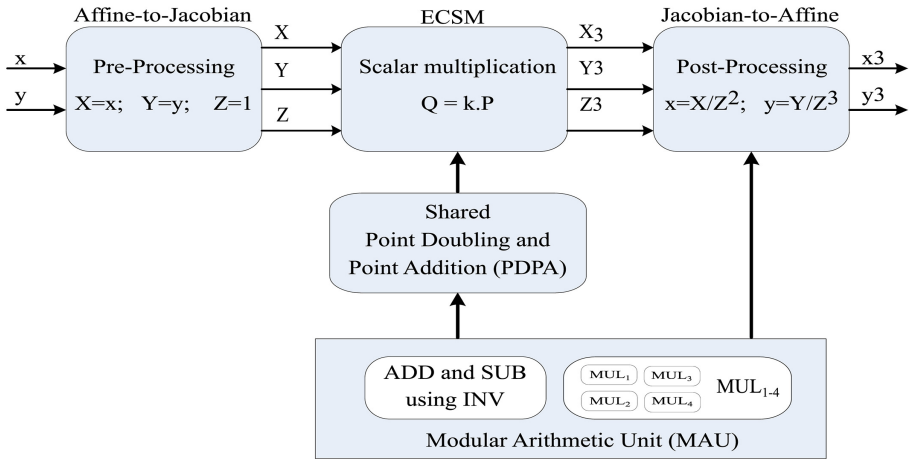


Figure 4.3: Simplified design for elliptic curve point multiplication [10]. The figure is retrieved from [10].

It is worth noting that in the choice of the hardware accelerators, the form of the curve also plays an essential factor.

Shah *et al.* [123] deal with this problem, and they highlight that today's designs are based on NIST-recommended curves. The authors propose an ECC implementation for arbitrary curves over the general prime field to solve this problem. Any curves (of Weierstrass form) can be used over the prime field size from 256 to 512 bits. Hence the prime p can have any value (in the range of 256 and 512-bit length). Over 256 bit field $\mathbb{F}_p$ the design's implementation on Virtex-6 FPGA can provide a latency of 0.65 ms @ 144.5 MHz.

Another work [124] also proposes a design that can operate not only on NIST-recommended curves. The authors also mention that the hardware design can operate on all of the "secure elliptic curves" [125] analyzed by the team of D. J. Bernstein (secp256k1 is included). The latency of this design implemented on Virtex-6 FPGA is 2.01 ms @ 95 MHz. In this study authors compare their design with related works. It can be noticed that NIST curve based designs are in general slightly faster. However among the general prime field designs this is one of the most efficient in terms of latency and area.

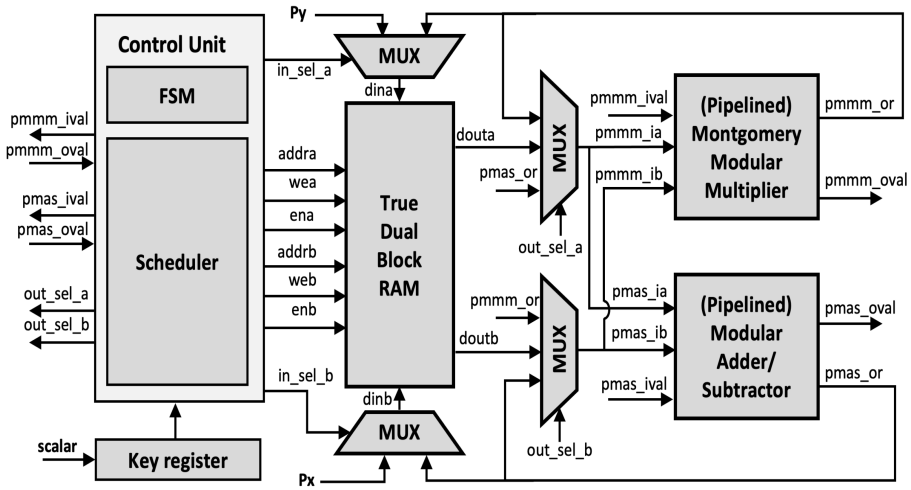


Figure 4.4: The elliptic curve cryptography (ECC) architecture proposed in article [11]. The figure is retrieved from [11].

A recent ECC processor design [11] also enables the point multiplication over general prime field (256 bit, including secp256k1 curve). The design is implemented on multiple of FPGA cards. The implementation on Virtex-7 FPGA provides a latency of 0.158 ms @ 204.2 MHz. On XC72020 (Xilinx Zynq-7000 family) the design's implementation provides a low latency 0.206 ms at 156.8 MHz of frequency. This architecture (see figure 4.4) implements a pipelined Montgomery Modular Multiplier (pMMM) to accelerate the operations, and it also contains a BRAM for storing the elliptic curve's parameters. That also means that the parameters have to be saved only once, and only the scalar integer value has to be written as input when performing the EC point multiplication.

It can be noted that even if edwards25519 is a NIST-recommended curve, the most of the designs defined over the prime field allowing operations on NIST-recommend curves are specified for Weierstrass form curves. Therefore, a specific hardware is required for edwards25519 curve. The operation on secp256k1 curve is possible with hardware accelerators designed over the general prime field, or with accelerators dedicated for this curve.

Mehrabi *et al.* [12] propose a design for secp256k1 and edwards25519 curves (see figure

4.5). The authors use a Residue Number System (RNS) for the integers representation that shows better performance in crypto-systems. The authors also implement multiple methods for scalar multiplication to determine which method works better with RNS systems. According to them, the GLV method is the most efficient for the secp256k1 curve, and DBC L-T method for edwards25519 curve. The latency of the design implemented on Virtex-7 FPGA when operating on the secp256k1 curve is 0.25 ms @ 125 MHz and 0.28 ms @ 125 MHz for operations on the edwards25519 curve. The Implementation on a more recent Virtex-UltraScale+ FPGA provides a latency of 0.176ms @ 181.8 MHz on the secp256k1 curve and 0.198 ms @ 181.8MHz on the edwards25519 curve.

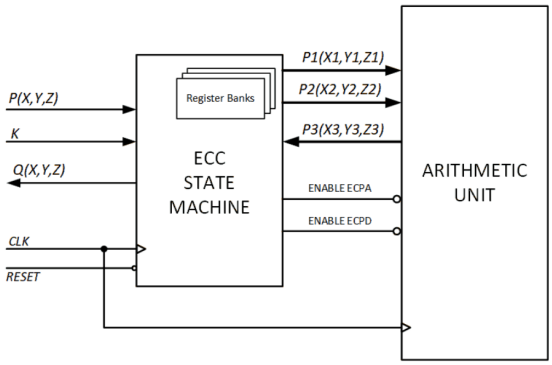


Figure 4.5: RNS ECC Core Hardware implementation [12]. The figure is retrieved from [12].

The paper [126] proposes an ASIC co-processor for ECC acceleration over $\mathbb{F}_p$ including secp256k1 curve (Montgomery Multiplier method is used for the multiplications). Unfortunately, this architecture is almost 100 times slower than the FPGA implementations of designs proposed in [12] and [11]. This significant difference is because the design proposed in [126] was supposed to be resistant against a Side-Channel attack. However, recent work [127] has succeeded with a Side-Channel attack on this design.

The design presented in a technical report [128] implements a key encapsulation mechanism, in which a hardware module for EC point multiplication on the secp256k1 curve is also mentioned. This Virtex-5 FPGA implementation provides a latency of 0.6ms @ 43MHz, close to the results of the design's implementation proposed in [12](0.25ms @ 125 MHz). Unfortunately, the ECC module cannot be used separately, because it is integrated in an complex hardware architecture.

There also exists another ASIC implementation for ECs over general fields [129]. However, this architecture operates on 556 MHz and provides a latency of 1.01 ms. This implementation is less efficient than the previously mentioned designs.

A hardware accelerator design implemented on Virtex-7 FPGA was developed in [130] to accelerate the EC point multiplication on the edwards25519 curve. This implementation provides a significant latency of 1.48 ms @ 177.7 MHz, and this implementation cannot beat the design's implementations for the edwards25519 curve proposed in [12].

In [13] the authors propose a hardware accelerator (see in figure 4.6) for EdDSA scheme (signature, verification, and key generation process). This design can perform the signature

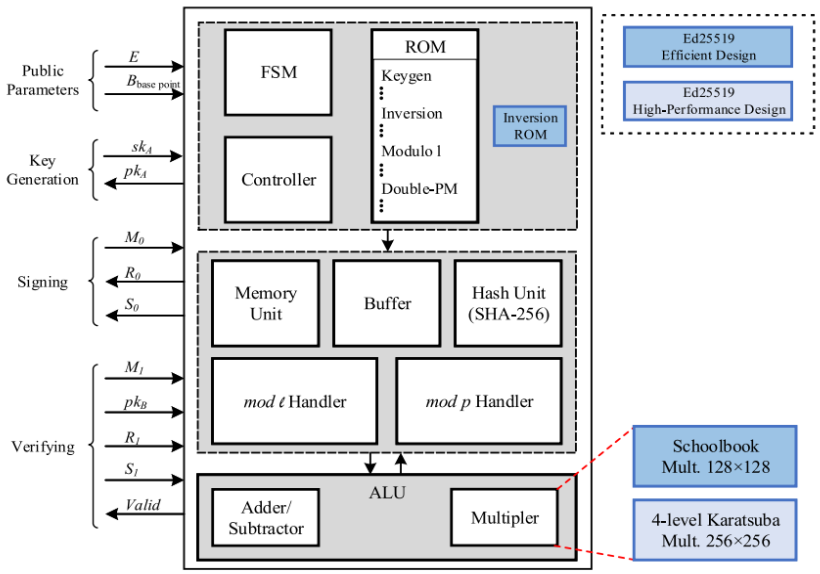


Figure 4.6: Hardware accelerator architecture for EdDSA signature, verification and key-generation [13]. The figure is retrieved from [13].

of an input message. Doing it by performing the SHA-512 hash of the input message and signing it according to the Ed25519 scheme. The EC point multiplication module that used in this architecture can achieve a significant 0.126 ms latency @ 73 MHz. It would have been obvious to use the complete architecture not only for EC point multiplication but for the whole signature process. However, the design is limited to use uniquely SHA-512 hash algorithm. For example, in the case of Substrate API the BLAKE2b hash algorithm must be used. The signature process on a 1024-bit message could be performed with a latency of 0.16 ms @ 73 MHz.

4.2.6 Discussion on ECPM hardware designs

Table 4.1 represents the characteristics of the hardware accelerators cited previously to have a more comfortable review of the possible designs. The gain (speedup of the signature process) is calculated as follows: the ECDSA and EdDSA algorithms with curves secp256k1 and edwards25519 were executed on BCM2837 (Raspberry Pi 3 B+) ARM-based architecture. The algorithms were deployed in Trezor-crypto cryptographic library. The total time taken by these signature algorithms were also measured. The digit signature of ECDSA is 2.6 ms, EdDSA processes the message digests signature in 0.71 ms.

The result of the analysis of the transaction creation (see p.64) showed that the point multiplication in ECDSA takes around 85% of the *ecdsa_sign_digest* function, making the signature on the hash of the message (message digest). Hence the speedup is done on 85% of the signature function, its 15% does not change. The total gain (speedup) can be calculated according to the equation:

$$Speedup = \frac{T_{total}}{T_{total}{}'} = \frac{T_{total}}{(T_{total} * 0.15) + T_{HardwareAccelerator}} \tag{4.8}$$

with T_{total} the total execution time before the acceleration, $T_{HardwareAccelerator}$ the latency

of the hardware accelerator and T'_{total} the total execution time after acceleration.

The speedup for EdDSA signature can also calculated by using equation 4.8. However the ECC point multiplication takes 81.4% of the digit signature process, which also means that T_{total} has to be multiplied by the value 0.186 instead of 0.15.

Arch.	Impl. Type	Prime p 256-bit / Curve	Latency (ms)	Freq. (MHz)	Power (mW)	Speedup
[10]	Virtex-7	NIST	2.44	122.8	-	-
[123]	Virtex-6	Any	0.65	221	-	2.5
[124]	Virtex-6	Any	2.01	95	-	1.08
[126]	ASIC	secp256k1	23.06	100	-	-
[12]	Virtex-US+	secp256k1	0.176	181.8	-	4.59
[12]	Virtex-US+	edwards25519	0.198	181.8	-	2.15
[12]	Virtex-7	secp256k1	0.25	125	-	4.06
[12]	Virtex-7	edwards25519	0.28	125	-	1.72
[128]	Virtex-5	secp256k1	0.6	43	-	2.63
[11]	XC7Z020	Any	0.206	156.8	-	4.36
[11]	Virtex-7	Any	0.158	204.2	-	4.74
[129]	ASIC	Any	1.01	556	-	1.85
[130]	Virtex-7	edwards25519	1.48	177.7	-	-
[13] *	XC7Z020	edwards25519	0.126	73	-	5.63

Table 4.1: Comparison of ECC point multiplication hardware accelerators. Here *Any* means any curves of **Weierstrass** form (see p.85). (*) the design is not protected against side channel attack.

It can be observed that several hardware accelerators do not provide a speedup because, in the BCM2837 (ARM-based) architecture, the software implementation can already be executed fast. It can also be seen that the designs are implemented on different FPGAs, this fact also makes it hard to decide which design is the most efficient.

Because the EC point multiplication module cannot be called separately proposed in the work [13], the design proposed in [12] and implemented on Virtex Ultra Scale+ is selected to be used in the proposed IoT architecture model (green line in Tab. 4.1). It must be noted that in this design the coordinates of the curve point (output) use an RNS format that must be converted before the cryptographic software application can use them.

In the following discussion, we compare the two most promising designs for computing EC point multiplication on the secp256k1 curve. We call *design1* the architecture proposed in [11] and *design2* the architecture realized in [12].

When *design1* is implemented on Virtex-7, a higher speedup can be achieved comparing with *design2* implemented on the same FPGA. However, the *design1* requires a higher frequency. This same design can also be implemented on XC7Z020 (Zynq-700) and it can produce almost the same speedup as the *design2* Virtex-7 implementation. The only negligible bottleneck of *design1* implemented on XC7Z020 is that it has to apply a slightly higher frequency than *design2* applies (156.8 MHz and 125 MHz).

One of the disadvantages of *design2* implementations is that the input/output coordinates of the curve points must be formatted as a Residue Number System (RNS). The generator

point coordinates can be stored initially in the IP's registers, however when retrieving the result from the IP the coordinates must be converted from RNS and from Jacobian to affine-coordinates. The *design1* uses affine coordinates, no additional conversation from RNS is required, and it also allows the storage of curve parameters in its BRAM (see figure 4.4).

It can also be noted that the *design1*'s implementation on Virtex-7 uses nearly 2 times less FPGA components (FF, LUTs etc.) than the *design2*'s implementation on Virtex-7. That also means that *design1* is 2 times smaller and consumes at least 2 times less energy.

Due to the additional RNS conversions, choosing the *design2* for multiplications on secp256k1 curve does not seem to be optimal. The design of*design1* implemented on XC7Z020 can produce almost the same latency while applying almost the same frequency as the *design2*'s implementation. Therefore, the proposed IoT architecture model includes the *design1* in order to perform EC point multiplication on the secp256k1 curve (orange line in Tab. 4.1).

It can also be observed that the execution of EC point multiplication operation on edwards25519 curve on the top of a complex architecture like BCM2837 can be performed already fast, and the speedup provided by a hardware accelerator is less significant than in the case of secp256k1 curve. EC point multiplication on the secp256k1 curve provided by the hardware accelerator proposed in [11] can be four times faster than execution on complex architecture mentioned before.

Architecture	Latency (ms)	Speedup
Cortex M0+	683.110	6.65
Cortex M0	471.009	6.64
Cortex M4	124.938	6.59

Table 4.2: The estimated speedup of EC multiplication (secp256k1 curve) comparison between less complex ARM-based architectures and the hardware accelerator design proposed in [11]. The latency of the architectures are retrieved from [5].

A more significant speedup can be observed if the architecture executing the EC multiplication is less complex than BCM2837. The related work [5] measured the latency of the ECDSA signature on secp256k1 curve (signature of a transaction in Ethereum) on different ARM-based architectures. These architectures are less complex than the BCM2837. The following table represents the estimation of the speedup of the hardware accelerator that can be achieved against the three studied ARM architectures (Cortex M0+, Cortex M0, Cortex M4). The speedup estimation are calculated with the equation 4.8. It can be concluded that in less complex constrained devices the speedup can be more significant.

4.2.7 Conclusion

A brief background was presented at the beginning of this section in order to understand better the elliptic curve digital signature algorithms (ECDSA and EdDSA). It was also demonstrated how the point multiplication appears in these algorithms.

The second part of the section discusses the requirements to be considered while selecting a hardware accelerator for secp256k1 and edwards25519 curves. It can be concluded that there are many requirements that must be taken into account (e.g., prime field, prime field

length, curve's form, frequency, latency, etc.) while choosing a hardware accelerator design for secp256k1 and edwards25519. The rest of the section compares the existing design for EC point multiplication using secp256k1 and edwards25519 curves. Speedup estimation is also given, comparing the latency of the hardware accelerator and the ECPM operation execution on a complex ARM-based architecture (BCM2837) (the list of the compared hardware accelerator designs can be seen in tab. 4.1).

The speedup that can be achieved against more constrained devices while using the secp256k1 curve (presented in table 4.2). It can be observed that in complex and constrained ARM-based architectures, the speedup that can be achieved while accelerating ECPM on the secp256k1 curve is in the range of 4-6.5.

4.3 Hash functions

The goal of any hash function is to produce a short fixed-length hash value, also called a message-digest, from an input message of arbitrary length. The hash-digest must be a unique representation of the message, also called the fingerprint of the message [14] [39]. The hash function can be represented as the following equation depicts:

$$H = h(m) \tag{4.9}$$

Where *h()* is the hash function, *m* is the message and H is the hash value.

The hash functions are one-way operations, which means that the message cannot be retrieved from the hash value. There is no inverse function ($h(H)^{-1}$) for doing so. The one-way also means that the function does not use a secret key for the hash production, unlike in the encryption primitives (AES uses a secure key to encrypt and decrypt the message). The hash value is a unique short fixed-length representation of the message. In the modern hash functions, this length is greater than 128 bits. Most of the blockchain technologies use hash functions performing the message digest of 256 bits length. The message digest's length is an essential factor in the choice of the hash function. The higher the number of bits used for the message digest, the higher the resistance to collision attacks. The collision attack or birthday attack consists of finding the message that corresponds to the hash value. To determine successfully the message which corresponds to the hash value of n bits, the attacker has to hash $2^{n/2}$ messages [14].

Any cryptographic hash function is sensitive when the message to hash varies. One bit difference performs an entirely different hash value. An example of hash (SHA-256) production is represented below when only one bit has been changed in the input message. Between the ASCII code of the letter "c" and "d", only one bit differs.

- `h("abc") = ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad`
- `h("abd") = a52d159f262b2c6ddb724a61840befc36eb30c88877a4030b65cbe86298449c9`

The previous sections described the digital signature algorithms, such as ECDSA, EdDSA, and Schnorr-like signatures. It could be observed that the hash function is an essential component of all of these algorithms because the message (or transaction in the case of blockchains) is hashed, and finally, it is the hash of the message that is signed and not the

message itself. Performing the signature of the hash of the message and not the message itself is done for two reasons.

First, the digital signature primitives (ECDSA, EdDSA, RSA) are implemented to sign a fixed-length input message. In the case of ECDSA using the secp256k1 curve, the length of the message to sign is 256 bit. It makes sense that ECDSA using the secp256k1 curve (see page 87) applies the SHA-256 hash function, which produces 256 bits length message digest.

The second reason is also due to the property of the fixed-length input. An input message to be signed can have an arbitrary length. If the signature is to be performed on an input message of arbitrary length, the message must be divided into chunks with a chunk length that meets the requirements of the given ECDSA input length. In the case of ECDSA using the secp256k1 curve, the chunk length is 256 bits.

This problem is represented by the equation below, $sign(M, PrivKey)$ is the signature function that signs the message *m* with a private key (*PrivKey*). The generator point *G* is not indicated in the sign function in order to simplify the demonstration. It can also be observed that this method is expensive in terms of computation because as many signatures should be performed as the number of chunks. In this case it is also clear that sending a signed message from A to B would have taken a longer information because the signature would have been twice longer than the message itself.

$$\begin{gathered} M = \{m_1, m_2, \ldots, m_n\}, \quad i \in \{1, \ldots, n\}, \quad len(m_i) = 256\ bits \\ sign(M, PrivKey) = \{signature_1, signature_2, \ldots, signature_n\} \\ with \quad signature_i = sign(m_i, PrivKey) \end{gathered} \tag{4.10}$$

The message to be signed has an arbitrary length. The hash function is a perfect candidate to perform the unique short fixed-length representation of the message. Sign the hash value of the message makes sense as the signature algorithms operate on a short fixed-length value input. Figure 4.7 depicts how the signed message sending is done between Bob and Alice.

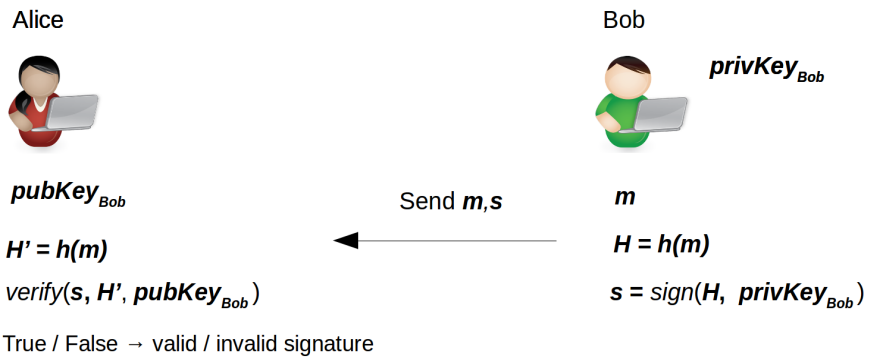


Figure 4.7: Advanced example of the signature process

Bob wants to send a signed message (m) to Alice, and Alice has Bob's public key ($pubKey_{Bob}$). After Bob's message is hashed, the hash value of the message ($H = h(m)$)

is signed with Bob's private key ($privKey_{Bob}$) thanks to a signature function ($sign()$). Bob sends the message (m) and the signature (s) to Alice.

First, Alice performs the hash of Bob's message ($H' = h(m)$). It should be noted that Bob and Alice use the same hash function ($h()$) to obtain the same hash value on both sides. Next, she verifies the signature with the hash of the message and with Bob's public key ($verif(s, H', pubKey_{Bob})$). If the message were manipulated, the hash that Alice created would defer to Bob's ($H' \neq H$), and the signature will not be validated. Same idem, if the message were not manipulated but the signature was signed by someone else than Bob ($privKey_{Bob}$), the signature would be invalid.

It can be concluded that the cryptographic hash function is an essential element of the digital signature algorithms, whether it be an elliptic curve signature primitive or a classic one (RSA, for example). It can also be noted that the hash creation procedure is also used in cloud computing and simple email sending to prove that the message was not altered while sending it. Today hash function is a necessary element of cryptography and secure communication.

The following section gives an overview of the fundamental operation of the cryptographic hash algorithms identified in section 3.7.

4.3.1 Hash Algorithms

This section describes the principle of how cryptographic hash algorithms work. It can be noted that the objective of this section is to give a piece of helpful information about the hash algorithms and about how to use these algorithms. The section does not go into hard mathematical details of the given algorithms. The following picture 4.8 illustrates an example of the basic components (steps) and the basic operation of most hash functions.

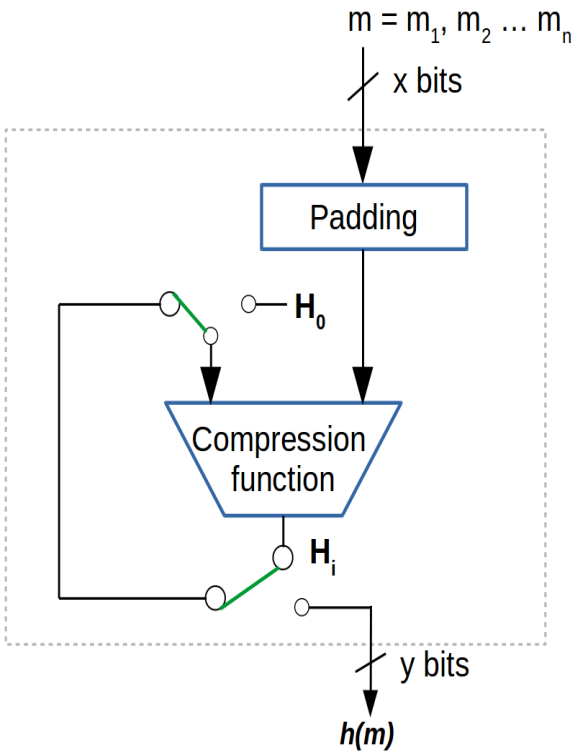


Figure 4.8: General representation of a hash function. The figure is adapted from [14].

The input of the hash function is the message (m). It was already mentioned that the input message length is arbitrary. The input message is divided into x bits length chunks, also called blocks ($m_1, m_2...m_n$). The length of a chunk depends on the hash function type. For example, an SHA-256 hash function's chunk length is equal to 512 bits (x is equal to 512 bits). The message has to be padded to be able to work with x bits length chunks.

The padding is also dependent on the given hash function. In several hash functions, only zeros are added to complete the x bits chunk. However, there are pending methods in which, in addition to the zero-padded last chunk, the total length of the message is also added to this same last chunk.

The core of the hash function is the compression function (in the Keccak hash function, this element is called the random Permutation function). The Compression function contains different types of bitwise operations such as XOR, rotation shift, etc. The sequences of these operations can be executed in rounds (in loops). In a software implementation, the chunks of x bits are divided into arrays of 64, 32, or 8 bits, depending on the implementation.

The Compression function takes two inputs, a chunk of the message (m) and the intermediate hash value, which was calculated for the previous chunk. For better understanding, the equation below shows an example. For computing, the hash of chunk m_2, the previous chunk's m_1 hash value is needed, so H_1. This intermediate H_1 was performed in the previous sequence and stored in H_i, where i is 1 in that case. So H_1 is looped back to the Compression function. The result for m_2 is H_2, if the message (m) would have divided only into two blocks, the hash value of m would have been equal to H_2.

The initial hash value (also called initial state) is denoted as H_0. This initial hash value varies according to the hash function. In SHA-256, 512, and Blake, this value is composed of a prefixed constant. In Keccak the first chunk of the message is mixed in a particular way with a 1600 bits register set to zero. In a software implementation, the hash value H is also composed of an array. An SHA-256 intermediate hash can be represented as an array of eight 32 bits length elements (8x32 bits): $H_i{}^0$, $H_i{}^1$, $H_i{}^2$, $H_i{}^3$, $H_i{}^4$, $H_i{}^5$, $H_i{}^6$, $H_i{}^7$.

SHA-256 and SHA-512

Secure Hash Algorithms (SHA) is a family of hash functions that takes part in the Federal Information Processing Standard (FIPS). It can be noted that this hash family is one of the most used hash families, and most Internet applications are based on it. Most software implementations of SHA (libraries in several programming languages) are based on the Request for Comments (RFC) standards [131].

SHA-256 performs a hash value of 256 bits (32 bytes) according to an input message that length is supposed to be less than 2^{64} bits. The input message is divided into chunks of 512 bits before the padding operation.

SHA-512 performs a hash value of 512 bits (64 bytes) according to an input message that length is supposed to be less than 2^{128} bits. The input message is divided into chunks of 1024 bits before the padding operation [131].

Keccak-256

The Keccak hash (sponge) function family was selected as a winner of the SHA-3 Cryptographic Hash Algorithm Competition proposed by NIST [132]. According to the input parameters, the Keccak hash function can perform hash values of 224, 256, 384, and 512 bits [133]. The Keccak-256 hash function performs a hash value of 256 bits. The input message is divided into 1088 bits blocks. The compression function of Keccak is called Keccak-f[1600], where 1600 refers to the number of bits over which the operations to obtain the internal hash are done. In the case of Keccak-256 the hash digest is 256 bit long.

BLAKE2b

BLAKE2 is also an RFC standard [134] that is also known as a concurrent of the SHA-3 hash family [135]. Likewise SHA-512, BLAKE2b performs 512 bits hash values. The input message length must be less than $2^{128} - 1$ bytes.

It can be noted that the most expensive part of the hash function in terms of computational power is the compression function. This function and the padding logic vary for each hash function, making the execution faster or more resilient to attacks.

In general, PC architectures (with 64 bits processors) can run the hash task relatively quick near to 1GB/sec throughput [14]. It is not unusual that the hash algorithms and mainly the compression functions are implemented in ASIC to achieve a smaller architecture and faster operation. Hash hardware accelerators are usually implemented in embedded systems next to the processor. The following subsections represent some of the hash hardware accelerators which can be used in the proposed IoT hardware architecture model.

4.3.2 Hardware Implementations of hash algorithms

This section describes hardware accelerators for accelerating the hash algorithms that were identified in the previous chapter. Some of the architectures are open-source to be able to deploy them on FPGAs or on ASIC. Most of these hardware designs take part in academic research works. It can be noted that the performance characteristics, such as latency, energy consumption of the following architectures could seem not up to date. This is natural because some of the architectures are implemented with older CMOS technology. However, estimating the performance characteristics of ASIC-implemented architectures when using newer CMOS technologies is possible, thanks to the CMOS scaling estimation method [104].

SHA-256

The hardware accelerator design proposed in [136] can realize SHA-256, Keccak-256, Blake, and many other hash functions in a single chip (1875µm x 1875µm). This hardware architecture accelerates the core (compression function) of each hash function[1].

Our previous work [19] that contributes to this thesis work shows that the SHA-256

[1]The source code of this architecture can be found: https://iis-people.ee.ethz.ch/~sha3/

hardware accelerator design proposed in [103] can achieve a speedup of 127 against ARM-based operation (one hash operation takes 14307.2 ns on Raspberry Pi 3 B+).

This hardware accelerator generates one internal hash function in 112.64 ns, using 45 nm CMOS technology. The Raspberry Pi 3 B+ uses a CMOS near 40 nm. For achieving such a significant gain, a frequency of 294.55 MHz must be applied. Another previous work pf us [17] that contributes to this thesis work, also proposed hardware acceleration on SHA-256. In this case the hash creation is faster (24.48 ns), but a frequency of 794MHz must be applied to achieve this latency.

In mentioned work [17], the main objective was to accelerate the hash function as much as possible. However, the applied frequency is high, and it can cause a negative impact on the dynamic power consumption, which is given by the equation below.

$$P_{dynamic} = \alpha * C * f * v^2 \tag{4.11}$$

With α is the portion of an IP which works at the given time. $\boldsymbol{C}$ is the capacitance of the architecture, $\boldsymbol{f}$ is the frequency and $\boldsymbol{v}$ is the supply voltage. The following table (4.3) represents a comparison of the characteristics of the two mentioned ASIC hardware accelerators for hash creation, which were implemented on older CMOS technology. The CMOS scaling method [104] was used to scale these architectures to a 40-45 nm CMOS technology. The throughput (*Tp*) is calculated according to (40-45 nm CMOS technology).

Arch.	Impl. Type	Tp (Gbit/s)	Latency (ns)	Freq. (MHz)	Gain	Open-source
[136]	ASIC	4.502	112.64	294.55	24.5	✓
[103]	ASIC	20.9125	24.48	794	112.75	✗

Table 4.3: Comparison of ASIC implementation of SHA-256 hash function

The gain that can be achieved against ARM is calculated according to the ARM's (BCM2837) SHA-256 operation which is 2760 ns of latency (CryptoPP C++ library), retrieved from [17].

It worth noting that, if the same frequency (794MHz) would have to be applied in the IP proposed in [136], the latency of the other architecture proposed in [103] would only be twice less important as the architecture proposed in [136]. The most optimal choice between these two implementations is marked in orange in the table. The IP proposed in [136] performs a significant gain by applying a relatively low frequency. Thanks to these features this IP was chosen to be used in the proposed IoT hardware model as it is marked in orange in table 4.3.

SHA-512

A previous work [17] that contributes to this thesis work, also proposed hardware acceleration on SHA-512 realized by the hardware accelerator proposed in [103]. In this case, the hash creation takes 32 ns, but a significant frequency of 746MHz must be applied to achieve this latency. It is clear that the frequency applied by this IP is too significant, and it causes a negative impact on power consumption.

The following table (4.4) represents a comparison of the characteristics of the mentioned ASIC hardware accelerator, and another hardware ASIC design proposed by [137]. All of

the implementations were realized on older CMOS technology. Therefore, CMOS scaling method [104] was used to scale these architectures to a 40-45 nm CMOS technology. The throughput (*Tp*) is calculated according to (40-45 nm CMOS technology).

Arch.	Impl. Type	Tp (Gbit/s)	Latency (ns)	Freq. (MHz)	Gain	Open-source
[103]	ASIC	31.836	32	746	293.18	✗
[137]	ASIC	10.18	101	250	97.35	✗

Table 4.4: Comparison of ASIC implementation of SHA-512 hash function

The gain that can be achieved against ARM-based architecture BCM2837 is calculated according to the ARM's SHA-512 operation which is 9832 ns of latency (CryptoPP C++ library), retrieved from [17]. The architecture implementation marked in orange of the table 4.4 can be considered as the optimal one. The IP realized by [137] provides a gain that is three times less than the gain of the design proposed in [103]. According to the gain, the applied frequency of the design proposed in [103] is three times less important than the frequency required by the design proposed in [137]. Therefore, the same remark can be given as in the case of SHA-256. If the IP of [137] would have been used with the same frequency applied in the design proposed in [103] then the IP proposed in [137] would have also performed the same gain.

Keccak

In the previous section that discusses the hardware acceleration of SHA-256, it was already mentioned that the IP proposed in [136] can also accelerate the Keccak-256 function. The following table 4.5 compares this accelerator with another ASIC implementation. The gain that can be achieved against ARM-based architecture BCM2837 is calculated according to the latency required for Keccak operation performed in BCM2837. This latency is equal to 30768 ns when CryptoPP C++ library's Keccak implementation is executed (result retrieved from [19]).

Arch.	Impl. Type	Tp (Gbit/s)	Latency (ns)	Freq. (MHz)	Gain	Open-source
[136]	ASIC	44.01	35.2	485.44	890	✓
[138]	ASIC	37.345	43	284	715	✗

Table 4.5: Comparison of ASIC implementation of Keccak hash function

It can be noticed that the IP proposed in [138] performs a gain close to the IP proposed in [136], but it requires a frequency that is more than half lower. The IP that is used in the proposed IoT hardware model is marked in orange in the table 4.5.

BLAKE2

BLAKE2 was published some years after BLAKE as an improved version of BLAKE. BLAKE2b is used as a default hash function in the Substrate blockchain framework. However, the hardware implementations of BLAKE2b in related research works and the industrial

market are insignificant. The following table 4.6 represents one ASIC implementation. The gain is determined according to the latency measurements of the BLAKE2b hash function of the CryptoPP C++ library executed on BCM2837. When The intermediate hash is calculated in 13203.2 ns.

Arch.	Impl. Type	Tp (Gbit/s)	Latency (ns)	Freq. (MHz)	Gain	Open-source
[139]	ASIC	33.71	15	225.73	880.2	✗

Table 4.6: Comparison of ASIC implementation of BLAKE2b hash function

4.3.3 Conclusion

This previous part of the thesis gave a helpful mathematical definition of the cryptographic hash function. It was also defined why the hash function is an essential element of the digital signature process, and it was also highlighted that the bit length of the hash digest must be long enough to be efficient against attacks (like brute-force).

In order to accelerate the hash function used in the studied blockchains, hardware accelerator (ASIC) designs were compared and chosen to be embedded in the proposed IoT architecture model. The speedup of the different hardware accelerators is calculated according to the latency measured in BCM2837 ARM-based architecture. In our previous works [19], [17] and [9], the proposed accelerators provided a significant speedup; however, these designs required a significant frequency to be applied. The high frequency causes a higher dynamic power consumption of the hardware design. In order to achieve a more optimal power consumption but still achieve a significant gain, new designs were proposed (found in the literature) to be used (SHA-512, Keccak) in the final IoT hardware model.

4.4 Conclusion

This chapter described the essential cryptographic primitives that are used in blockchains. The simplified mathematical background is also given for better understanding the elliptic curve cryptography and the hash functions. The previous chapter demonstrated the impact of the cryptographic primitives in the blockchain APIs allowing the interaction with a given blockchain. One of the main goals of this chapter was to identify how hardware acceleration is possible on these primitives. ASCI and FPGA designs were searched, studied, and compared according to the literature.

It was also highlighted that it is challenging to find hardware accelerators for EC point multiplication on secp256k1 and edwards25519 curves. The hardware accelerator operating on these curves must be specified on the general prime field (of 256-bit length) or on the prime field that was not dedicated to NIST-recommended curves.

The form of the curve also has importance. The hardware accelerators defined over the general prime field can operate on any arbitrary Weierstrass form curve (including secp256k1, but excluding edwards25519). Therefore, for edwards25519 curve the hardware accelerator must allow operating on this form of the curve.

The results show that the hardware implementations that are dedicated to secp256k1 and edwards25519 can produce a high speedup, and at a considerably low frequency. It can be concluded that for today's blockchains, the hardware accelerators dedicated to secp256k1 and edwards25519 can meet all requirements to be implemented in new IoT architectures. However, if new curves of Weierstrass form would be used in blockchain technology of the future, the hardware accelerators over the general prime field are preferred to be implemented. These designs are mainly programmable for any curves of Weierstrass form. The best design choice for this purpose is the design proposed in [11].

The selected hardware accelerator ([11]) used in the final IoT model for accelerating the operations on the secp256k1 curve is an FPGA design that can achieve a four times faster execution of EC point multiplication against BCM2837 and 6.5 faster execution against more constrained ARM-based architectures. It can be noted that this design can be synthesized to obtain an ASIC, which can achieve even more speedup at the same or lower frequency.

For accelerating the ECPM operation on the edwards25519 curve the design proposed in [12] is selected to be implemented in the final IoT architecture model. This design provides a low latency at a relatively low frequency. The main drawback of the design is that it is based on RNS which means that the coordinates has to be converted to integer representation and from Jacobian to affine coordinates when the IP is called.

The chapter also detailed the hardware acceleration possibilities on the hash algorithms such as SHA-256, SHA-512, Keccak, and BLAKE2b. The objective of this part of the chapter was to find and compare ASIC implementations with the constraints of achieving a high speedup at a relatively low frequency. The frequency usage has an important impact on the power consumption, which has to be the more optimal possible.

The selected designs have an optimal frequency usage with a high enough speedup. In the literature, for any hash hardware accelerator implementation, the SHA-256 serves as a point of comparison. The basic throughput requirements of any of these implementations are around 2.2 Gbit/s. This throughput is achieved in all of the selected hash hardware implementations.

Chapter 5

Development of an IoT architecture model dedicated to blockchain applications

5.1 Introduction

One of the main objectives of this thesis is to model a specific low-power consumption IoT architecture (this architecture can also be considered as a SoC). It must be noted that the objective is to create a model of the architecture and not a design that is ready to be fabricated. Modeling SoC architectures makes sense because it is easier to test and validate the architecture in its early phase instead of verifying the final fabricated hardware architecture. Error detection in the early phase (on the model) is more cost-effective than on the fabricated architecture because it is unnecessary to fabricate the SoC repeatedly until the error is solved.

The number of requirements of a new SoC architecture increases incredibly. High computational power is required, but the power consumption has to be minimal. The architectures became more and more complex, multi-core processors are applied with more and more peripherals. Special modules are also required for new types of protocols. In modern SoC development, time-to-market pressure is increasingly present, making it more challenging to deploy a high-quality product. Development methodologies (ex. SoC V-Model [140]) can be used to save time and money during the development phase. In this development phase, the hardware and software implementations can be done in parallel [141]. When the first hardware model is ready (left wing of the ”v”, see figure 5.1), the first version of the software can be tested. From this time, the hardware and software testing and validation process can continue in parallel (right wing of the ”v”).

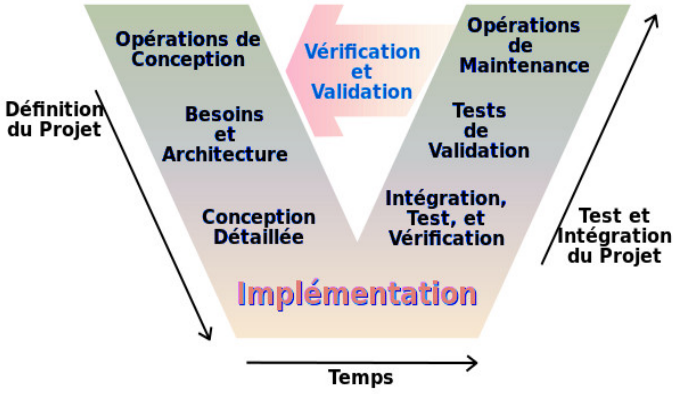


Figure 5.1: Example of V-Model methodology. The figure is adopted from [15]

The main objectives of this chapter are the descriptions of the functional and power-managed model of the proposed IoT architecture. In a first step, the chapter describes the software tools such as Quick EMUlator (QEMU), SystemC-TLM, and virtual co-simulation platforms (combining QEMU and SystemC-TLM) tools, which allow the modeling of the proposed hardware architecture.

In a second step, the chapter gives detailed information about the models of the hardware accelerator designs corresponding to the identified cryptographic primitives (modeled in SystemC-TLM) and their co-simulation with the QEMU emulated ARM-based CPU architecture. The chapter also describes the development of Linux Kernel device drivers, which are necessary to allow the API's communication with the hardware accelerator modules.

Once the first functional model is validated, the next step is to develop a power-managed architecture model. The chapter outlines the PwClkARCH SystemC library which allows the power management and power consumption monitoring of the architecture. The development of a specific Linux Kernel device driver that orchestrates the power management according to the blockchain API execution is also explained in this step.

The proposed power-managed architecture is analyzed in the last step by running the blockchain APIs on the top of the architecture. This last step of the chapter also provides results about the architecture energy consumption and execution time. The first results of the power-managed architecture can show the need for a posteriori work on the architecture to achieve more efficient power management, or it can also prove the completion of the power-managed architecture.

5.2 Selected software tools to model the architecture

In the previous chapter (p.91 and p.97) hardware accelerators were chosen in order to accelerate the essential cryptographic functions used in the blockchain APIs. Adding these design models can be done thanks to hardware description languages. The VHDL/Verilog languages could be used. However, the designs would be complex, and they do not allow hardware/-software co-simulation. SystemC-TLM [16] hardware description language eases the implementation of application-specific hardware. Thanks to the application-specific hardware description feature, this programming language meets one of the main objectives of this thesis work, thus creating a hardware architecture model that can accelerate blockchain applications. Details on SystemC hardware description language are provided on page 109.

Modeling a complex processor architecture is possible using VHDL or Verilog hardware languages. However, implement all of the behaviors and all of the processor components is a challenging task that requires much time and experience. Running operating systems on such a model is also challenging. Therefore, there is a solution in which the behaviors and components of the given complex architecture can be emulated. Using QEMU [34] or GEM5 [142] avoid making complex architecture designs. These tools can emulate architectures such as ARM, x86 and SPARC for example. In the proposed hardware model, QEMU (detailed on p.112) is used to emulate the ARM-based CPU architecture. QEMU was chosen for the CPU emulation because several research results provide tools for combining QEMU with SystemC-TLM.

It must be noted that QEMU and SystemC are two completely different environments. The proposed IoT architecture model is based on these two different simulation environ-

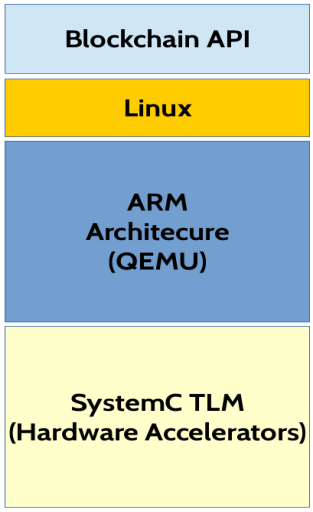


Figure 5.2: The simplified scheme of the modeled architecture.

ments, which must be synchronized in order to operate together.

After the functional architecture model is implemented and validated, dedicated power management is deployed for this proposed architecture. That could be done thanks to the PwClkARCH (Power Clock Arch) SystemC library allowing the visualization and management of the architecture's power consumption. This tool is discussed later (p.112).

The simplified scheme of the modeled architecture with QEMU and SystemC-TLM co-simulation, running a Linux Operating System which is able to execute blockchain APIs (depicted in figure 5.2).

5.2.1 SystemC-TLM

In the previous section (see sect. 5.2), SystemC was presented as a hardware description language. SystemC is an open-source C++ library IEEE 1666 standard, that is popular in the industrial domain in the early phase of development of the hardware architectures.

The figure 5.3 represents a comparison between the hardware description languages. It can be noted that SystemC covers a large part of abstraction layers, from the architecture until the RTL (Register Transfer Level).

Unlike in hardware description languages such as VHDL and Verilog, in SystemC, the definition of each bit, each component, and each cycle is not a mandatory. Thanks to this advantage of SystemC, a significant amount of time can be saved in the very early phase of the architecture's deployment. In this mentioned phase, only the basic definition and operation of the architecture would be verified (functional verification).

5.2.1.1 Basic components of SystemC

Modules are the fundamental elements of SystemC designs. A module is a class similar to a class that can be deployed in C++. A SystemC design can contain an unlimited number of modules that can depend on each other respecting a hierarchy. As a module is a class, it can be instantiated as an object in other modules. Thus modules can use other modules' features.

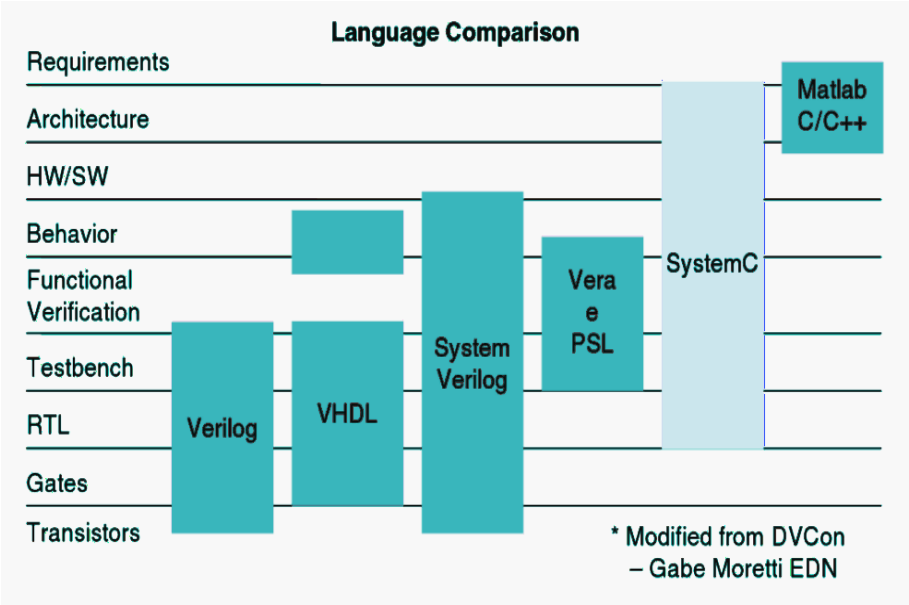


Figure 5.3: Comparison of Hardware Description Languages, figure retrieved from [16].

A module can contain processes which can be *SC_THREAD* or *SC_METHOD*. *SC_THREAD* is run once when the SystemC simulation is started (*sc_start()* function is called in *sc_main*), and it can wait on events using the *sc_event* class. An event can be triggered thanks to the *notify()* function. Unlike *SC_THREAD*, the *SC_METHOD* is sensitive to inputs that must be defined when the method is declared. Another difference is that *SC_METHOD* is executed only when an input signal among its sensitivity list (that can be specified while the method's declaration) is present, and it cannot include the wait function.

The modules can also contain ports used to connect modules among them via signals to send various types of data. The signal can be seen as a communication channel between the modules. Clock signals can also be connected to ports of the modules, for example. The communication between modules can also be established by using TLM transactions that are detailed later (p.111).

5.2.1.2 The notion of time in SystemC

It is essential to understand the difference between the so-called wall clock time and the simulated time. The wall clock time is the time that elapses in the host machine while the SystemC simulation is running. The simulated time is the time that is seen by the modules of the design, thus the time that is modeled in the design. It can be noted that timed functional modeling is also possible with SystemC because the notion of time can also be implemented in the design. The previous section mentioned that a *SC_THREAD* process of a module could wait for an event that can trigger some types of operations. This process is not limited to event waitings. It can also wait a certain amount of time thanks to the *wait()* function, with an amount of time as input. After the specified amount of time is elapsed, it is added to the simulated time, which unit is in the range of femtosecond to second.

5.2.1.3 TLM 2.0 - Transactions

Transaction Level Modeling is a standard that is used in SystemC. When a design includes TLM, the design is modeled in SystemC-TLM [143]. The TLM 2.0 is used to establish communication between the modules via function calls. One of the main objectives of TLM 2.0 is modeling memory-mapped busses that can interconnect other modules with each other.

Transaction Level Modeling uses transactions, also called generic payloads, for communication. Previously it was mentioned that in SystemC the modules could be connected via a signal between their ports. Unlike in SystemC, in SystemC-TLM modules can also contain a socket object that enables direct communication via the transaction function calls (or the so called core interface). The communication via these sockets is based on the transfer of a data structure filled in the module's process. The module that wants to forward the data via TLM must contain an Initiator socket connected to a target socket of another module (depicted in figure 5.4). The module containing the Initiator socket calls the *b_transport* or *nb_transport* method to forward the generic payload of the transaction (this payload is also depicted in figure 5.4).

TLM 2.0 allows to use two time models, the loosely and the approximately timed model.

The loosely timed model is implemented with the blocking interface (*b_transport*) using simple transaction communication, which means that the target module gives a direct response to the initiator request. The approximately timed model is implemented with the non-blocking interface (*nb_transport*), in which transactions contain phases (e.g., a handshake phase between the initiator and target, before the exchange of the adequate data). This communication mode realizes more accurately the hardware communication of a BUS, for example; however, this type of model requires a more complicated code implementation. It can be noted that the loosely timed model is used in the first phase of the development for verifying the correct functioning of the design. The approximately timed model is better to use in the second phase of the development in which performance analysis can be performed on the architecture's communication [143]. The proposed IoT architecture model of this thesis is developed by using the loosely timed model of SystemC-TLM.

The payload contains the command (read or write), the address from/to the read/write is done. The data pointer, length and also the response of the target socket.

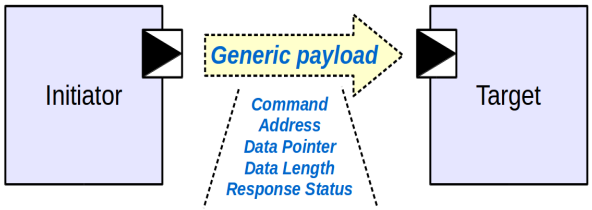


Figure 5.4: TLM *b_transport* socket communication

It is also important to note that every initiator socket must be connected to a target socket. TLM allows a faster simulation by using the temporal decoupling method. When an Initiator uses the *b_transport* function, in which a time delay is included, the Initiator increases its local time. As the Initiator manages its own local time, it can be different from the SystemC's simulation time. Time synchronization is needed between the local simulation and the global time. For doing so, a global quantum is set to a constant value. When the Initiator's local time reaches the global quantum value (for example, 1 us), synchronization is required with

SystemC simulation time. The synchronization is done by calling the $wait()$ function with the global quantum value (this mechanism is hidden from the user). In the loosely timed model the synchronization occurs more frequently according to a small quantum that causes a more accurate but more extended simulation (wall clock) time. A more significant quantum can decrease the simulation time (wall clock), nevertheless, it also causes a less accurate simulation.

5.2.2 QEMU

Quick EMUlator (QEMU) [34] is a hardware emulator that enables the emulation of CPU architectures containing or no devices such as serial ports, VGA display, peripherals, etc. This open-source hardware emulator is based on C, and it uses dynamic binary translation in order to translate the target architecture's instructions into the host architecture's. The target architecture is the one that aimed to be emulated, and the host architecture is the one that runs the emulation. The target architecture is also called guest architecture.

QEMU is independent of the operating system (OS). The emulation can be executed on Linux, Windows, Mac OS X, and the target architecture can execute any operating system (even micro-operating systems such as uCos II, for example).

Today QEMU allows emulating architectures such as ARM, x86, MIPS, Microblaze, Risc-V, PowerPC, and SPARC (around 200 different architectures are available to be emulated). QEMU allows emulating embedded CPU architectures, and it also makes it possible to add new modules into the existing architectures or even modify them. All of these features ease the development of brand new architectures.

It can be noted that QEMU is an instruction accurate emulator. The host architecture executes the target architecture's instructions as fast as possible. Therefore the target CPU's frequency is different from the real CPU that is emulated. It is a piece of evidence because if the host architecture operates on a lower frequency than the target architecture's, it would be physically impossible to produce this same higher frequency.

It must also be noted that QEMU has three main clocks [144]. The *host_clock* is used to get information on the time of the host system. The standard *gettimeofday()* C function is used to determine this time. The *rt_clock* uses the *clock_gettime()* function on the *CLOCK_MONOTONIC*, providing the number of nanoseconds that have passed since the system's boot. The *vm_clock* (virtual machine clock) is used as the reference for the target system. Unlike *rt_clock* the *vm_clock* is reset to zero when the virtual machine stops. However, it also calls *clock_gettime()* to obtain the time. It is possible to specify a parameter *-icount n* that increases the time with 2^n nanoseconds when a guest instruction is executed. For example, *-icount 3* increases *vm_clock* with 2^3 (8) nanoseconds every time an instruction is executed. In that case, the virtual processor frequency would be equal to 167 MHz.

5.2.3 PwClkARCH SystemC library

PwClkARCH C++/SystemC-TLM library was implemented in the LEAT laboratory, in which this thesis contribution was also born. This framework provides a power management deployment and power consumption visualization/verification of the functional SoC design modeled in SystemC-TLM language. The idea of the PwClkARCH comes from the UPF (IEEE standard 1801, Unified Power Format) [37] tool. Thanks to UPF, it is possible to

define Power Domains, power switches, supply networks, and other components that can be associated with the entities or IPs (Intellectual Properties) of the functional SoC design. Therefore, a power architecture is designed for the functional SoC model at RTL (Register Transfer Level).

In order to make more accessible the development, achieve a faster simulation, and designing a power architecture at Electronic System Level, PwClkARCH provides power-controlled model designing at the Transaction-Level (TL) [38]. PwClkARCH allows managing dynamic power consumption by using Dynamic Voltage and Frequency Scaling (DVFS) and clock gating techniques [145] [43]. Static power consumption can be managed using the gate power technique by switching the supply voltage, resulting in zero power consumption. The equations of the dynamic and static power consumption are given below (eq. 5.1 and 5.2).

$$P_{dynamic} = \alpha * C * V^2 * F \tag{5.1}$$

The dynamic power consumption is produced by the Intellectual Property's (IP) supply voltage (V), the clock frequency (F), and the capacitance (C), which is related to the CMOS technology that was used to fabricate the given SoC. This capacitance is the sum of the interconnect, gate, diffusion, and Well capacitance [146]. The α parameter is the portion of an IP that works at the given time. When the IP is accessed for writing or reading, only its registers are in the active phase, which means that the α is around 10%, for example. However, α can be equal to 95% or 100% when the IP is in the computing phase. This parameter can also be called workload when the IP is a CPU. Dynamic power consumption can be optimized by scaling (changing) the frequency or/and the voltage (DVFS).

$$P_{static} = \frac{V^2}{R_{leakage}} \tag{5.2}$$

The static power consumption (eq. 5.2) is composed of the supply voltage (V) and the leakage resistance ($R_{leakage}$) that causes the leakage current in the CMOS transistors.

These power optimization techniques can be applied thanks to the features of PwClkARCH that are described below.

Figure 5.5 represents an example of how the PwClkARCH can be used to realize a power architecture starting at a functional level. Each functional model of IPs (CPU, SRAM, DCT) correspond to a DE (Design Element). The declaration of correspondences are done in the top-level or main program. The DEs are declared in a given Clock Domain (CD) and a Power Domain (PD). The choice in which PD and CD a given DE is declared depends on the developer.

The IPs (or DEs) included in a CD are supplied by the output frequencies that are provided by a Digital Phase Locked Loop (DPLL) unit.

The input clock of a DPLL is a reference clock frequency. The output frequency of the given DPLL is connected to the Clock Manager (CM), which distributes the corresponding frequencies (clock signals) to the DEs. This Clock Manager can also apply a clock division and/or gating on the clock frequency that was generated by the DPLL and it is distributed to

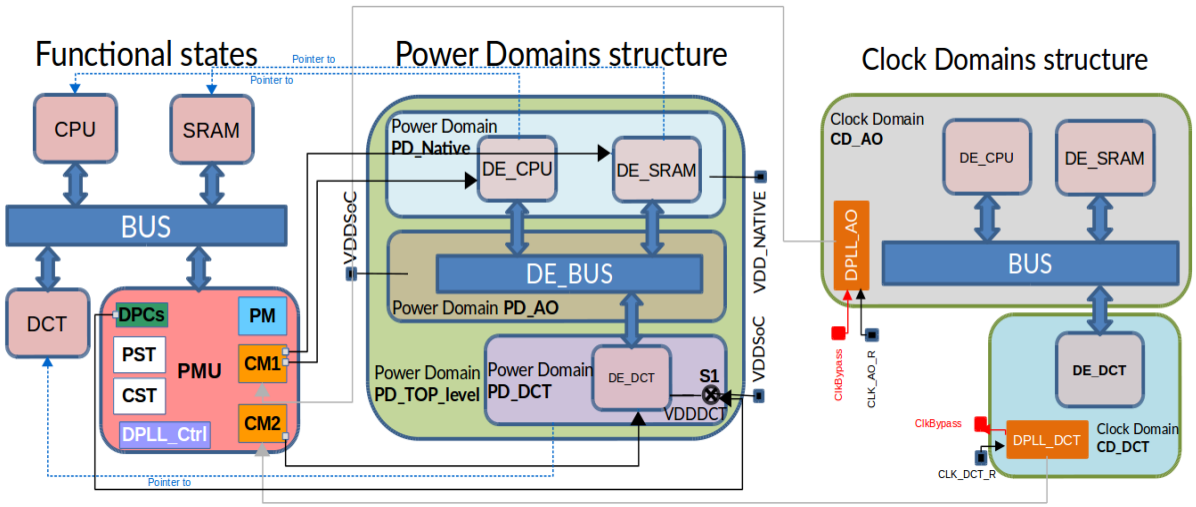


Figure 5.5: PwClkARCH deployment over a functional architecture to obtain a power architecture.

DEs. Clock scaling and gating occur when the Power Manager (PM) sends a request to the CM that applies the given division ratio or switch on/off the clock frequency.

The IPs of the same power domain (PD) have the same supply net, which can be switched (using a switch component) or divided thanks to the power controller units. The Power Management Unit contains the CM, the power domain controllers (PDCs), and the PM. This unit is connected to the functional model in order to allow the power management over the functional model. The PM implements the power management strategy. The information about the strategy is declared in the form of a Power State Table (PST) containing the state of the power supply corresponding to the power domains (see example in figure 5.6). Clock State Tables are also declared with the ratios of division over the input clock of the corresponding Clock Domain (see example in figure 5.6). An initiator module can call the power management strategies (for example, CPU executing an API) in the form of Operating Performance Points (OPPs) that are also declared in a so-called OPP table. A line of the table is composed of the division/multiplication ratios of every DPLL clock frequencies, and the index i,j correspond to the line numbers to be called in the PST (i) and line j in the CST (see figure 5.6).

CST	CM1_DE_AO	CM2_DE_DCT
DCT Active	1	2
DCT Idle	4	0

PST	VDD_SoC	VDD_NATIVE	VDD_DCT
DCT On State	ON_L	ON_H	ON_H
DCT Off State	ON_H	ON_H	OFF

OPP	DPLL_AO	DPLL_DCT	Index
DCT Active	8/4	2/1	(1,1)
DCT Idle	4/8	1/1	(2,2)

ON_H = 3.3 V
ON_L = 1.2 V
OFF = 0 V

Figure 5.6: Example of OPP, CST, and PST tables. In this example, the OPP and CST tables contain two states, *Active* and *Idle*; The PST contain an On and Off state (the switch *S1* is in on/off state) in this example. In fact the tables can contain unlimited number of states.

It must be noted that all of the IPs must contain observer objects of PwClkARCH that

are used to observe the IPs' activity in order to inform the PMU, that the IP computes and to calculate the power consumption.

5.2.4 Virtual Platforms: Combination of QEMU and SystemC

Previously, SystemC-TLM and QEMU were detailed. It can be observed that QEMU and SystemC are two completely different tools operating with different time constraints and environments. Until QEMU emulates CPU architectures as fast as the host system allows, SystemC-TLM is based on two times (wall clock and simulated time). The simulated time is also synchronized with the global quantum to enable the advantages of TLM communication.

It can be noted that bringing together QEMU and SystemC-TLM is a real challenge. The synchronization of these two systems is also a challenging task. This section describes three virtual platforms that enable implementing architectures that are based on the combination of QEMU and SystemC-TLM.

5.2.4.1 Hiventive Platform

This platform was invented by the French start-up Hiventive. The open-source version of this platform [147] realizes a model of BCM2837 architecture that is also used in Raspberry Pi 3 B+ models. The QEMU emulates a quad-core ARM Cortex-A53 CPU architecture with the associated RAM and SD Card Control. The peripherals of the CPU, such as the UART, Interrupt Controller, and GPIO modules, are implemented in SystemC (see figure 5.7). The communication between the QEMU domain and the SystemC modules is realized thanks to a QMG2SC interface that can transform the QEMU instructions into SystemC-TLM socket communications (*b_transport*). It can also be noted that the *b_transport* calls are "hidden" in a register class, which can be mapped in the memory. Callbacks can be implemented on the registers that notify the read or write access on the given register. The use of these registers simplifies the implementation phase because the read and write operations on a register can be done the same way as it can be done in C++ arrays.

The synchronization of QEMU with SystemC is done in a particular way in which the QEMU is the Master of the simulation. That also means that SystemC Kernel occupies only the SystemC modules' synchronization and not synchronizes the overall SystemC-QEMU system. This type of synchronization procedure can allow for a faster simulation. However, QEMU is not sensitive to the time elapsed in the SystemC module. Thus, when a *wait(time)* is applied in a SystemC module, the simulation is stalled.

Previous works [17] and [19] giving a contribution for this thesis work also used this platform. In these works, SHA-256, SHA-512, Keccak-256, and Salsa hardware accelerator modules were added to the existing BCM2837 model. These works have shown that the APIs can call the SystemC modules via Linux Kernel device drivers; hence the untimed functional model can work correctly.

It was already mentioned that dedicated power management could be applied on the top of the modeled architecture. The power measurement or management is applied during a given time slot. Thus the notion of time is a critical element when deploying the power management.

This platform does not allow timed functional modeling. Thus, today it is not possible to model a power-managed architecture with this platform.

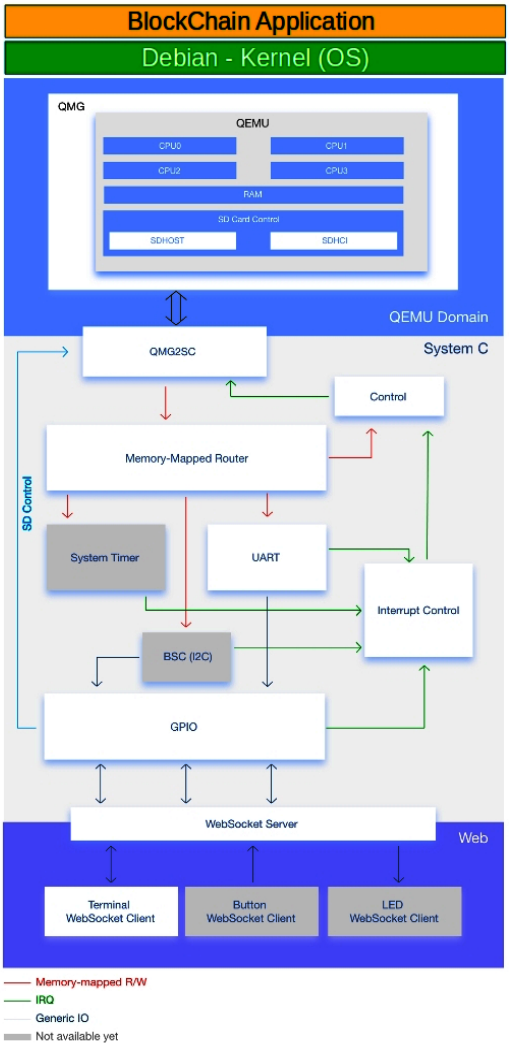


Figure 5.7: BCM2837 architecture modelled by Hiventive QEMU-SystemC platform. The figure is adopted from [17]

5.2.4.2 QBox

Unlike in the Hiventive platform, in QBox, the ”Master” of the simulation is the SystemC kernel [148] [141] [144]. The QEMU is integrated into a SystemC module (QBox’s name comes from QEMU in Box), which means that the QEMU and the other SystemC instances have their own thread. QEMU and SystemC-TLM can be synchronized thanks to the global quantum, and the fact that QEMU is treated as a SystemC-TLM module. QEMU is wrapped into SystemC-TLM 2.0 interfaces (TLM2C wrapper) in order to be treated as a SystemC module. While simulating QEMU the *-icount* option is used in order to determine the local time by counting the number of instructions.

It is worth noting that QBox library contains specific CPU cores in C, and QEMU is compiled into QBox shared libraries. Likewise, the Hivetive platform QBox also uses a register class to hide and simplify the use of *b_transport* socket communication. Contrarily to the Hiventive platform, QBox is time-sensitive which means that a *wait(time)* can be applied in the SystemC modules, and it does not cause a simulation stall.

According to analysis and implementations, this platform can work correctly as a timed functional model. Power management can also be implemented thanks to PwClkARCH library (example in [149]). The main disadvantage of this model is that since 2019 the project is no more open-source, and there are no available CPU cores anymore.

5.2.4.3 SystemC-TLM 2.0 Co-Simulation (Xilinx)

This co-simulation of QEMU with SystemC-TLM is an open-source project allowing the simulation of Xilinx modified QEMU and modules written in SystemC-TLM [35] [36]. This tool was developed to co-simulate Zynq-based and Versal ACAP products. Figure 5.8 presents the co-simulation environment. QEMU emulates the Processing System (PS).

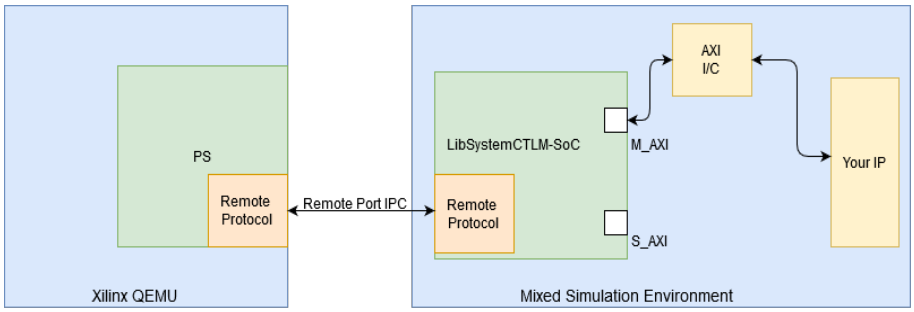


Figure 5.8: Co-simulation environment, figure is retrieved from [18]

The Programmable Logic (PL) is the SystemC environment in which the SystemC-TLM modules can be deployed. A SystemC-TLM 2.0 wrapper (libSystemCTLM-SoC) is used to encapsulate the PL's modules for allowing the Remote-Port connection between the PS and the PL. The Remote-Ports are used for communication and the time synchronization between the QEMU and the SystemC-TLM simulations. Using libSystemCTLM-SoC and Remote-Ports enables the PS (QEMU) to be treated as a SystemC module similarly to QBox, however in the case of libSystemCTLM-SoC the QEMU is the "Master".

QEMU instantiates several remote port devices like memory masters and wires. The Remote-Port packets contain timestamps that are sent by QEMU to the SystemC side when memory transactions or wire updates occur and QEMU also sends periodic synchronization updates.

Basically, QEMU waits for the SystemC side on memory transactions. When using *-icount* option in QEMU, lockstep can be achieved in QEMU when a *wait(time)* is applied on SystemC side. Thanks to this feature, the QEMU and SystemC processes can be synchronized, and QEMU emulation is sensible to time. The time sensibility allows the "Timed Functional" modeling and the simulation will not stall when the SystemC applies a *wait(time*).

It can also be noted that libSystemCTLM-SoC uses AXI ports to connect the developed SystemC TLM IPs. This platform's two most significant advantages are that this Co-simulation is time-sensitive, the time can be simulated in the SystemC simulation (*wait(time)* works). The second advantage is that this platform is open-source, and its community is active. The code snippet below shows the instantiation of QEMU with the global quantum (*sync-quantum*) and the *-icount* parameters.

```
#!/bin/bash

qemu-system-aarch64
        -M arm-generic-fdt-7series \
        -m 2G \
        -machine linux=on \
        -kernel uImage \
        --initrd umy_ramdisk.image.gz \
        -dtb system-top.dtb \
        -machine-path /tmp/machine-aarch64 \
        -icount 1 \
        -sync-quantum 100000 \
        ...
```

Listing 5.1: Code snippet of QEMU instantiation for the emulation of ARM platform

It must be noted that the Xilinx QEMU guide suggests that the *icount* parameter has to be set to 7 when the Zynq-7000 device is co-simulated. However, with such a configuration, the boot time is too significant when using a Linux/arm 5.4.0 Kernel, around 10-15 minutes. When applying an *icount* of 1, the boot time is around 30 seconds, and the simulation accuracy can remain similar to the simulation's using *icount* 7. The simulation results of this thesis work were performed while the *icount* parameter is set to 1.

It is worth noting that in the case of the co-simulation of Zynq UltraScale+, it is possible to run more than one QEMU instances. The first instance are intended to emulate Cortex-A53s and the Cortex-R5 CPUs. The other is intended to emulate a microBlaze microcontroller. In this context, the microBlaze plays the role of a Power Management Unit. In the documentation, it is not specified, but theoretically, the first QEMU instance is the master of synchronization between the SystemC modules and the QEMU instances.

5.2.4.4 Comparison of the Co-simulation platforms

The following table (Tab. 5.1) compares the three studied co-simulation platforms allowing to combine QEMU and SystemC-TLM simulations.

	Synchronization's Master	DTS modification	Applying SystemC-TLM *wait(time)*	Functional Model Realization	Power Controlled Model Realization	Open-source Code
Hiventive Platform	*QEMU*	*Required*	✗	✓	✗	✓
QBox	*SystemC Kernel*	*Required*	✓	✓	✓	✗
TLM Co-Sim (Xilinx)	*QEMU*	*Not Required*	✓	✓	✓	✓

Table 5.1: Comparison of the Co-simulation platforms

The first column of the table shows if QEMU or the SystemC Kernel is the "master" of the synchronization. In the comparison each of the emulated architectures runs a Linux OS.

Linux boot process requires Device Tree Blob, a compiled version of a Device Tree Source (DTS) containing the architecture description that runs Linux OS. Hiventive Platform and QBox require the modification of the DTS, which means that the SystemC modules' parameters (address etc.) have to be added to the DTS.

Except for the Hiventive Platform, a *wait(time)* can be applied in the SystemC modules without the stall of the simulation. This mutually means that if the notion of the time cannot be applied, then the Power Management neither be implemented. In all of these platforms, the untimed functional model works correctly. The advantage of Hiventive Platform and TLM co-Simulation is that they are both open-source projects.

As SystemC-TLM 2.0 Co-Simulation (Xilinx) is an open-source platform with all of the characteristics allowing the implementation of a Power Managed model, this platform was selected to implement the proposed IoT architecture model of this thesis work.

5.3 The proposed architecture with dedicated hardware accelerators

The previous chapter discussed about the hardware accelerator implementations which were selected in order to accelerate the cryptographic primitives that were identified in chapter 3 (p.45).

In the proposed architecture model, the implementations of the cryptographic primitives are modeled in SystemC. These SystemC modules of the cryptographic primitives are connected to a 32-bit data BUS or Interconnect that is also implemented in SystemC (BUS is provided by this Co-Simulation Xilinx tool). The objective of this BUS is to facilitate the communication between the SystemC modules and the CPU architecture emulated by QEMU. This BUS is also connected to the libSystemCTLM-SoC (see figure 5.7, p.116) wrapper to enable *b_transport* communication between the SystemC modules and the QEMU emulated CPU architecture. Figure 5.9 represents a simplified representation of the proposed architecture in which the wrappers are hidden, and the QEMU emulated ARM CPU architecture is directly connected to the BUS. In this figure, the yellow modules correspond to the implementations of the cryptographic primitives developed in SystemC-TLM. The models of the cryptographic primitives: SHA-256, SHA-512, Keccak, BLAKE2b an EC point multiplication for the secp256k1 and edwards25519 curves are detailed in the following sections.

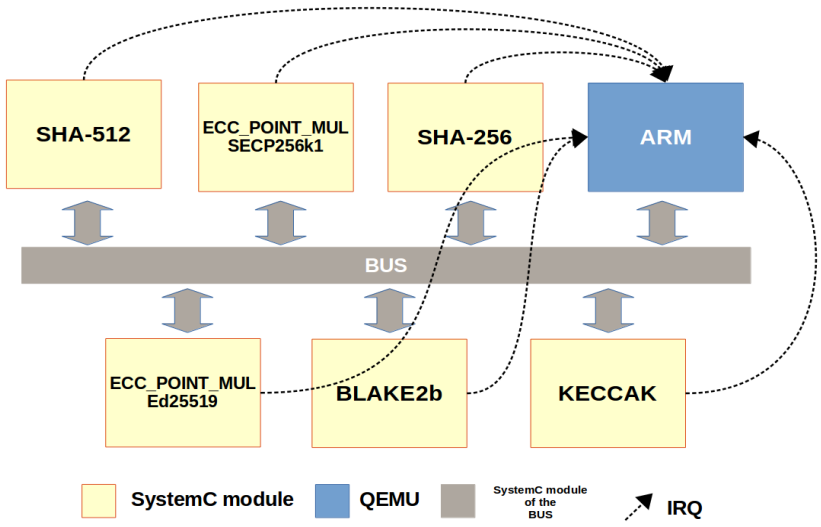


Figure 5.9: The proposed IoT architecture model

The blue entity represents QEMU emulated ARM CPU architecture, which is explained

in more detail later.

It must be noted that all of these modules use TLM-2.0 socket communication in Loosely Timed mode, in which the ARM CPU module is the Initiator. That means that only the ARM CPU model can initiate read and write accesses from/to the other modules (cryptographic primitives). Probably this is the best example of how the TLM-2.0 socket communication can be applied and why TLM-2.0 socket communication requires an Initiator and client modules.

The emulated ARM architecture runs a Linux operating system that allows executing the API communicating with a given blockchain. The API contains several cryptographic primitives that can be performed faster if these primitives are executed by the given hardware accelerator instead of running it on the CPU. When a CPU should run a cryptographic primitive, the input data of the given primitive has to be sent to the module which performs the primitive. When the data is written to the given module (the data can be written into a register of the module), it can start to perform the cryptographic primitive on the written data (for example, that SHA-256 module takes 512-bit data as input and performs its hash). After the computation of the primitive, the CPU can read the result from a register of the module.

It must take into account that the CPU does not know when the result is available. A dedicated register can contain a value that specifies if the module is busy or in a ready state. The CPU could read this register until the module changes its busy to ready state. However, this approach has two main drawbacks: first, the CPU has to perform read accesses until the module is busy, slowing down the communication over the BUS. Delbergue *et al.* [148] have shown that looping IO access in the case of QBox has caused a significant slowdown of the simulation (host time) because an IO access could take 40 ms. The same feature can be observed in the case of SystemC-TLM 2.0 Xilinx Co-Simulation.

Second, the proposed architecture runs a Linux operating system. To access the hardware from an API, Linux Kernel device drivers are required for performing IO access. Until a module is busy, other functionalities of Linux cannot be accessed because the API's execution is stalled at the device driver level.

Therefore, Interrupt ReQuest (IRQ) presented with arrows in figure 5.9, can be applied to avoid the stalled task problem. Each module has its own IRQ that is connected to the CPU architecture. Using IRQs avoids making looped read access of the CPU. When a module finishes its computation, it generates an IRQ for the CPU, which can now read the result.

Here this procedure is simplified. A Kernel device driver has to subscribe to the module's IRQ, which also means that until the device driver waits for the interruption, the device driver task becomes a background Linux task, and the other functionalities of Linux are available again. When the IRQ arrives, the device driver task wakes up and reads the result on the corresponding module. It must also be noted that the wake up and IRQ waiting procedures cannot be done in hard real-time because Linux is not a real-time system. This also means that the wake up and IRQ handling of Linux can cause negative effects on the architecture overall energy consumption. These effects would be described later (p.148). The Linux device drivers are also detailed later (p.125).

The proposed IoT architecture model represented by figure 5.9 is a functional model including the notion of time. It must be noted that this architecture model does not include power management. The power-managed architecture is described later (p.132).

5.3.1 QEMU emulating ARM-based architecture

Today SystemC-TLM 2.0 Co-Simulation (Xilinx) provides three demo examples[1] in which QEMU[2] can emulate two ARM-based and one Risk-V CPU architectures. Each architecture allows communication with custom SystemC-TLM 2.0 modules.

The ”zynq_demo” is intended to co-simulate a Zynq-7000 architecture, including a dual-core ARM Cortex-A9 32-bit processor with the corresponding peripherals such as GPIO, I2C, SPI. The Zynq-7000 architecture also contains a 28nm Artix®-7 FPGA accessible by the CPU.

The architecture Zynq-7000 which is emulated in a single QEMU instance is depicted in figure 5.10.

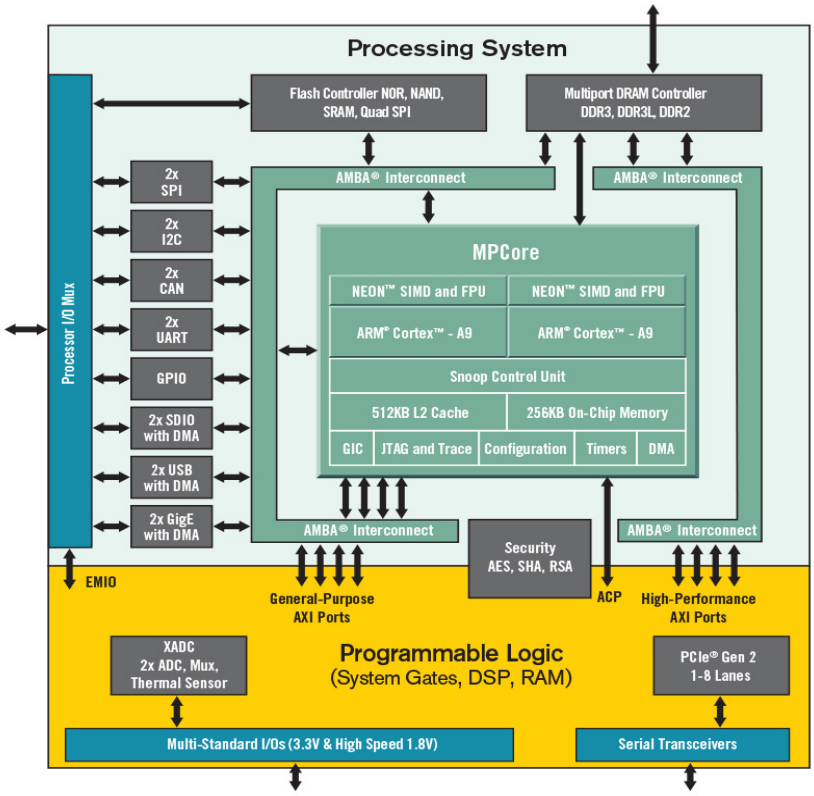


Figure 5.10: Zynq-7000 architecture

The ”zynqmp_demo” was invented to co-simulate Zynq UltraScale+ architecture, which includes a dual or quad-core ARM Cortex-A53 64-bit processor and an ARM Cortex-R5 architecture with their peripherals. It should be noted that a second QEMU instance can be run to emulate a microBlaze microcontroller that can be implemented on the FPGA part of Zynq UltraScale+ architecture. The rule of the microcontroller is to apply dedicated power management by changing the frequency and, in some cases, the voltage in Cortex-A53 and Cortex-R5. This unit also allows switching off the supply voltage of the Cortex-R5.

The ”riscv_virt_lmac2_demo” serves for the co-simulation of a Risc V CPU architecture with a SystemC model (corresponding to the architecture that can be developed in an FPGA

[1] Demos are available at https://github.com/Xilinx/systemctlm-cosim-demo.

[2] This version of QEMU is modified by Xilinx in order to allow communication with custom SystemC-TLM 2.0 modules (the wrapper and Remote Ports are added), available at https://github.com/Xilinx/qemu.

cart).

In the proposed architecture, ARM Cortex-A9 CPU architecture (used in iPhone 4S, for example) was chosen to be emulated by QEMU. This CPU is the core of the Xilinx Zynq-7000 family that is still popular in the research domain because this architecture allows implementing designs in the FPGA and access them from the CPU. This CPU architecture is less modern than the ARM Cortex-A53. However, in the simulation environment, implementing the architecture with QEMU and SystemC is more straightforward. It should be noted that with slight modification in the BUS connection, the proposed architecture can also work with the ARM Cortex-A53 CPU architecture.

This thesis work also contributes to the master branch of the demo GitHub ("zynq_demo") about some specific pieces of information to ease the interactions with IRQs between the emulated ARM CPU and the SystemC modules[3].

5.3.2 SystemC hardware accelerator modules

SystemC is a language that enables high-level hardware modeling. The ASIC or FPGA implementations of the cryptographic hardware accelerators are low-level modeled architectures. As the high-level modeling does not require the design of every single bit operations, the hardware accelerator implementations described in the previous chapters are slightly simplified. That means that not every signal (represented in 1 bit) is modeled. Only essential elements are modeled, such as the input-output buffers and the signal or register that commands the IP to compute. The buffers' cells and the control register are accessible thanks to specific addresses. The address offset between the control register and the buffers' cells is 4 bytes.

The CPU cannot do write and read accesses consecutively, because it is impossible to read the result until it is not computed. Therefore, an IRQ signal indicates that the IP is not busy (finished the computation), so the CPU can do a read or write access again.

Every module of the modeled hardware accelerators and their basic functioning can be modeled as follows (see figure 5.11): the module contains at least a 32-bits word length input and output buffer. The length of the buffer depends on the input and output length of the given cryptographic function (e.g., the SHA-256 function has 64 bytes length input, thus the input buffer corresponds to sixteen cells, each of which is 32 bits long). When the CPU finishes writing to the input buffer (in some cases the out buffer is an in-out buffer, because the initial hash state can also be set to this buffer), it must inform the IP to start computing. Therefore, a control register is deployed whose content must be set to 1 when the IP can start computing. When the control register value toggles to 1, an event notification is generated (*start_compute.notify()*). The thread that includes the computation process of the IP is blocked until this event is not notified (*wait(start_compute)*). After the notification, the thread can continue its execution, and the IP's computation can be done.

It can be noted that the calculation of the IP is done thanks to the cryptographic function, which would have been called in the API executed by the CPU if it was not accelerated. For simulating the latency of the hardware accelerator, the thread of the computation implements a *wait(time)* in which the time is equal to the hardware accelerator's latency. The latencies of the given hardware accelerators can be found on page 91 and page 101.

[3]The issue is available at: https://github.com/Xilinx/systemctlm-cosim-demo/issues/10

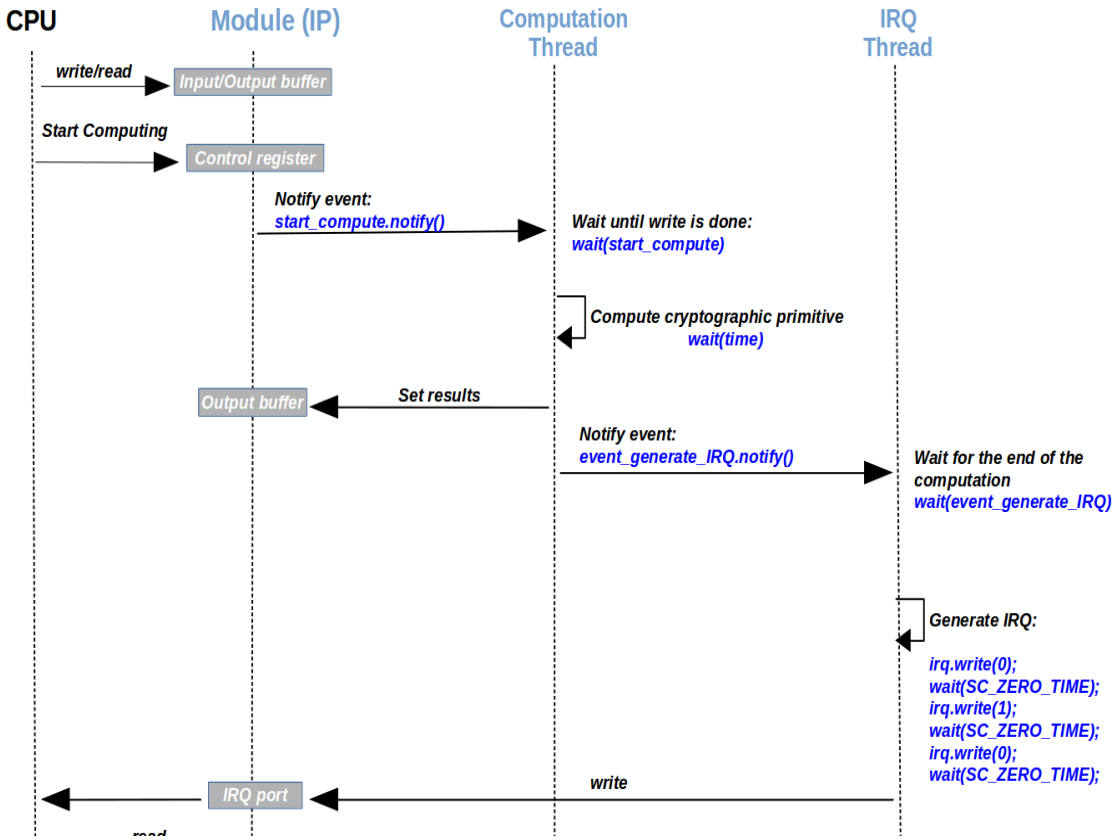


Figure 5.11: Basics of modules functioning. *The computation Thread* and the *IRQ Thread* take part in the Module.

When the computation is done, the next step is to fill the output buffers with the results (e.g., SHA-256 output buffer length is 32 bytes = 256 bits).

The CPU cannot read back the IP's results until the IP does not inform the CPU that the computation is performed and the results are ready to be read. When the output buffers are filled with the results, an event is notified (*event_generate_IRQ.notify()*) in order to generate an IRQ for the CPU. A second thread is instantiated to generate this IRQ that corresponds to a rising edge on the IRQ port of the module. The IRQ is not generated until the *event_generate_IRQ* SystemC event is not notified (*wait(event_generate_IRQ)*).

It can be concluded that figure 5.11 represents the basic functioning of the modeled hardware accelerators. It can be noted that between the modeled hardware accelerators modules, there may be slight differences; however, their logic of functionality is the same.

5.3.3 Cryptographic Hash modules

The previous subsection described the basic functioning of the SystemC modules modeling the cryptographic accelerators. The subsection also mentioned that slight differences could occur among the modules.

The acceleration on the hash functions is applied in the core of the hash function (so called *Compression function*), which means that the hash creation on a message chunk is accelerated. The CPU writes the input message chunks to the given hash module's input buffer until the last chunk of the input message. The input buffer length corresponds to the

length of one chunk of the data to be hashed (the division of the input message into chunks is explained on page 99). The CPU writes the input message chunks one by one to the input buffer. After a chunk is written an internal hash is computed and stored in the output buffer.

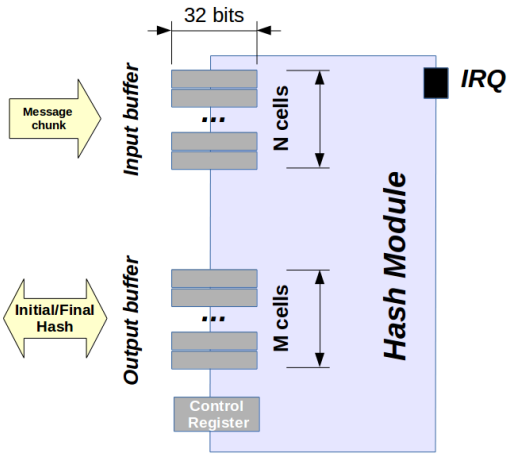


Figure 5.12: Generic representation of a hash module. The number of input buffer cells is calculated as: N = message chunk's bit length / 32 bits. The number of output buffer cells is calculated as: M = hash digest's bit length / 32 bits.

The hash value of the last chunk corresponds to the final hash value representing the hash of the input message. The buffer length corresponds to the hash length that the given hash function provides (see page 99). The CPU performs a read access to the output buffer only if the last message chunk is hashed. That also means that the internal hash value for the last message chunk is set to the output buffer.

In fact the IP is modeled in a way that the output buffer contains the internal hash value. When the final hash was read from the output buffer the IP reset the output buffer to the default initial hash value.

In the previous section, it was mentioned that the output buffer can be accessed for writing in some cases. In figure 4.8 of chapter 4 (p.99), H_0 represents the initial hash value or initial state that is required for hash computing. In general these values are constant that are required by the given hash function. In the case of the SHA-256, SHA-512, and BLAKE2b and Keccak, the initial hash value (H_0) is stored in an internal buffer of the IP. In special cases these default values can be modified. In the proposed IP models initial hash value can be written to the output buffer, that would be stored in the internal buffer.

Figure 5.12 represents a generic model of the hash modules to understand these hash modules' components better. The message chunk size of the studied hash functions is 512, 1024, 1088 and 1024 bits accordingly to SHA-256, SHA-512, Keccak-256, BLAKE2b. The hash digest sizes are 256, 512, 256, and 256, according to SHA-256, SHA-512, Keccak-256, BLAKE2b.

5.3.4 EC point multiplication module

The elliptic curve point multiplication (using the secp256k1 and edwards25519 curves) is described as $Q = k \times P$, where k is a 256 bits scalar value, P is the generator point of secp256k1 and edwards25519 curves. The chosen implementation (design proposed in [11])

of the point multiplication can realize point multiplication over any Weierstrass form curves over 256-bit fields (e.i., secp256k1 curve can be used).

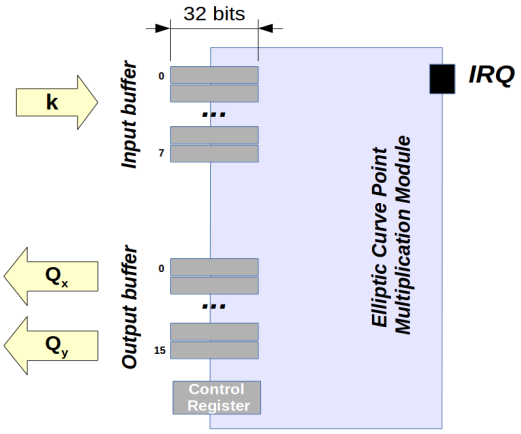


Figure 5.13: Generic representation of the Elliptic Curve Point Multiplication (ECPM) module.

The architecture of the article [11] is equipped with a BRAM that can store the given curve's parameters, such as the generator point P, its curve order n. This also means that the curve's parameters must be written only once (for example, at the boot time) to the IP and not every time before the point multiplication. Figure 5.13 represents the schema of the point multiplication module, in which the k (scalar) is stored in a 256 bits long buffer (8x32bits). The output buffer contains the coordinates of the result of the point multiplication (Q_x, Q_y). Each of the coordinates is represented in 256 bits.

It must be noted that the coordinates and other parameters of the curves can be stored in the IP. It is assumed that these parameters are already stored in the IP. Hence the write access of the coordinates P_x, P_y and other parameters are not simulated.

The hardware accelerator implementation for the ed25519 curve (architecture proposed in [12]) is slightly different than the module described latter. It operates on Residu Number System (RNS) representation, and it also uses Jacobian coordinates, which means that a point P is composed of x, y and z 256 bits long coordinates (P_x, P_y, P_z). The generator point can also be stored in the register of the IP similarly like in the case of the architecture described latter. As this architecture is dedicated to the edwards25519 curve, other parameters are not required. The output buffer of the module contains the result of the point multiplication, Q_x, Q_y, Q_z, each coordinate is represented on 256 bits. For achieving the throughput of the proposed implementation it has to be assumed that the coordinates are in RNS representation.

5.3.5 Bridge between Linux user space and SystemC modules

Previous sections have described the essential components of every SystemC modules modeling hardware accelerators. The sections also highlighted the essential communications between the CPU and the IPs (write/read access from the CPU, IRQ sending to the CPU from the given IP). Section 5.2 (p.108) mentions that the modeled architecture must allow the execution of the Linux operating system, which allows the execution of the blockchain APIs. When the API is in a phase in which a cryptographic function has to be performed, the CPU has to call the given hardware accelerator, which means that it writes/reads input/output data to/from the IP following the same logic as described in figure 5.11 (p.123). However,

the problem is that the CPU cannot directly access the hardware accelerator modules from an API. An application runs in the so-called user space, which means that the application cannot directly access hardware addresses. An application uses a predefined zone of the memory and thus the addresses.

The hardware IP can be accessed from an application thanks to the Linux Kernel devices drivers [150] that run in Kernel space and are authorized to access the hardware. These device drivers can be considered as bridges between the user space applications and the hardware of the given architecture.

In the proposed architecture (see figure 5.9, p.119), Linux Kernel Device Drivers allow the communication between the CPU executing a blockchain application and the hardware accelerator IPs modeled in SystemC-TLM. The proposed device drivers provide access to the given IPs (see page 134) and implement the logic of calling Operating Performance Points (OPP) which are used in PwClkARCH Power Management Unit (PMU). This specific device drivers is called Power Management Device Driver (PMDD) (see page 136), which also implements Linux-driven frequency division (CPUfreq, see page 138) on the ARM-based CPU architecture.

5.3.5.1 Developing Linux Kernel Device Drivers

Linux Kernel device drivers can be classified into three groups: Character devices, Block devices, Network interfaces [150].

- Character (char) devices:

  these devices can be accessed like a file under */dev* Linux repertory */dev/my_device* with a stream of bytes. These devices perform *open*, *write*, *read*, *ioctl*, and *close* system calls. For example, in user space application, the device can be opened by calling *file = open(/dev/my_device*). From that time, the file can be called in the application to make read or write accesses (e.g., *write(file, data, data_size)*).

- Block devices:

  can be accessed the same way as char devices. The main difference between char and block devices is that the data is managed differently internally by the Kernel. These devices are usually used to handle I/O operations. The I/O operations are usually handled with *ioctl* (input/output control) calls. From a development point of view, block devices have a more complex structure and are more complicated to implement than char devices.

- As the name suggests, network interfaces are intended to handle network transactions and data exchange with other hosts.

The communication between the blockchain API executed on the CPU and the IPs modeled in SystemC-TLM of the proposed architecture (see figure 5.9, p.119) is provided via char devices. The development of char devices is less complicated and faster than block device development. However, it should be noted that block devices could also perfectly handle the communication between the CPU and the IPs.

It can also be noted that all of these devices are written in C language with some dedicated functions for Kernel operations. The compilation of these devices requires Linux Kernel

headers provided thanks to the Linux Kernel build process. If the Kernel device were not compiled for the same Kernel in which the device will be used, the device would not be able to work, or the Kernel execution will be stalled completely. As the blockchain APIs were written in C++ language, they can call the char devices without any difficulty.

5.3.5.1.1 Basic functions of the device drivers

The objective of a device driver is to access the given IP and creating a bridge between the user space application and the given IP. The given IP can be accessed thanks to its address, which is also called physical address. The physical address is declared both in the device and Device Tree Source (DTS). The DTS describes the hardware components of the architecture, including the physical addresses, register ranges, clock frequencies, IRQs, and etc. Linux requires this description in order to know on which hardware architecture Linux is launched. When the device driver is mounted to the Kernel, it determines the virtual address of the given IP by calling *virtual_address = ioremap(physical_address, register_range)*. Reading and writing to this virtual address allows access from/to the IP.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include<sys/ioctl.h>

#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)

int main()
{
    int file;
    int32_t value;
    int32_t data[8];

    file = open("/dev/my_device", O_RDWR);

    write(file, data, 32); // 4*8 byte write

    read(file, data, 32); // 4*8 byte read

    ioctl(file, WR_VALUE, (int32_t*) &value); // write value

    close(file);

    return 0;
}
```

Listing 5.2: Example of the API calling a char device driver.

The following list describes the simplified functioning of a char device when user space application calls *open*, *write*, *read*, and *ioctl* (input/output control) functions. Listing 5.2

shows an example of a flow of calls within an user space application[4].

- ***open***: before the application could perform read/write accesses, it must open the device driver (it is logical because device drivers are similar to files, they must be opened before accessing it). The device can be opened by *file = open("/dev/my_device", O_RDWR)*.
- ***write***: the application aims to write to the IP. It calls the function *write(file, data, data_size)*. The *dev_write(...)* function is called in the device driver. Within this function, ***any logic can be implemented***. Usually, *copy_from_user(...)* function is implemented to retrieve the data from the application. The retrieved data then can be written to the IP by using *iowrite32(data, virtual_address)*. When *dev_write(...)* terminates (returns a 0), the user space application continues to run.
- ***read***: when the application wants to read from the IP it calls *read(file, data, data_size)*. This function corresponds to the *dev_read(...)* function in the device driver. This function is implemented to read the data from the IP and forward it to the user space application. Usually, it uses *data = ioread32(virtual_address)* instruction to retrieve the data from the IP and copies this data to the application by calling the *copy_to_user(...)* function. However, ***any logic can be implemented*** in this *dev_read(...)* function.
- ***ioctl***: when input/output control function is called, usually the application attempts to make a read or write access on the IP. The *dev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)* function in the device driver includes a *switch(cmd)* instruction to determine which command (*cmd*) the application attempts to call. This command can also be considered as a pointer to an action that the *ioctl* realizes. It must be noted that the command must be a unique identity that can be generated with *_IOW(...)* and *_IOR(...)* Kernel macros. It also has to be noted that an action of the *dev_ioctl(...)* ***can implement any logic***. The *arg* parameter of the function can be used to write or read data to/from the device driver.

The *ioctl(...)* function can be used instead of *write(...)* and *read(...)* functions. However, Linux limits the number of commands (identities) that can be generated for *ioctl* calls. Another drawback is: "The unstructured nature of the ioctl call has caused it to fall out of favor among kernel developers" [150].

The *dev_write*, *dev_read*, and *dev_ioctl* calls can implement any logic, as was already mentioned previously. The communication's logic between the CPU and an IP presented in figure 5.11 (p.123) includes the interruptions (IRQs) required to deny write/read access to the IP until it computes. The CPU must wait for the IRQ until the IP is busy. This IRQ management can be done thanks to two properties of device drivers.

First, a so-called blocking I/O can be deployed, which means that the process doing the I/O can be put to sleep mode while the data is inaccessible ("go to sleep waiting for data"[150]). When the process is in the sleep phase, the processor is freed up for other uses, so the device driver that implements this feature does not put the whole system into a blocked state until the data is ready. To implement this type of I/O, a wait queue object is necessary to be initialized in the initialize phase of the device driver.

[4]Useful tutorials can be found at: https://github.com/Embetronicx/Tutorials/tree/master/Linux/Device_Driver

```
wait_queue_head_t my_queue;
init_waitqueue_head (&my_queue);
```

The process can be put into the sleep phase by calling a wait event function that waits for an event and verifies if the Boolean condition (*my_condition*) associated with the event is true.

```
wait_event_interruptible(wait_queue_head_t my_queue, int my_condition);
```

The process remains in sleep mode as long as the process in the corresponding queue is not awake. The process can be woken up by calling:

```
wake_up_interruptible(wait_queue_head_t *queue);
my_condition = 1;
```

The event should be woken up when the IP specifies that it has completed the calculation by sending an IRQ to the CPU. The device driver should subscribe to the IP's IRQ.

```
struct device_node *np;
// Find rp_wires_in node in DTS
np = of_find_node_by_name(NULL, "rp_wires_in");
int irq = irq_of_parse_and_map(np, IP_IRQ_NUM);

// Subscribe to irq, => callback to irq_handler function
// 0x81 : IRQ active on rising edge
result = request_irq(irq, (irq_handler_t)irq_handler, 0x81, DEV_NAME, (void
    *)(irq_handler));
```

These instruction should be declared in the internalization phase of the device driver. When an IRQ occurs (sent by the IP) the *irq_handler* callback function is executed. In this function the previously mentioned wake up process can be applied.

A concrete example of the SHA-256 device driver for the functional architecture is given in Appendix B (p.167). It should be noted that the power-managed architecture requires a slightly modified device driver.

5.3.5.1.2 Basic logic of the device drivers corresponding to the IPs

The basic logic that is deployed in all of the device driver handling the accesses of an hash IP, applied in the functional architecture is represented in the following algorithm (see Algorithm 4).

When the device driver is mounted to the Linux Kernel, the first step is the initialization of the device driver (*Init procedure*). The driver subscribes to the IP's IRQ, and it initializes the waiting queue, which can enable to set the read and write procedure into a sleep mode.

When the API performs write access (more exactly from the cryptographic library), the write procedure is executed in the device driver. A previous part of the thesis (p.123) mentioned that before the first message chunk is written into the input buffer of the IP, the output buffer can be filled with an initial hash value (H_0) that is not the default one. The first condition of the hash device driver is to identify the length of the input data. If the length is equal to the length of a hash digest (e.g., 256 bits in the case of SHA-256) it means that the API wants to write an initial hash to the IP. Therefore the device driver writes the input data (initial hash) to the output buffer of the IP. When the length is equal to the length of a message chunk, the input buffer of the IP should be filled with the input data (message

Algorithm 4 Basic logic of device drivers for hash modules (functional architecture)

```
1:  procedure  Init device driver
2:      Subscribe for IP's IRQ
3:      Init wait queue ⇒ my_queue, my_condition
4:      Other init for device driver
    procedure write( input_Data )
2:      Init Hash_length
        Init Data_chunk_length
4:      firstMessageChunk = 1
        inputIsSet = 0
6:      if length(input_Data) == Hash_length then
            write initial hash value (H_0) to the IP's output buffer ⇒ write(input_Data)
8:      else if length(input_Data) == Data_chunk_length then
            if firstMessageChunk == 1 then
10:             write First Message Chunk to the IP's input buffer ⇒ write( input_Data)
                inputIsSet = 1
12:         else if firstMessageChunk == 0 then
                Wait until the internal hash is computed, thus waiting for the IRQ ⇒
14:             wait( my_queue, my_condition )
                write Message Chunk to the IP's input buffer ⇒ write(input_Data )
16:             inputIsSet = 1
            else
18:             return Error
        else
20:         return Error
        if inputIsSet == 1 then
22:         The device driver wrote the input_Data to the IP's input buffer
            write to the IP's control register (start compute) ⇒ write(START)
24:     return 0
    procedure read( output_Data )
        Wait until the internal hash is computed, thus waiting for the IRQ
3:      Get hash digest from IP ⇒ output_Data = read(IP's output buffer)
```

chunk). However, another condition must be done before the write access. If the input data represents the first message chunk, it can be written directly to the IP's input buffer. Else, the device driver must wait for the IRQ of the IP specifying that it has already finished the computation of an internal hash value. The write procedure is taken into a sleep state until the IRQ arrives. When the IRQ arrives, a callback function wakes up the process (using my_queue and $my\_condition$). When the IRQ is received the device driver can write the message chunk to the IP's input buffer. At the end of the procedure, if the input buffer of the IP was filled with the input data, the driver writes into the control register of the IP, which specifies that the IP can start the hash computation.

The read procedure is called from the API (via the cryptographic library which is used) to read the output buffer of the IP. Before the driver can read the data from the output buffer

of the IP it has to wait for the IP's IRQ specifying that the IP fished the hash computation.

The device drivers for the elliptic curve point multiplication ($k \times P(x,y) = Q(x,y)$) are slightly different from the device drivers for the hash producer IPs. The following algorithm describes the basic logic deployed in the device drivers of the EC point multiplication drivers.

Algorithm 5 Basic logic of device drivers for EC modules (functional architecture)

procedure Init device driver
    Subscribe for IP's IRQ
    Init wait queue $\Rightarrow$ my_queue, $my\_condition$
    func_convert_coord_256_to_288($coord_{256}$ **)**
    func_convert_coord_288_to_256($coord_{288}$ **)**
    Other init for device driver
procedure write(k **)**
    Init $k\_Is\_Written = 0$
    k_{256} = **func_convert_coord_288_to_256(** k **)**
    write k to the IP's input buffer $\Rightarrow$ **write(**k_{256}**)**
    $k\_Is\_Written = 1$
    if $k\_Is\_Written == 1$ **then**
        The device driver wrote the k to the IP's input buffer
        write to the IP's control register (start compute) $\Rightarrow$ **write(**$START$**)**
        $k_Is_Written = 0$
    return 0
procedure read($output\_Data$ **)**
    Init $P\_x\_Is\_Read = 0$
    Init $P\_y\_Is\_Read = 0$
    Init $tmp\_coordinate$
    Wait until the internal hash is computed, thus waiting for the IRQ
    if ($P\_x\_Is\_Read == 0$) && ($P\_y\_Is\_Read = 0$) **then**
        Get P_x from IP $\Rightarrow$ $tmp\_coordinate$ = **read(**IP's output buffer**)**
        $output\_Data$ = **func_convert_coord_256_to_288(** $tmp\_coordinate$ **)**
        $P\_x\_Is\_Read == 1$
    else if ($P\_x\_Is\_Read == 1$) && ($P\_y\_Is\_Read == 0$) **then**
        Get P_y from IP $\Rightarrow$ $tmp\_coordinate$ = **read(**IP's output buffer**)**
        $output\_Data$ = **func_convert_coord_256_to_288(** $tmp\_coordinate$ **)**
        $P\_y\_Is\_Read == 1$
    else
        return Error

When the device driver is mounted to the Kernel it passes to the initialization phase (*Init device driver*). In this phase, the IP subscribes to the IRQs and initializes the queue that would enable to put the read process into sleep mode. Another important entity in this phase is the declaration of *func_convert_coord_256_to_288* and *func_convert_coord_288_to_256* functions. The IPs that were chosen for the EC point multiplication operate on 256-bit length coordinates (256 bits per coordinate). However, most cryptographic libraries represent these coordinates and scalar values (e.g., k) on more than 256 bits (e.g., secp256k1 library uses

320 bits, Trezor-Crypto library uses 288 bits format). This is due to some software-level optimization possibilities which work better with these formats. Nevertheless, most cryptographic libraries provide functions to convert the 256-bit format to/from another format.

In the APIs Trezor-Crypto library was used to perform the EC point multiplication, and it uses 288-bit format. However, the IP requires the coordinates and the scalar on 256 bits format. Therefore, the device driver must convert the 256-bit representation to a 288-bit presentation. An from 288-bit to 256-bit representation.

For doing so, the *func_convert_coord_256_to_288()* function converts the 8x32-bit array to 32x8-bit array that serves as the input of *bn_read_be()* function converting the input into 32 bit x 9 (288 bits) array. This function is used to convert the the P point's coordinates from 256-bit format (provided by the IP) to a 288-bit representation (required by the cryptographic library). The scalar (k) is represented in 288 bits in the API. k must be converted to an 256-bit format in order to being used by the IP (k_{256}).

func_convert_coord_288_to_256() function converts the 32x9-bit array to a 32x9-bit array that serves as the input of *bn_write_be()* function converting the input into 32 bit x 8 (256 bits) array.

The EC point multiplication is composed of $k \times P(x, y) = Q(x, y)$, where k is a 256-bit integer (288-bit in the cryptographic library) and Q is the generator point of the curve (already stored in the IP). For producing the multiplication, the API calls the write procedure of the device driver that first convert k (288-bit) to a 256-bit format, (thanks to *func_convert_coord_288_to_256* function) after it writes k_{256} (256-bit format) to the IP input register, then it writes to the control register of the IP, which specifies that the IP can start the computation.

For reading the IP's result, the read procedure must be called. In which the coordinates of P cannot be read until the IP is busy. When the IP sends its IRQ, first, the driver reads the x coordinate (256-bits) from the IP to a temporal variable (*tmp_coordinate*). This value is converted into a 288-bit representation thanks to *func_convert_coord_256_to_288()* function and the *tmp_coordinate* variable. When the conversion is done, the API can receive the 288-bit length coordinate. The same steps are done for the y coordinate of P.

It should be noted that this driver can be used for the secp256k1 curve. In the case of the edwards25519 curve, it was assumed that the coordinates use an RNS representation in Jacobian coordinates. The Jacobian representation means that there is a z coordinate in addition to the x,y coordinates. In this case, the driver converts three coordinates into a 288-bit representation.

5.3.6 Power-managed architecture using PwClkARCH

The previous sections described the functional model of the proposed IoT architecture and the Kernel device drivers, which allow the communication between the API executed on the CPU and the IPs.

This section describes how the power-managed architecture is implemented (by using PwClkARCH) and which new functionalities the Kernel device drivers must include to implement the power management for the whole architecture. The section also explains the proposed power management logic and its implementation thanks to the device drivers. The proposed power-managed or powered architecture is represented in figure 5.14.

It must be noted that the power-managed architecture contains exactly the same IRQs

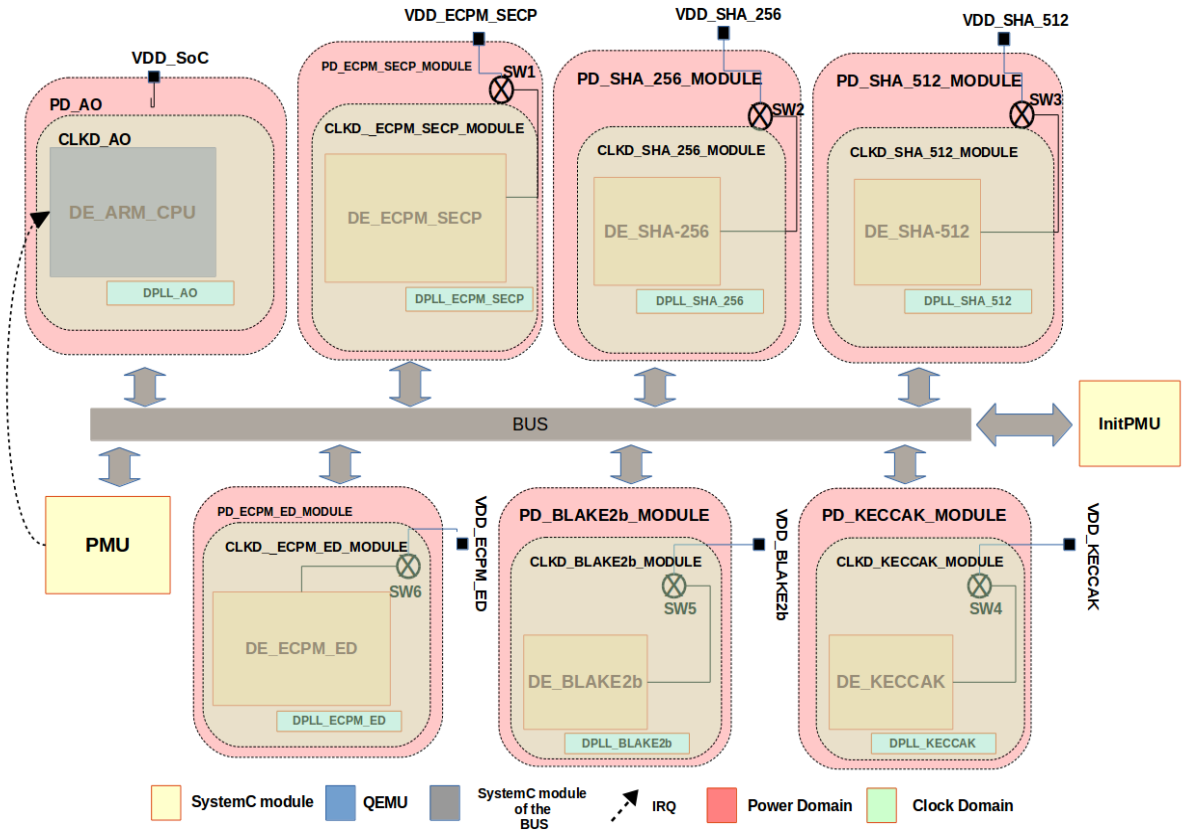


Figure 5.14: Power-Managed model of the IoT architecture (provided by PowerClkARCH).

that the functional architecture (see figure 5.9, p.119). In order to simplify the architecture's representation, only the IRQ between the Power Management Unit (PMU) and the CPU is represented. The IRQ generation by the PMU is a new feature that is also a contribution of this thesis to the PwClkARCH library. The interest of this interruption is detailed later.

In the power-managed architecture, the functional modules modeling the hardware accelerator IPs are associated with Design Elements (DEs) in the same manner as described on page 112. It is also described that all DEs can participate in a Power Domain (PD) and a Clock Domain (CD) (it is also possible that multiple DEs participate in the same PD or CD). In the proposed architecture, each DEs has its unique PD and CD. It can also be observed that except for the ARM-based CPU's power domain, each power domain has a power switch (SW) that enables to switch off the supply voltage (voltage gating). When the supply voltage of an IP is switched off, its dynamic and static power consumption is zero (see equations 5.1 and 5.2, p.113). Applying the voltage gating decreases the overall power consumption of the architecture. The ARM-based CPU's power domain is called *PD_AO* which means Power Domain Always On. The CPU's supply voltage cannot be switched off because it would stop the whole architecture working. The clock domains of the IPs include DPPLs, which enable applying frequency scaling. The frequency scaling technique is useful for decreasing the dynamic power consumption (see equation 5.1, p.113). The ARM-based CPU's clock domain contain DPLL (*DPLL_AO*). However, this DPLL is only needed for the power consumption measurement purpose because this ARM-based CPU architecture already contains a DPLL that can be controlled from the Linux standard CPUfreq device driver (more details on page 138). The control over the CPU's DPLL allows applying the frequency scaling in the CPU.

It was mentioned above that each IP participates in a unique PD and CD. It would be an obvious question to ask why not associate a single PD and CD for multiple IPs?

Due to the task flow that the APIs execute, there are no cryptographic functions that run simultaneously. Therefore, when a given cryptographic function is called, only the associated IP has to be activated (switched on). The other IPs can be in a switched-off state with zero power consumption. After the computation, the IP can be switched off again.

One Clock Domain per IP is required because every IP requires a different clock frequency.

Another important remark is that the BUS does not participate in any clock and power domain. For better understanding, it is worth examining figure 5.10 (p.121), which shows the architecture of the Zynq-7000. Any circuit designs implemented on the FPGA part of the Zynq-7000 board can be connected to the $AMBA^{®}$ Interconnect of the ARM CPU via AXI Ports. In the co-simulation, the AXI ports are simulated by the remote-ports, and the SystemC TLM BUS simulates the part of the $AMBA^{®}$ Interconnect. The BUS used to connect the SystemC modules of the IP can be considered a simulation tool and not a model of a given physical BUS.

The *InitPMU* module is also a simulation tool that initializes the Power Management Unit (PMU) module when the simulation starts. The PMU is the head of the power management of the architecture and is intended to be a physical IP. Austerely the PMU should take part in the same PD as the CPU because this unit must be active to apply the power management of the architecture.

5.3.6.1 Proposed Operating Performance Points and PMU

The PMU module contains the Operating Performance Points (see figure 5.6, p.114), which describe the power and clock states into which the power-managed architecture can be set. The proposed power management can be described as follows:

The first state of the power management is the booting state in which the CPU and all of the IPs are activated.

In the second state, the CPU executes tasks that cryptographic hardware IPs cannot accelerate. Therefore, only the CPU is activated (*VDD_SoC* is *ON*), the other IPs are switched off (power gated).

In a third phase, when a cryptographic function has to be executed, the given IP would be switched on (it is switched on before the device driver starts to write to the input/output buffers), and during its computation, the CPU's frequency is divided (frequency scaling). That also means that for every IP, a switch-on state is associated. When an IP ends its computation and the device driver has already read the results, the OPP can be changed. The new choice of the OPP (state) depends on the API's task. If another cryptographic function is to be executed, a switch-on state of the given IP is chosen. Else the power management returns to the second state, in which only the CPU is activated.

The proposed Operating Performance Points (OPP) with the Power State Table (PST) and Clock State Table (CST) are given below (see figure 5.15).

The ON/OFF state of the PST is respectively 1-1.2/0 V. In the OPP table, when an IP is active (except the Boot state), the clock frequency of the CPU (provided by *DPLL_AO*) is divided by three. When the power management returns into the OPP *Only_CPU_active* the

PST	VDD_SoC	VDD_ECPM_SECP	VDD_SHA_256	VDD_SHA_512	VDD_KECCAK	VDD_BLAKE2b	VDD_ECPM_ED
Boot	ON	ON	ON	ON	ON	ON	ON
Only_CPU_active	ON	OFF	OFF	OFF	OFF	OFF	OFF
ECPM_SECP active	ON	ON	OFF	OFF	OFF	OFF	OFF
SHA_256 active	ON	OFF	ON	OFF	OFF	OFF	OFF
SHA_512 active	ON	OFF	OFF	ON	OFF	OFF	OFF
KECCAK active	ON	OFF	OFF	OFF	ON	OFF	OFF
BLAKE2b active	ON	OFF	OFF	OFF	OFF	ON	OFF
ECPM_ED active	ON	OFF	OFF	OFF	OFF	OFF	ON

CST	CM1_AO	CM2_ECPM_SECP	CM3_SHA_256	CM4_SHA_512	CM5_KECCAK	CM6_BLAKE2b	CM7_ECPM_ED
Boot	2	2	2	2	2	2	2
Only_CPU_active	1	0	0	0	0	0	0
ECPM_SECP active	1	1	0	0	0	0	0
SHA_256 active	1	0	1	0	0	0	0
SHA_512 active	1	0	0	1	0	0	0
KECCAK active	1	0	0	0	1	0	0
BLAKE2b active	1	0	0	0	0	1	0
ECPM_ED active	1	0	0	0	0	0	1

OPP	DPLL_AO	DPLL_ECPM_SECP	DPLL_SHA_256	DPLL_SHA_512	DPLL_KECCAK	DPLL_BLAKE2b	DPLL_ECPM_ED	index
Boot	1/1	1/1	1/1	1/1	1/1	1/1	1/1	(1,1)
Only_CPU_active	1/1	1/1	1/1	1/1	1/1	1/1	1/1	(2,2)
ECPM_SECP active	1/3	1/1	1/1	1/1	1/1	1/1	1/1	(3,3)
SHA_256 active	1/3	1/1	1/1	1/1	1/1	1/1	1/1	(4,4)
SHA_512 active	1/3	1/1	1/1	1/1	1/1	1/1	1/1	(5,5)
KECCAK active	1/3	1/1	1/1	1/1	1/1	1/1	1/1	(6,6)
BLAKE2b active	1/3	1/1	1/1	1/1	1/1	1/1	1/1	(7,7)
ECPM_ED active	1/3	1/1	1/1	1/1	1/1	1/1	1/1	(8,8)

Figure 5.15: The proposed power management described by the OPP, PST and CST.

CPU clock frequency is not divided.

This section described the basic states that can be chosen for power management. It can be concluded that the states or OPPs are chosen according to APIs' task run on the CPU. If a cryptographic function is called, the OPP has to be changed. Likewise, if an IP ends its computation, a new OPP must be chosen and set.

It must also be noted that the Clock and Power State Tables described above were given according to a specific use case in which only one IP could be used at a time. In another use case where more than one IP must be used simultaneously, the Clock and Power State Tables could be changed according to the requirements.

In principle, the CPU is intended to set new OPP according to the API's task running. For doing so, the CPU has to access the PMU. This operation cannot be done directly as it was described in the case of the IP accesses. A device driver is necessary to change the OPP, this proposed device driver is called Power Management Device Driver (PMDD). In addition to performing access to the PMU for OPP selection, this driver also implements the logic of the decision of which OPP has to be selected. The OPP selection is not integrated into the API or the given cryptographic library but is hidden in device drivers and especially in PMDD. This is an advantage because the API structure does not need to be changed if the aim is to modify the power management logic.

5.3.6.2 Development of Power Management Device Driver (PMDD)

In previous sections, it was already mentioned that the CPU executing an API could not perform direct access to an IP. This is exactly the case whit the PMU. New OPP can be set by calling the dedicated device driver of the PMU called PMDD.

In fact, the API could perform a write access via the PMDD to the PMU by specifying the desired OPP to be set. However, in this case, the API structure and the cryptographic libraries must be modified each time new power management logic is deployed (choice of OPP when calling a given cryptographic function). In the proposed power management logic, the choice of the OPPs can be hidden from the API thanks to the PMDD and the drivers used to call the hardware accelerator IPs. This logic is detailed below.

The PMU and the PMDD must be interfaced, which means that the PMDD must contain the same OPPs as declared in the PMU OPP table.

The PMDD also contains the names of all of the device drivers that are used to interact with the given IP (stored in a list of IPs). The activity of the IP is intended to show whether the IP is active (it is computing, it is in a switched-on state). The specification of the IP activity is the key point in deciding which OPP to chose to be applied. Figure 5.16 represents the power management orchestration realized thanks to the logic of PMDD and IP drivers.

Write access to the IP is instantiated from the given API's cryptographic library using the IP device driver as a bridge (step **1**). Before writing the to the IP's input buffer (step **2**) in the power-manged architecture, the given device driver should call the function *requestNewOPP(DEV_NAME)* (step **3**).

In the case of hash functions, the *requestNewOPP(DEV_NAME)* function is called only before the first message chunk is written. This makes sense because as long as the final hash is not produced, the given IP must be activated, which also means that the OPP remains the same in power management.

The function *requestNewOPP(DEV_NAME)* is declared with the *EXPORT_SYMBOL()* macro in the PMDD, which allows being called from other device drivers. When an IP device driver calls this function, it is executed in the PMDD, and the relevant device driver waits for the *requestNewOPP(DEV_NAME)* to finish. This function first identifies the device (caller) by its name (step **4**). After the identification, the corresponding IP's activity is specified (set to 1) (step **5**).

In the following phase (step **6**), the PMDD verifies the list of IPs in order to find the active IP. The active IP corresponds to an OPP declared in the PMU (see figure 5.15, e.g., *SHA_256 active*). Therefore, this OPP has to be applied in the PMU. As the decision of the OPP was made, now the PMDD can write the chosen OPP to the PMU in order to change the power management's state (step **7**).

The power-managed architecture contains several electronic components such as the DPLLs and power switches, which allow changing the clock frequency and supply power of an IP. The frequency shifting and power switching cannot be done in zero time. That also means that setting OPP in the PMU is not done instantaneously (step **8**). The PMDD has to wait until the PMU performs the change of a new OPP (same wait method as introduced in the IP's device drivers see page 127).

When the PMU changed the OPP, it sends an IRQ to the CPU, which specifies that the PMDD can continue working (step **9**).

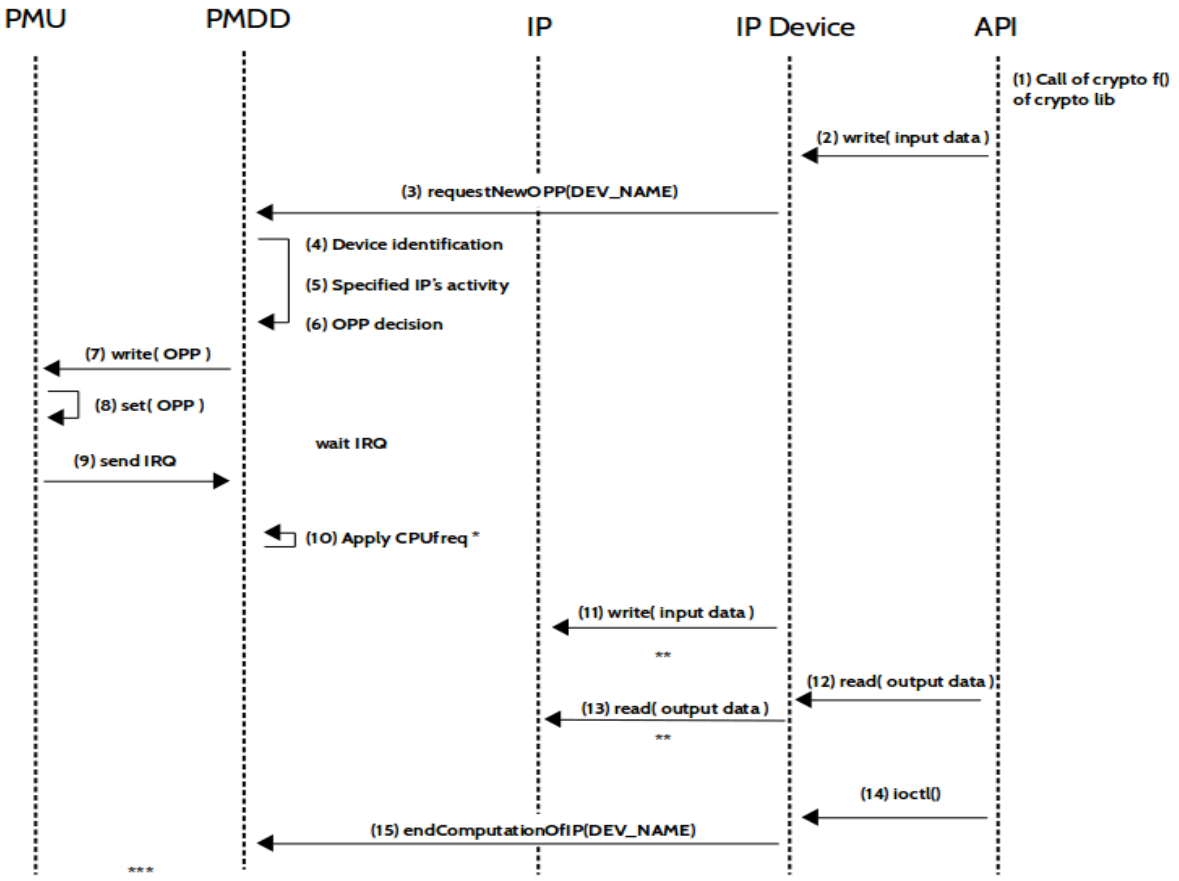


Figure 5.16: Power management orchestration with PMDD, IP drivers, and PMU. (*) the CPUfreq is applied by calling dedicated Linux device driver which allows scaling the CPU's clock frequency. (**) specifies the process flow described in algorithms 4 and 5, p.130. (***) PMDD identifies the device, sets the corresponding IP's activity to 0, and decides which OPP must be chosen. If there is no active IP, the *Only_CPU active* state is chosen. The following steps are similar to the steps from **7** to **11**.

In the following step (step **10**) is the frequency scaling of the CPU's frequency. The new frequency is set by calling the functions declared in the standard CPUfreq Kernel device driver (more details on page 138). The factor of division is declared in the structure of OPP's.

Now the *requestNewOPP(DEV_NAME)* function ends, which means that the given IP device can write the input data to the IP's input register (step **11**). After the IP device driver continue its execution (same process flow described in algorithms 4 and 5, p.130).

When the IP finished its computation the device driver can read the result of the computation which is stored in the IP's output buffer. This result is then copied to the API (user space). (steps **12-13**)

After reading of the output buffer an *ioctl()* (**14**) is called that makes the IP device driver call the *endComputationOfIP(DEV_NAME)* function (step **15**), which specifies that the computation and the output buffer reading of the given IP is done, and the a new OPP must be selected. It must be noted that in the case of the EC point multiplication the *ioctl()* is not necessary, the IP device driver can call *endComputationOfIP(DEV_NAME)* directly after reading the IP's output buffer. However, in the case of the cryptographic hash functions the

ioctl() is required because it specifies that the IP would not be used for a while.

The *endComputationOfIP(DEV_NAME)* of PMDD identifies the device, sets the corresponding IP's activity to 0, and decides which OPP must be chosen. If there is no active IP, the *Only_CPU active* state is chosen. The following steps are similar to the steps from **7** to **10**.

The most significant advantage of the PMDD is that it allows the deployment of any OPP of power management and applying it without modifying the API. It was also observed that the IP device drivers of the power-managed architecture are slightly different from the functional architecture. These drivers call external functions *requestNewOPP* and *endComputationOfIP* to specify to the PMDD which IPs must be powered. It should be specified that calls between the PMDD and the IP devices drivers can be enabled if a dependency is configured in the compile process (all of the drivers compiled together).

5.3.6.2.1 Basic Modifications on cryptographic libraries

The previous section mentioned that the API does not need to be modified when new power management is implemented and applied. This is also true for the read/write access to an IP. When an API calls a cryptographic function, it uses a cryptographic library that deploys the given function. In fact, it is not the API that deploys read/write calls of a device driver but the cryptographic library itself by using the same basic functions which were described in the example 5.2, p.127.

The example of cryptography libraries' modification is demonstrated in the EC point multiplication function.

The ***EC_mult*** function in the corresponding cryptographic library is modified as follows. First the scalar value k (nonce value) is written to the IP after the IP's device driver is opened.

The result can be read just after writing the scalar value. The device driver would block the reading until the IP computes the multiplication.

The coordinates of the point (result of the multiplication) are read one by one (*coordinate.x*, *coordinate.y*), as figure 5.17 depicts.

In the case of hash function after reading the result, an *ioctl()* is called to specify that the hash production is done. This also means that a new power management phase would be decided.

It can be seen and concluded that when new power management or functional logic (in device driver level) would be added to the architecture, the API's structure does not need to be modified. Therefore the API is separated from the device drivers because it is compiled with the modified cryptographic libraries.

5.3.6.2.2 Applying CPU frequency scaling by Linux over the CPU of the Processing System

Newer Linux Kernel (version up to 3.4) can contain essential drivers for applying CPU frequency scaling. This frequency scaling enables to set the CPU frequency up or down depending on the performance that the CPU wants to provide. When a lower CPU frequency

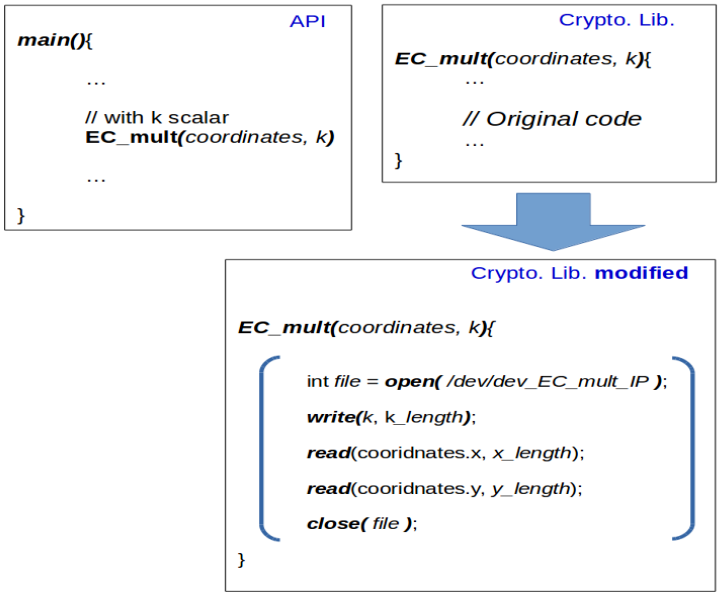


Figure 5.17: Principle of the API cryptographic EC multiplication call, and the modified cryptographic library calling the IP device driver.

is applied, the dynamic power consumption is less. Contrary, when faster computation is required, the CPU frequency can be increased, making a higher power consumption.

The CPU frequency scaling can be applied dynamically or manually depending on the frequency scaling governor is used. For example, the *ondemand* governor scales the frequency according to the actual workload. The power management applied in the PMDD, and the PMU (see above) aims to scale the frequency to the given value determined according to the PMU division factor. For doing so, the *userspace* governor must be used. The bash command can set the governor any time:

```
echo userspace > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

In the proposed architecture, this command line is called while the Linux system boots. The new CPU frequency can be requested by calling:

```
echo $freqValue > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

Where *$freqValue* is the new value of the CPU's frequency. It must be noted that it is impossible to set any value of the frequency. The available values depend on the hardware architecture, and these values are specified in the architecture device tree source (DTS).

```
cpu0: cpu@0 {
     compatible = "arm,cortex-a9";
     device_type = "cpu";
     reg = <0>;
     clocks = <&clkc 3>;
     clock-latency = <1000>;
     cpu0-supply = <&regulator_vccpint>;
     operating-points = <
       /* kHz    uV */
         666667  1000000
         444443  1000000
```

```
12            333334  1000000
13            222223  1000000
14        >;
15    };
```

The *operating-points* section describes the available frequency-supply voltage pairs of the processing system of Zynq-7000 (based on multicore ARM Cortex-A9). The number of the applicable frequencies of the CPU is limited, but the minimum frequency can be three times less than the maximum frequency. With a frequency less then 222 Mhz the ARM Cortex-A9 stops working.

It should be noted that setting a new frequency value by a bash code is not possible from the device driver. Therefore, the PMDD sets the frequency by calling the specific functions of frequency device drivers.

It is necessary to know that only the essential device drivers should be added to the Kernel in the case of constrained devices that run Linux OS. This makes sense because including all the drivers in the Kernel makes growing the Kernel size quick, and the constrained device has limited memory place.

Including the device drivers which enable the CPU frequency scaling can be done while compiling the Kernel image. However, including the dedicated devices for CPU frequency scaling can be challenging. This thesis work contributes to the descriptions and configuration files to enable the CPU frequency scaling in *xilinx-zynq-a9* device emulated by QEMU[5].

Another contribution is also available that is intended to create all the necessary for the co-simulation demo[6]. This repository also includes a Linux Kernel device driver which enable writing access to the *debugdev* SystemC-TLM module that is provided by default in the zynq-demo.

5.3.7 Conclusion

The previous sections described the proposed IoT architecture model, in which QEMU emulates a complex ARM CPU architecture, and the hardware accelerator IPs for cryptographic primitives are modeled in SystemC-TLM. These sections also described the necessity of deploying Linux Kernel device drivers, which allow the interaction between an API executed on the CPU and the IPs.

The sections also mentioned the necessary modification of the functional architecture model in order to obtain a power-managed architecture. An advantage of the power-managed architecture is that the modification of the power management logic does not require modifying the API, but only the modification of the device drivers, especially the PMDD, the PMU, and eventually the given cryptographic libraries. It can be noticed that the modification of the cryptographic libraries is usually not required when new power management is applied. They usually follow the same steps as declared for the functional architecture.

The deployment of an architecture model that enables running a Linux Operating System and blockchain APIs is a real challenge. Emulating an ARM architecture running a Linux OS with QEMU hides many difficulties.

First of all, every parameter of QEMU must be well-chosen, the Kernel image of Linux must be compiled with all the necessary options (including all of the necessary drivers and

[5]GitHub repository is available at https://github.com/KRolander/QEMU_CPUFreq_Zynq.git
[6]GitHub repository is available at https://github.com/KRolander/Co-Simulation-Zynq-7000-Tools

Xilinx u-boot tools) in order to allow the frequency scaling with the CPUfreq command. Another challenge is the creation of the root file system that must also be well-formatted for QEMU, which means that in several cases, the root file system must be compressed to another format if its size is significant. It also has to contain several Linux command binaries.

The Kernel device drivers can also make surprises when the included Kernel headers' version is different from the Kernel image used. As the Kernel C language is slightly different from standard C, the debug of Kernel device drivers can also be challenging because a simple error makes the crash of whole Kernel. The device driver deployment, including IRQ handling and interruptible processes, also requires specific knowledge about device driver implementation.

A git repository has been made that explains which Linux configurations are needed and which parameters are necessary for launching QEMU in a co-simulation environment. This repository also provides bash commands that help to create well-formatted root file systems easily. The contribution also shows how to compile/cross-compile Kernel device drivers / API's for the co-simulation. The contribution also focuses on the IRQ management between the device drivers and the co-simulated SystemC-TLM IPs.

It can also be noted that this thesis work is the first contribution in which the PwClkARCH power management tool was ordered from a Linux OS running a dedicated API.

5.4 Running blockchain APIs on the power architecture model

Previous sections and chapters described the implementation of blockchain APIs and the functional and power-managed IoT architecture deployment. The following sections provide the possible results of the power management when the given blockchain API is executed on the top of the architecture.

The results consist of the total energy consumption and total execution time of the given API executed on the architecture. The architecture energy consumption is measured by PwClkARCH. The total energy consumption is determined according to the power consumption which is the sum of the dynamic and the static power consumption. It should be noted that each unit of the proposed architecture model (IPs and the ARM CPU architecture) has its own dynamic and static power consumption that is summed together to obtain the total power consumption.

$$P_{dynamic} = \alpha * C * V^2 * F \tag{5.3}$$

$$P_{static} = \frac{V^2}{R_{leakage}} \tag{5.4}$$

$$P_{total} = P_{dynamic} + P_{static} \tag{5.5}$$

The description of PwClkARCH (see page 112) mentioned how this tool measures power consumption and which techniques can optimize this consumption. However, it was not discussed that the capacitance (C) and the leakage resistance ($R_{leakage}$) are confidential information of the circuit manufacturer. Unfortunately, this thesis work cannot provide exact

values for resistance and capacitance because no manufacturer has provided these values. Therefore, to show that power management can optimize the overall power consumption of the proposed architecture, the values of C and R are estimated.

In the case of ASIC circuits (for example, the hardware accelerators for the hash computation), the authors give the total power consumption of the designs. However, it is hard to estimate how many percentages the dynamic and static power consumption occupy in the total power consumption. Therefore, it can be assumed that static consumption takes 10% of the total power consumption, but these values can be changed anyway.

The literature about the EC multiplication designs (FPGA designs) does not provide information about power consumption. Is it absurd to think that the proposed IoT architecture would contain an FPGA board in order to implement the EC multiplication hardware. In general, the size of an IP design implemented on an FPGA is 10 times larger than the same design on an ASIC. Therefore, it can be considered that the ASIC implementation of the EC multiplication IP could consume at least 10 times less energy.

The power consumption of the ECPM designs are not provided but, the number of FPGA components, such as the number of LUT, BRAMS, etc., are. Xilinx provides a power estimation tool (Xilinx Power Estimator [151]) that can estimate the power consumption of a given device according to the design components implemented in the FPGA. Thanks to this tool, the EC point multiplication IP power consumption and the PS (ARM CPU architecture) can be estimated.

The power estimation of the EC point multiplication on the secp256k1 curve (design of [11]) is obtained by setting the number of components (FF, DSP, LUT, etc.), and the device clock frequency with a toggle rate of 12.5 %. According to XPE the dynamic power consumption of the IP is equal to 308 mW and its static power consumption to 67 mW (the supply voltage is equal to 1V). These two values are divided by 10 to obtain a power consumption of the on ASIC. The capacitance and leakage resistance are determined according to the equations 5.3 and 5.4. The capacitance (C) is equal to 0.194 nF. The leakage resistance ($R_{leakage}$) is equal to 149.25 Ω.

The power consumption of the EC point multiplication on edwards25519 curve (design of [12]) was estimated in the same was as in the previous case. However, it must be noted that the estimation was performed for the implementation on Virtex-7 board, because the authors of [12] did not provide the design's parameters on Ultra Scale + board. The capacitance and leakage resistance are retrieved according to the dynamic an static power consumption. The capacitance (C) is equal to 0.503 nF. The leakage resistance ($R_{leakage}$) is equal to 55.24 Ω. The estimated values for the ECPM designs are represented in tab. 5.3.

The power consumption of the ARM CPU architecture (PS of the Xilinx-7000 board) can also be estimated by XPE. In the estimations a maximal 667 MHz clock frequency, a supply voltage of 1 V and a workload of 100% was applied. According to the static and dynamic power consumption, the leakage resistance and the capacitance are the follows: $R_{leakage}$ is equal to 27.72 Ohm, and C is equal to 1.138 nF. The estimated values are represented in tab. 5.2.

Estimating the capacitance for the ASIC IPs (hash hardware accelerators) is less com-

Design	P_{dyn}	P_{stat}	C	$R_{leakage}$
ARM CPU arch. (PS)	759 mW	44 mW	1.14 nF	27.72 Ω

Table 5.2: Dynamic and static power consumption, capacitance and leakage resistance of ARM CPU architectures (PS)

plicated because, in most designs, the authors provide the IP's dynamic consumption. The scaling method represented in [104] is used to scale the original power consumption of the IP to a power consumption that the IP would have if it had been realized in 35-40 nm CMOS technology.

It was already assumed above that the dynamic power consumption of an ASIC IP takes approximately 90% of the total power consumption. Therefore, the static power consumption takes the rest (10%) of the total power consumption. According to these assumptions, the capacitance and the leakage resistance can be calculated.

After knowing the dynamic and static power consumption, the capacitance and the leakage resistance can be calculated for all of the hash accelerators, except the SHA-512 design. The authors did not publish the power consumption of this IP. Therefore, for estimating its power consumption, the SHA-256 IP's consumption is used. The SHA-512 algorithm makes twice more operations than the SHA-256. Following this logic, the power consumption of the SHA-512 IP could be at least two times higher than the SHA-256 IP's.

The dynamic-static power consumptions and the estimated values of the capacitance and leakage resistance of the IPs are listed in table 5.3.

Design	P_{dyn}	P_{stat}	C	$R_{leakage}$
ECPM secp256k1 [11]	30.8 mW	6.7 mW	0.194 nF	149.25 Ω
ECPM edwards25519 [12]	91.5 mW	18.1 mW	0.5 nF	55.24 Ω
SHA-256 [136]	5.93 mW	0.658 mW	17.26 pF	1.77 kΩ
SHA-512 [137]	11.86 mW	1.32 mW	40.67 pF	883.36Ω
Keccak [138]	9.56 mW	1.06 mW	23.34 pF	1.36 kΩ
BLAKE2b [139]	0.108 mW	0.012 mW	0.85 pF	46.88 kΩ

Table 5.3: Hardware accelerator IPs' dynamic and static power consumption, capacitance and leakage resistance.

It should also be noted that in the case of BLAKE2b design, the value of the leakage resistance seems suspicious. However, this value could not be compared with other designs' values because the only design was found in the literature is provided in the article [139]. In the case of SHA-256 and Keccak, the leakage resistance values may also be surveyed, because these values also seem a bit high.

5.4.1 Preliminary results of the power management

In the previous section, the power consumption of the IPs and the ARM CPU architecture were estimated (see tab. 5.2 and 5.3) in order to be used in the overall energy consumption

measurements of the proposed IoT architecture. The preliminary results are obtained according to the measurements of the execution of Ethereum API (see page 61), and a Keccak hash creation procedure in which multiple internal hash values are created. In all of the cases, the CPU's frequency is divided by 3 when the frequency scaling is applied (a given IP is called). The penalty (latency) of the DPLL is set to 300μs as it is the worst-case time penalty that can happen in the ARM Cortex-A9. The latency of the power switches are set to 100 ns.

The power management is based on two strategies: first, only the ECPM operation is accelerated (the Keccak hash function is performed by the CPU). Second, ECPM and the Keccak function is also accelerated. Figure 5.20 (p.149) represents the variance of the overall energy consumption with and without applying a power management.

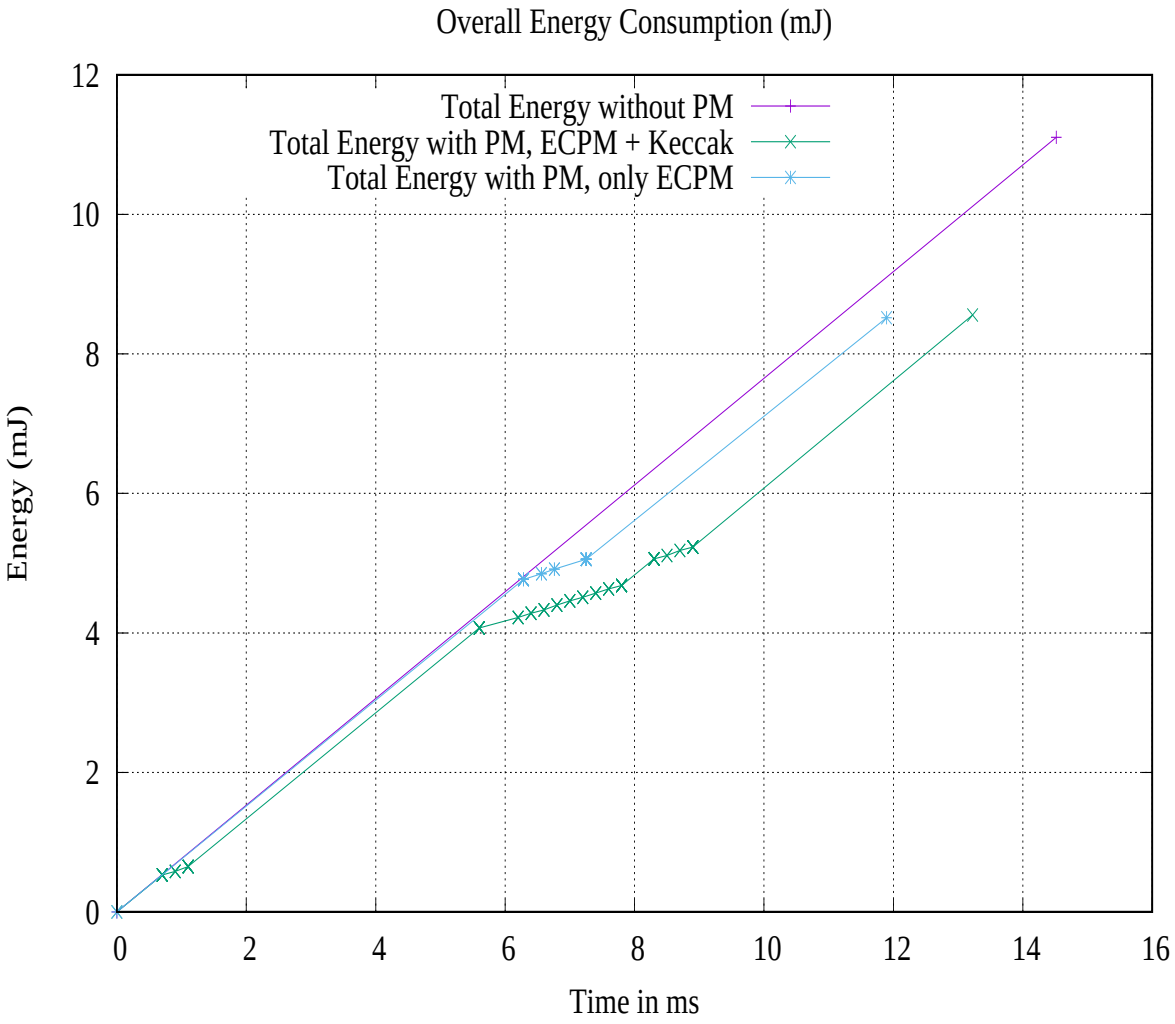


Figure 5.18: Preliminary results: Overall Energy Consumption of Ethereum API, retrieved by using PwClkARCH. *PM* in the titles of the curves means Power Management.

The curve in purple represents the energy consumption of the architecture when no power management is applied (hardware accelerators are not used).

The curve in blue represented the energy consumption when only the ECPM IP was called to accelerate the point multiplication over the secp256k1 curve. In this case, a significant 23.3% of overall energy consumption reduction can be achieved. In additions the energy consumption, the total execution time is also reduced by 18%.

In the second power management strategy, in addition to the ECPM, the Keccak hash function is also accelerated (curve in green, period between 5.5 and 7.8ms). Thanks to this power management strategy, the overall energy consumption can be reduced by 21%. It can be observed that when accelerating Keccak and ECPM together, the overall energy consumption and total execution time are slightly higher than hardware accelerating only the ECPM operation. These results are also shown in Tab. 5.7 (p.149), which resumes the mean of ten measurements per power management strategies.

Power Management (PM)	Overall Energy Consumption	Overall Energy Reduction	Total Execution Time	Total Time Reduction
No PM applied	11.11 mJ	0%	14.52 ms	0%
ECPM only	8.52 mJ	23.3%	11.9 ms	18%
ECPM+Keccak	8.77 mJ	21%	13.22 ms	9%

Table 5.4: Overall Energy Consumption and Total Execution Time of the proposed architecture when running Ethereum API. Three different power management were applied.

The over-consumption when hardware acceleration the Keccak hash function is due to Linux's scheduling and IRQ handling latency. Unfortunately, when one of the IPs of hash creation is called too frequently, the latency for performing one internal hash cannot compensate for the latency that is required for the IRQ and task scheduling of the OS.

In this scenario, the Keccak IP is called in a flow for performing eight internal hash digests. An additive $\sim$200μs has to be waited between every internal hash creation because of the IRQ and wake-up latencies

For better understanding how the power management is present in the flow of hash creation, the following figure represent the dynamic power consumption of a hash accelerator IP (Keccak but true for any hash acceleration). In the curve of the power consumption (see fig. 5.19), two main levels can be observed. In the high level (peak), the IP is doing the hash computation, which also means its workload is 100%. In the other lower levels, its workload is only 10% (these parameters can be set in PwClkARCH). In principle, the power consumption is much lower when the IP is accessed for reading and writing than in its computation phase. In this figure two internal hash are created, that is the reason of the two peaks.

It can be observed that the duration ($\sim$100-200μs) of these low levels is more significant than the peak of the consumption. In principle, the duration of the low level is due to the read and write accesses on the IP. However, these latencies are much higher than the measured I/O accesses (around 0-10us). These high latencies are due to the nature of the Linux operating system's IRQ and Kernel scheduling latency. The logic of the device drivers of the IPs is to wait for the IP's IRQ before writing or reading to/from it (see algo. 4 and 5, p.130). When the driver waits, the task is preempted. When the IRQ arrives, the task is woken up, the preemption and wake-up process is expensive in terms of latency around 100-1000μs. These latencies also influence the overall energy consumption of the architecture.

As the IP's driver has to wait until an internal hash is performed, a scheduling latency is present between every intermediate hash creation (every call of the IP). It can also be observed that the latency is also there because the PMU's device driver (PMDD) also uses IRQs to insure that the given OPP has been set successfully.

The influence of the scheduling and IRQ waiting on the overall energy consumption when

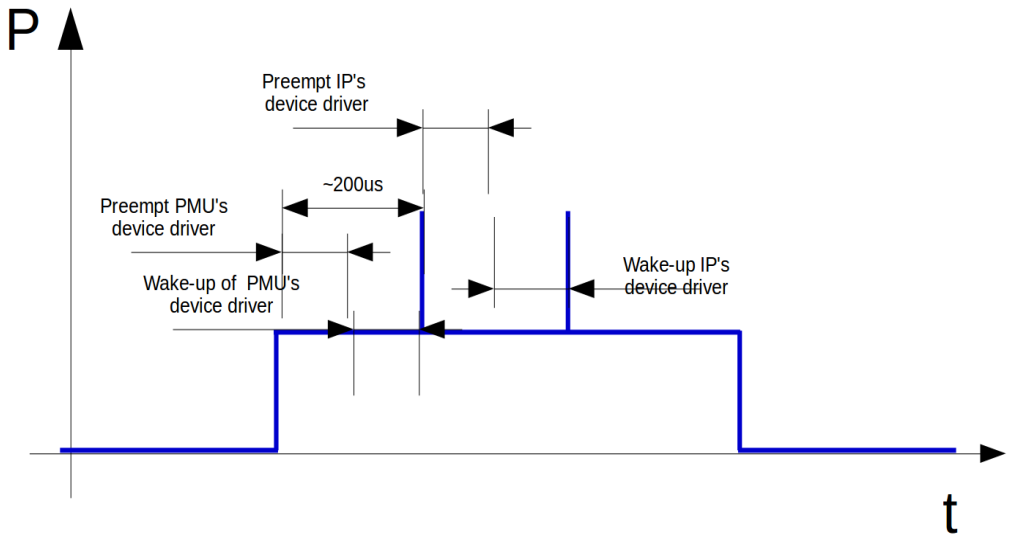


Figure 5.19: Example of the dynamic power consumption measured by PwClkARCH. The curve represents the power consumption of the Keccak IP, when a flow of internal hashes is performed.

a flow of the internal hash creation procedure has to be taken into place is represented in tab. 5.5. In this measurement the Keccak hash function is accelerated and 15 internal hash were performed. The first row represents the overall energy consumption when the hash procedure is not accelerated (it is executed by the CPU), the second row shows the consumption of the hardware accelerated operation.

Power Management (PM)	Overall Energy Consumption	Overall Energy Reduction	Total Execution Time	Total Time Reduction
No PM applied	0.7685 mJ	0%	1 ms	0%
Keccak Hash IP called	1.523 mJ	+98%	4.23 ms	+323%

Table 5.5: Preliminary results: Overall Energy Consumption and Total Execution Time when accelerating Keccak hash function, 15 internal hash is performed.

It can be observed that as the number of hash to perform increases an over-consumption of 98% happens, thus the use of the hash hardware accelerators does not make sense. When observing the total execution time, the additive execution time due to the IRQ waiting and task preemptions is a clear evidence (+323%).

There are two possible solutions for accelerating the hash creation procedures. It is possible to use a particular Linux patch called preempt-rt, which was created for real-time usage. A related work [152] showed that this patch on a Raspberry Pi 3 could produce a more efficient latency around 50μs. However, for using this patch, the device drivers and their logic should completely be modified, and the provided performance is probably not enough to achieve an efficient power saving.

Another possibility is to slightly modify the proposed architecture and the device drivers in order to avoid the maximum number of IRQs and scheduling latencies. For doing so, this thesis work proposes the slight modification of hash IPs. The modification of hash IPs

can be done as follows: the input buffer size can be extended to store ten times more input message chunks. In this way, the device driver can write input message chunks in a flow and write the number of the written message chunks to the IP's control register. When write access is done to the control register, the IP performs the hash procedure as many times as the value of the number written into the control register. When the device driver wants to write more message chunks, it waits (IRQ) until the IP finishes the number of operations that was declared in the control register.

The algorithm 6 represents the slightly modified logic implemented in the hash device drivers in order to use the IPs with wide buffers. It can be observed that before reading the final hash, the device driver writes the number of message chunks which was written into the IP's input buffer to the control register of the IP. It is also required in the device driver that it counts the number of message chunks and writes it to the hash IP's control register. It must be noted that when using the wide buffer the device driver cannot handle the initial hash writing into the IP.

It must be noted that the modification of the hash IPs would cause a higher power consumption. However, the IP would only contain more registers, a counter, and some basic logic elements, which means that the energy consumption would not increase enormously. It can be assumed that the overall energy consumption of the IPs would be increased by 15% maximum. In the following measurements and analysis the power consumption of the hash IPs equipped with wide buffers are increased by 15%.

Table 5.6 compares the overall energy consumption and total execution time of the 15 internal hash creation procedure when using the basic and the modified Keccak IP.

Power Management (PM)	Overall Energy Consumption	Overall Energy Reduction	Total Execution Time	Total Time Reduction
No PM applied	0.7685 mJ	0%	1 ms	0%
Keccak Hash IP called	1.523 mJ	+98%	4.23 ms	+323%
Keccak Hash IP wide buffer	0.68 mJ	-18.7%	1.45 ms	+45%

Table 5.6: Later results: Overall Energy Consumption and Total Execution Time when accelerating Keccak hash function, 15 internal hash is performed.

When embedding the wide buffer in the IP an 18.7% reduction can be achieved in the overall energy consumption, which is a significant improvement compared to the +98% of over-consumption. Using the wide buffered IP can optimize the overall energy consumption. However, the total execution time of hash (15 internal hashes) creation would increase by 45% (much significant reduction than in the case of basic IP, +323%).

These measurement were done in the case of Keccak IP, nevertheless the factor of redaction in overall energy consumption and increase in total execution time is true for every hash IPs because the latency of one hash creation of the studied hash IPs are close to each other.

Algorithm 6 Basic logic of device drivers for hash modules with wide buffers

```
1:  procedure  Init device driver
2:      Subscribe for IP's IRQ
3:      Init wait queue ⇒ my_queue, my_condition
4:      Other init for device driver
    procedure write( input_Data )
2:      Init Hash_length
        Init Data_chunk_length
4:      firstMessageChunk = 1
        inputIsSet = 0
6:      msgBlockNum = 0
        BUF_MAX = 10
8:      if inputIsSet == 0 then
            first msg chunk to set to the IP, the OPP has to be changed
10:         call requestNewOPP()
        else
12:         if msgBlockNum < BUF_MAX then
                write msg chunk to the IP's input buffer ⇒ write( input_Data)
14:             inputIsSet = 1
                msgBlockNum++
16:         else if msgBlockNum == BUF_MAX then
                Start computing, write the maximal number of chunks to the IP's ctrl. reg.
18:             ⇒ write(BUF_MAX)
                Wait until the internal hash values are computed, thus waiting for the IRQ ⇒
20:             wait( my_queue, my_condition )
                msgBlockNum = 0
22:             write Message Chunk to the IP's input buffer ⇒ write(input_Data )
                inputIsSet = 1
24:             msgBlockNum++
            else
26:             return Error
        return 0
    procedure read( output_Data )
        write the number of msg blocks (msgBlockNum)
3:      to the ctrl. reg. write(msgBlockNum )
        msgBlockNum = 0
        inputIsSet = 0
6:      Wait until the internal hash values are computed, thus waiting for the IRQ
        Get hash digest from IP ⇒ output_Data = read(IP's output buffer)
```

5.4.2 Final results of the power management

5.4.2.1 Ethereum API

In this measurement the Ethereum API described previously on page 61 is executed on the proposed IoT architecture. Figure 5.20 represents the overall energy consumption curve with

and without applying a power management. The power management is based on three strategies: first, only the ECPM operation is accelerated (the Keccak hash function is performed by the CPU). Second, only the Keccak and the ECPM functions are hardware accelerated. Finally, the ECPM and Keccak functions are hardware accelerated, but in this time the Keccak IP is equipped with a wide buffer (10 internal hash can be performed in row).

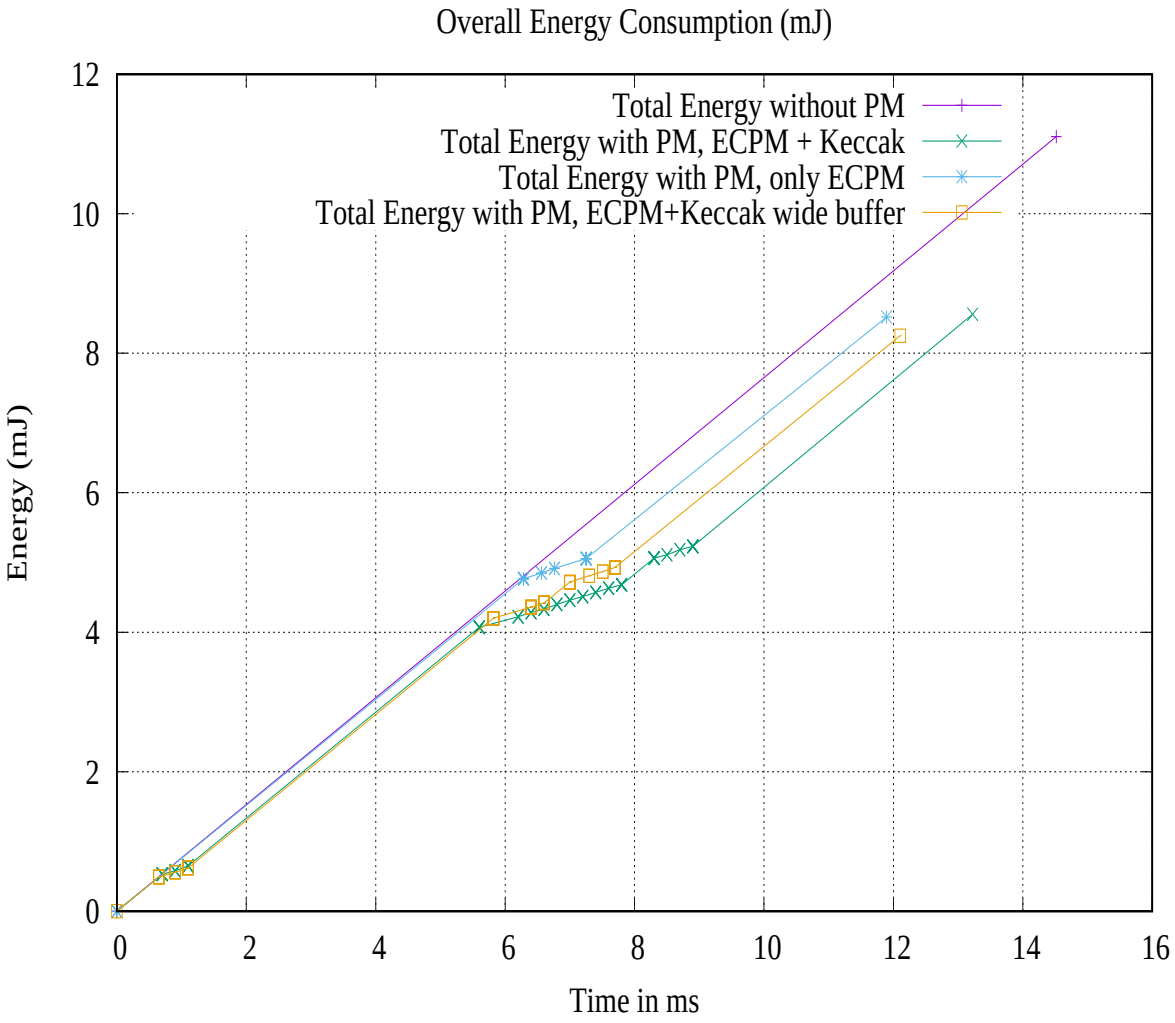


Figure 5.20: Overall Energy Consumption of Ethereum API, retrieved by using PwClkARCH. *PM* in the titles of the curves means Power Management.

Power Management (PM)	Overall Energy Consumption	Overall Energy Reduction	Total Execution Time	Total Time Reduction
No PM applied	11.11 mJ	0%	14.52 ms	0%
ECPM only	8.52 mJ	23.3%	11.9 ms	18%
ECPM+Keccak	8.77 mJ	21%	13.22 ms	9%
ECPM+Keccak wide buffer	8.31 mJ	25.2%	12.16 ms	16.3%

Table 5.7: Final results: Overall Energy Consumption and Total Execution Time of the proposed architecture when running Ethereum API. Three different power management were applied.

According to the results (see tab. 5.7 and figure 5.20), it can be concluded that in the case when the payload size is small (32 Bytes), the presence of Keccak IP with a wide buffer can improve the reduction on the overall energy consumption (23.3%) of the proposed architecture when executing Ethereum API. It can also be observed that accelerating the ECPM and Keccak operation with wide buffer consumes less energy, but takes slightly more time to be executed than in the case when only ECPM is accelerated (total execution time reduction 16.3% against 18%).

The results of hashing data of significant size (tab. 5.5, p.146) also clearly show the need for the Keccak hash IP with wide buffer in the proposed architecture. When the Ethereum API is used to send large sized data the presence of Keccak IP with wide buffer is essential.

It can be also concluded that each IP producing a hash requires a wide buffer in order to optimize the overall power consumption and avoid the over consumption of the architecture when the size of the data to send to the blockchain increases. However, the total execution time can be higher then executing the hash function by software.

5.4.2.2 Hyperledger Sawtooth API

This section analyzes the overall energy consumption of the proposed architecture when the Hyperledger Sawtooth API is executed. Similarly to Ethereum, multiple types of strategies are applied in order to find out which strategy can be the most efficient in terms of energy consumption. A previous section described that Hyperledger Sawtooth applies a double digital signature (one on the transaction and one on the batch). This phenomenon also affects the overall energy consumption. Hyperledger Sawtooth also uses two different hash algorithms SHA-512 and SHA-256. In order to observe how the acceleration of these hash algorithms affects the overall consumption, the measured power management strategies are the follows:

First, only the ECPM operation (x2) is accelerated, in a second time in addition to the ECPM, both hash functions are accelerated. In the third strategy the hash IPs are equipped with a wide buffer (10 internal hashes can be produced in raw until an IRQ is generated).

The overall energy consumption curves according to the different power management strategies are represented in figure 5.21. The purple curve represents the energy consumption when the power management is not applied. The green curve shows the consumption when ECPM operation is accelerated alone (only the ECPM IP is called). On this curve, the two regions in which the slope of the curve increases differently correspond to ECPM IP calls (circled in green). The curves blue and yellow represent the energy consumption when the SHA IPs are also called. Yellow curve shows the consumption when the IPs are equipped with wide buffers.

The mean of ten measurements and the percentage of the possible energy consumption and total execution time reduction is represented in tab 5.8.

It can be observed that when only the ECPM is accelerated, a significant 60.32% of reduction of the overall energy consumption can be achieved. The total execution time can also be reduced by 52.9%. When the SHA-256 and SHA-512 basic IPs are also used to accelerate the corresponding hash functions, less efficient energy consumption (34.1% of reduction) and total time execution (3.5% of reduction) can be obtained than in the case when only the ECPM has been accelerated. The decrease in performance is due to the same

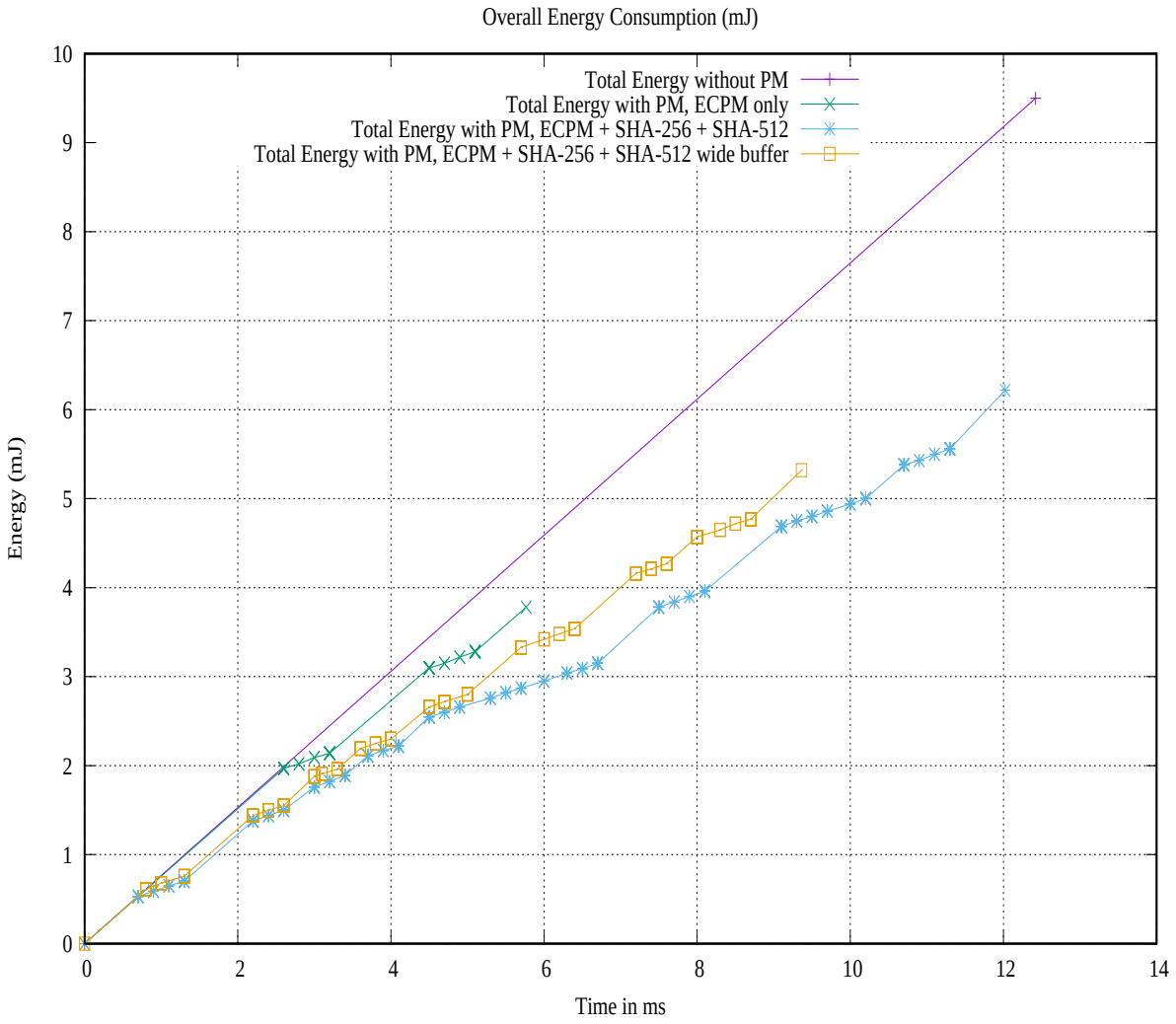


Figure 5.21: Overall Energy Consumption of Hyperledger Sawtooth API, retrieved by using PwClkARCH. *PM* in the titles of the curves means Power Management.

Power Management (PM)	Overall Energy Consumption	Overall Energy Reduction	Total Execution Time	Total Time Reduction
No PM applied	9.5 mJ	0%	12.42 ms	0%
ECPM only	3.769 mJ	60.32%	5.85 ms	52.9%
ECPM+SHA-512+SHA-256	6.26 mJ	34.1%	11.99 ms	3.5%
ECPM+SHA-512+SHA-256 wide buffer	5.37 mJ	43.47%	9.5 ms	23.5%

Table 5.8: Final: Overall Energy Consumption of the proposed architecture when running Hyperledger Sawtooth API. Three different power management were applied.

OS latency issues that were described previously.

Using hash IPs with wide buffers improve the reduction of energy consumption and execution time compared to the case in which hash IPs do not have wide buffers (overall energy consumption reduction of 43.47% against 34.1%, and total execution time 23.5% against 3.5%).

However, it is still less effective energy consumption and execution time than ECPM's

hardware acceleration alone. This is because, in this case, a simple low sized (32 bytes) payload message had to be sent to the blockchain. That also means that SHA-512 was called only once while producing the payload's hash. Same for SHA-256 which is called to create only 8 hashes in row when hashing the transaction's header.

When the hash functions are called only a few times, the performance of the overall consumption cannot be more efficient than accelerating the ECPM operation alone.

When the payload size is more significant, the SHA-512 function is more solicited. Thus the overall consumption can be better reduced. In the case of SHA-256, this function is called more when the input-output structure of the smart contract is more complicated (the number of the input/output variables increases, explained on page 67).

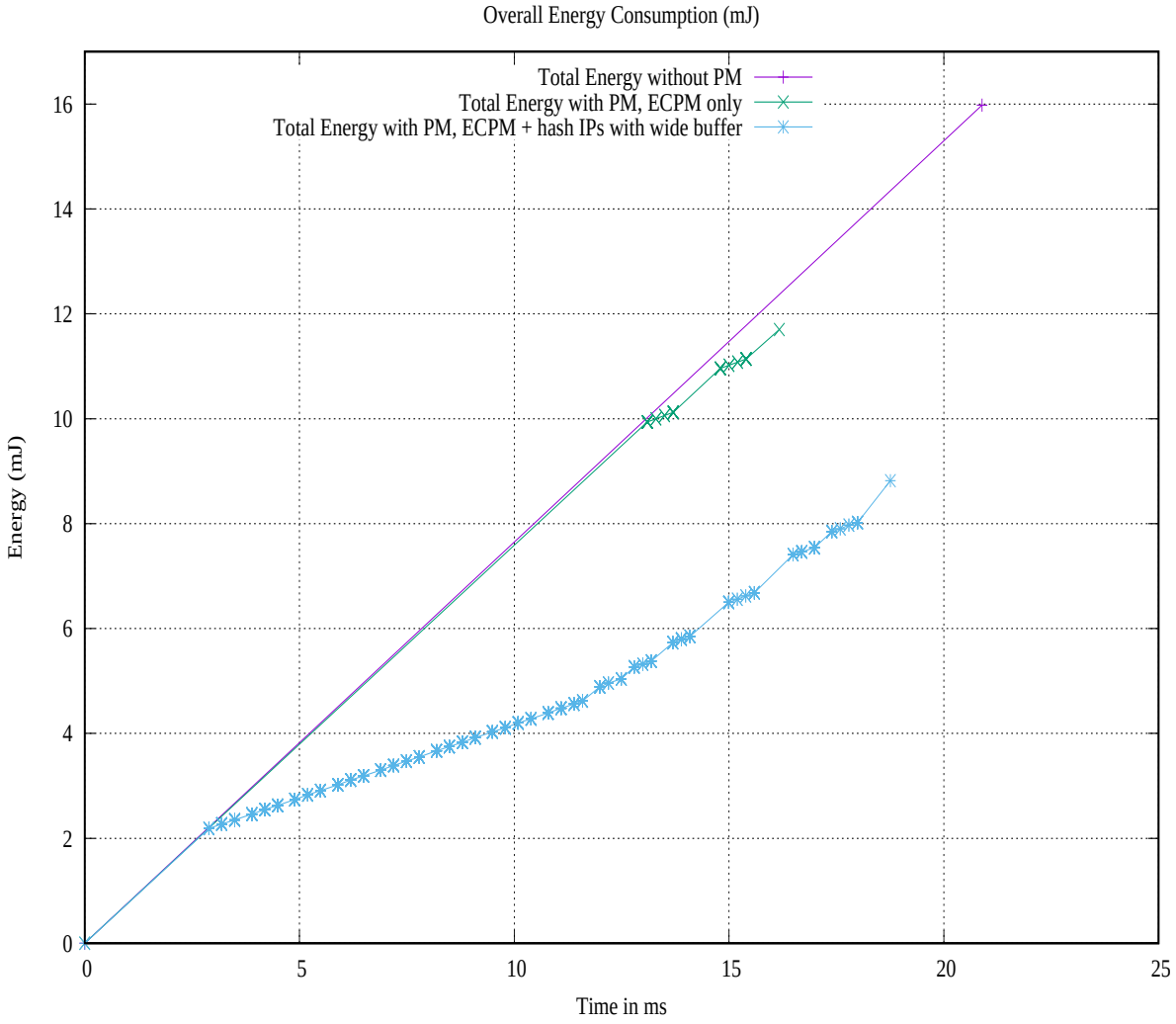


Figure 5.22: Overall Energy Consumption of Hyperledger Sawtooth API when the payload's size is 32KBytes, retrieved by using PwClkARCH. *PM* in the titles of the curves means Power Management.

In Ethereum API, the Keccak IP with a wide buffer has already shown a slightly better performance than accelerating only the ECPM operation. However, in the case of Hyperledger Sawtooth API hardware accelerating, only the ECPM operation seems more efficient in energy consumption an in execution time when the payload size is small.

In order to demonstrate that the hash IPs with wide buffers can achieve better energy

consumption when the payload size increases a 32 KBytes payload (1000 times higher than in the previous case) is applied. Figure 5.22 represents the overall energy consumption curve and total execution time when only the ECPM operation is accelerated and ECPM and also the SHA hash functions with wide buffer IPs. On the blue curve, the hash performing of the payload can be observed in the period from 3-11.5ms. Tab 5.9 represents the mean of ten measurements of overall energy consumption and total execution time. The objective of these measurements is to compare the energy consumption and execution time while applying the two strategies (ECPM only, ECPM and SHA accelerations) when the payload size is significant. The results also show the impact on the energy consumption and execution time when using hash hardware accelerators (and not only and ECPM IP). In Tab 5.9, the reduction rate of the energy consumption and execution time when using hash hardware accelerators is deduced according to the comparison with results obtained when no power management is applied.

Power Management (PM)	Overall Energy Consumption	Overall Energy Reduction	Total Execution Time	Total Time Reduction
No PM applied	15.98 mJ	0%	20.89 ms	0%
ECPM only	11.7 mJ	26.8%	16.27 ms	22.1%
ECPM+SHA-256	8.85 mJ	44.6%	18.64 ms	10.8%

Table 5.9: Final Result: overall energy consumption and total execution time of the proposed architecture when running Hyperledger Sawtooth API, when the size of the payload is 32 KBytes. The results help to compare the power management strategies: when the SHA functions are accelerated by hardware and executed by software.

According to the results represented in Tab 5.9 it can be deduced that using hash hardware accelerators can reduce the overall energy consumption by 44.6% compared to the consumption when no power management were applied. The architecture equipped with hash hardware accelerators provides a 17.8% more effective energy consumption than hardware accelerating only the ECPM operation.

It can also be observed that when using the hash hardware accelerators, the overall energy consumption decreases and more effective than hardware acceleration on ECPM only. However, the reduction of total execution time is less efficient, 10.8% compared to 22.1%.

It can be concluded that by using hash hardware accelerators when executing Hyperledger Sawtooth API (with the presence of a significant size of input data), a more optimal overall energy consumption can be achieved but the execution time is longer then in the case when only the ECPM operation is accelerated.

5.4.2.3 EOS.IO API

This section describes the results of the measurements when EOS.IO API is executed on the proposed IoT architecture. Likewise, in the previous measurements, this one also applies two different strategies of power management. In the first strategy, only ECPM is accelerated with calling the corresponding IP. In the second, the SHA-256 hash function is also accelerated with the associated hardware accelerator. Using a wide buffer in this IP is due to the previous results when Ethereum and Hyperledger Sawtooth were executed. It should be noted that the payload size is fixed to 32 bytes such as in the case of Ethereum and Hyperledger Sawtooth APIs.

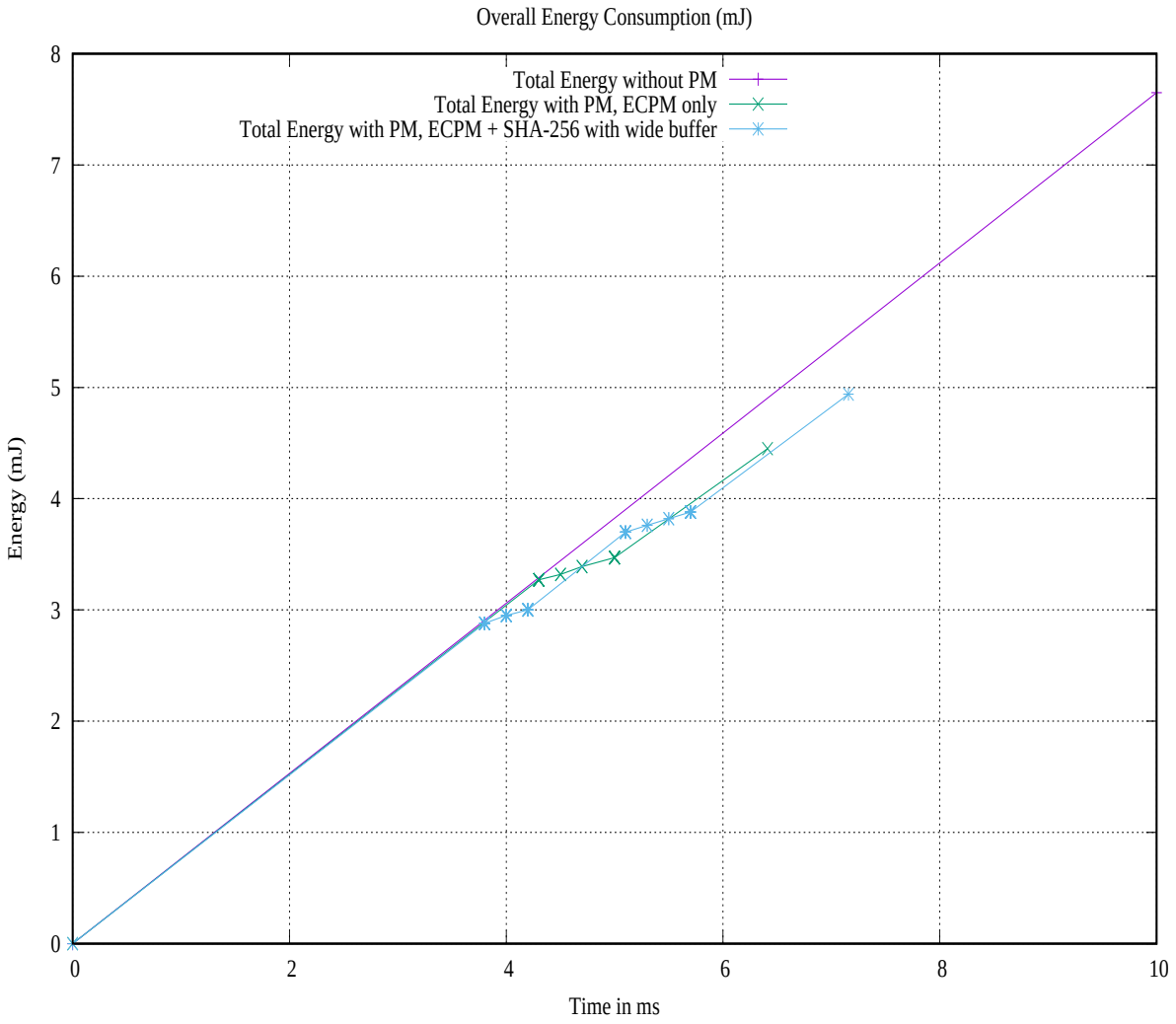


Figure 5.23: Overall Energy Consumption of EOS.IO API, retrieved by using PwClkARCH. *PM* in the titles of the curves means Power Management.

Figure 5.23 represents the curves of the overall energy consumption when no power management is applied (curve in purple), when power management is applied but only the ECPM operations accelerated by the corresponding IP (curve in green). Finally the scenario in which both the ECPM and SHA-256 functions are hardware accelerated (curve in blue).

Tab. 5.10 represents the mean of the overall energy consumption and total execution time of ten measurements. This table also demonstrates the percentage of the reduction in the overall energy consumption and total execution time.

Power Management (PM)	Overall Energy Consumption	Overall Energy Reduction	Total Execution Time	Total Time Reduction
No PM applied	7.65 mJ	0%	10 ms	0%
ECPM only	4.35 mJ	43.13%	6.36 ms	36.4%
ECPM+SHA-256	4.88 mJ	36.2%	7.26 ms	27.4%

Table 5.10: Final Result: overall energy consumption and total execution time of the proposed architecture when running EOS.IO API. Two different power management were applied.

The results show that when the payload size is small, accelerating only the ECPM operation seems more optimal (in terms of energy consumption and execution time) than accelerating the ECPM and also the hash-producing operation. This phenomenon is due to the same reasons as was explained in the two previous sections, the hash operation is not solicited too many times. It can be assumed that when the payload size increases, the SHA-256 IP would be called more, which also means that hardware accelerating this hash primitive would provide more optimal energy consumption. This finding was proved at the end of the previous section (see the figure 5.22, p.152).

5.4.3 Conclusion

The previous sections analyzed the overall energy consumption of the proposed IoT architecture when executing different blockchain-based APIs. The analysis also highlighted how the total execution time changes according to the different power management strategies. In the first imagined IoT architecture model, ECPM and hash IPs were modeled to simulate the essential behaviors of the given hardware IPs. It should be noted that the high-level model also respected the basic requirements on the input and output buffers used in the given hardware IPs. The preliminary results of the proposed power management applied to the proposed architecture clearly showed the interest in the high-level modeling approach. According to the results obtained when the data size is small, better performance can be achieved – in terms of total energy consumption and total execution time – by accelerating only the ECPM operation (thus without the hash IP call). Other preliminary results also showed that the latency issues of the Linux scheduling and IRQ waiting can produce over-consumption when calling hash IPs. These results also showed that the overall consumption cannot be reduced when the payload size increases because of the latency issues.

The advantage of the high-level modeling has clearly appeared at this point of the architecture model deployment by demonstrating that the hash IPs have to be equipped with a wider buffer and with some other components that allow counting the number of internal hashes that have to be performed in a row. It also means that the high-level modeling (early phase of the architecture model deployment) pointed out that the IPs would also have to be modified on low-level (addition of more buffers, counters etc.).

In order to use the hash IPs equipped with wide buffers effectively, the hash IP device drivers had to be slightly modified. However, the imagined power management described in the PMU and the PMDD did not need to be adapted. It can be concluded that a significant overall energy consumption optimization can be achieved on the architecture embedding the ECPM IP by applying the proposed power management in the case when the payload size is low. It can also be concluded that in this small-sized payload case, applying hash IPs equipped with wide buffers are not necessary, the ECPM's hardware acceleration provides a more optimized overall energy consumption, and faster execution in terms of total execution time.

However, when the payload size increases, which is often the case, the presence of the hash IPs equipped with wide buffers is essential. It can be noted that applying hash IPs when the payload size is high, the overall energy consumption can be significantly decreased. However it should be noted that the execution time is longer when the ECPM and the hash functions are hardware accelerated than only hardware accelerating the ECPM operation.

5.4.4 PwClkARCH impact on simulation time

The previous sections represented the efficiency of the proposed power management. It was also mentioned that for measuring and applying the proposed power management to the modeled architecture PwClkARCH library was used. The previous results clearly showed the advantages of this tool, and notably that it is easy to use and can easily be implemented in SystemC designs. However, PwClkARCH can also have an impact on the overall simulation time. Two functionalities of PwClkARCH were measured in order to determine its impact on the simulation time. First, the so-called monitoring is measured, in which PwClkARCH logs out every essential steps that happen during the power consumption. The monitoring is inevitable during the development phase in order to understand and observe each phase of the power management. In a second phase, when the power management is validated, the logs are not needed. In this phase, the power management is ”hidden” and no logs appear. Table 5.11 represents the simulation times (100 measurements were done) when the architecture executed one of the blockchain APIs.

App. Simulation Time Without PwClkARCH	App. Simulation Time With PwClkARCH Monitoring	Slowdown	App. Simulation Time With PwClkARCH Pw Mngt . (No logs)	Slowdown
1.416 s	7.485 s	x 5.286	1.809 s	x1.27

Table 5.11: Impact of PwClkARCH on the overall simulation time. The measurements were retrieved when the proposed architecture executed a blockchain API.

According to the results, when PwClkARCH shows each power management step, the simulation is highly slowed down (x5.3). However, when the logs are avoided, the simulation time increases acceptably (x1.3 times slower simulation).

5.5 Conclusion

This chapter discussed the principal development of this thesis work. The chapter also listed the co-simulation and power management tools that were selected to model a dedicated IoT hardware architecture executing different blockchain APIs. The chapter also highlighted the possible difficulties of synchronizing two completely different time environments, notably the synchronization of QEMU and SystemC-TLM.

The chapter also explained that thanks to the features of the PwClkARCH library, it is possible to associate a power and a clock domain for every IP, which can accelerate one of the cryptographic functions. When an IP is called for accelerating a cryptographic primitive in the proposed power management, the ARM CPU architecture’s frequency is scaled (divided by 3). When the IP is not used, the power gating technique is applied (it is switched off) in order to make the static power consumption equal to zero.

The designs of the cryptographic hardware accelerators listed in the previous chapter are modeled in SystemC-TLM, respecting the basic behaviors of designs’ functioning and the size of input/output registers/buffers.

The read and write accesses cannot be established directly between the blockchain API executed on the ARM CPU and the IPs. In this chapter, different device drivers were proposed to allow these accesses, including the preemption of the given task and IRQ waiting

functionalities. The chapter also proposes a Power Management Device Driver (PMDD), which is implemented to communicate with the Power Management Unit (PMU) and orchestrates the communication with the device drivers of the hardware accelerator IPs.

This chapter also explains the importance of modifying cryptographic libraries, which initiate the call of a given device driver when the given cryptographic function is called. Using the cryptographic libraries avoids any type of modification of the given API related to hardware acceleration or power management.

It must be noted that the famous capacitance and leakage resistance values are required for power consumption measurements and management. The values of the capacitance and the leakage resistance are industrial secrets of a given hardware design. These values had to be estimated and calculated in certain cases in order to obtain an idea of the overall power consumption of the proposed architecture. In the case of the designs of hash IPs (ASIC), the designers declared the designs' total power consumptions. Thus the capacitance and the leakage resistance had to be estimated. However, the static power consumption is unknown, which means that leakage resistance cannot be determined, but only estimated. In order to have an idea of the value of the leakage resistance and the capacitance, this work assumed that the static power consumption is equal to 10% of the total power consumption. In the case of ECPM architectures, the hardware designs are developed in FPGA board, and the power consumptions are not declared. However, the number of electronic components contained by the designs are known. Using Xilinx Power Estimator, it is possible to estimate a given model's dynamic and static power consumption. The ECPM IPs power consumption, according to the results of the Xilinx Power Estimator, are the consumptions of the designs implemented on FPGAs. This work also assumed that the power consumption of these designs, if implemented on ASICs, is estimated by dividing the Xilinx Power Estimator tool results by ten. Similar to the IPs, the values of the capacitance and leakage resistance of the ARM CPU architecture (ARM Cortex-A9) are not publicly available. However, thanks to the Xilinx Power Estimator tool, the power consumption could be estimated, and the capacitance and resistance values calculated.

It must also be noted that all of the power management results are strictly based on the estimated power consumption of the given architecture components (IPs, CPU). This also means that the estimations could be optimistic or even overestimated. The results do not reflect the real power consumption of the architecture that would be realized in an ASIC according to the model of the architecture. However, the profile and the energy consumption behaviors according to the power management are valid, and they can be obtained even if the power consumption estimations were optimistic or not.

The chapter also shows the fundamental importance of high-level modeling. The first results on Ethereum API have shown that power management can optimize the overall energy consumption significantly. However, it was also highlighted that accelerating only the ECPM operation without accelerating the hash operations when the payload size is not important can produce a better energy consumption. The preliminary results pointed out that modifications are required on the hash IPs for obtaining a power-efficient hardware acceleration when performing one of the hash operations. The first results of the high-level modeling highlighted that the hash IPs low-level designs should be slightly modified by adding more

buffers and some logic to be able to perform as many internal hashes in a row as the device driver specifies.

It can be concluded that meaningful energy consumption and execution time reduction can be achieved on the proposed IoT architecture when running a given blockchain API. According to the results, when the payload size is low using only ECPM hardware accelerator can produce a lower overall energy consumption than hardware accelerating ECPM and the hash operations.

It can be noted that in the case when a given blockchain is combined with another DLT such as IPFS, the raw data have to be hashed before sending it to the blockchain. Previously an IoT-Blockchain network architecture was described (p.76) in which the IoT device records the data that is hashed before it is sent to the Hyperledger Sawtooth blockchain. It was also demonstrated the relation between the increasing size of data and the total execution time of the API (see figure 3.17 at page 79). According to the results demonstrated by Tab. 5.9 (p.153), this total execution time and also the overall energy consumption of the proposed IoT architecture can be significantly reduced.

It can be concluded that the hash hardware acceleration is an essential component to optimize the overall energy consumption in blockchain-IoT network structures using the same strategy to store data that was presented on page 76. It must be noted that the energy consumption minimization can be achieved in an architecture using Linux OS, when the hash hardware accelerator designs are equipped with wide buffers.

It can also be remarked that thanks to PwClkARCH the power management can be easily deployed. However, during the development phase, this tool has a quite important impact on the simulation time, but still a much less impact than the values of the global quantum and the *icount* parameters of the co-simulation. Xilinx documentation assumes that the *icount* parameter must be set to 7 when running Zynq architecture, but this can increase the boot time from 30 seconds (when *icount* is equal to 1) to 15 minutes.

Chapter 6

Conclusion and perspectives

This chapter aims to conclude this thesis contribution, make remarks, and give perspectives for future works that could be done or improve the results. The overview of the thesis work can be demonstrated thanks to figure 6.1.

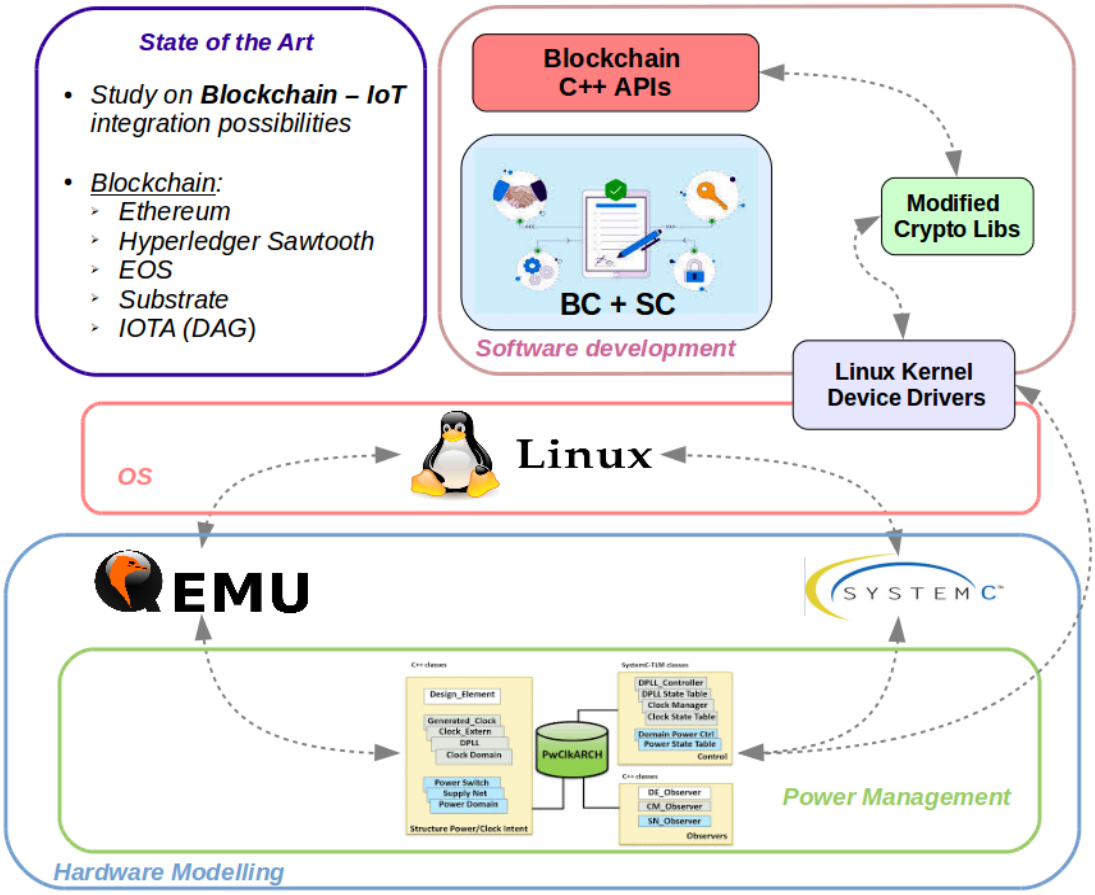


Figure 6.1: The overview of the thesis contribution and challenges.

The top of the figure at the left shows that the thesis has emphasized state of the art about integration possibilities of IoT with blockchain technology. This part also shows that this thesis analyzed different blockchains such as Ethereum, Hyperledger Sawtooth, EOS.IO, and Substrate blockchain framework. This analysis was needed to determine the specific requirements that have to be met for valid blockchain transaction creation. The specification of the requirements allowed the development of IoT APIs (top of the figure at the left) that can pro-

duce valid blockchain transactions enabling communication with the given blockchain. This thesis contributes to Ethereum, Hyperledger Sawtooth and EOS.IO – open-source libraries – which can be applied in IoT devices.

One of the main objectives of this thesis contribution is to model an IoT architecture that can communicate with multiple types of blockchain. This objective was successfully achieved. Thanks to the proposed blockchain APIs, the proposed architecture model enables communication with Ethereum, Hyperledger Sawtooth, and EOS.IO blockchains.

The analysis of the developed and found APIs in related work highlighted that cryptographic hash and digital signature algorithms are necessary for valid blockchain transaction creation. The analysis results showed that cryptographic primitives require a significant amount of computational power, and therefore, their computation takes a significant amount of time.

Another main objective of the thesis is the modeling of a specific low-power consumption IoT architecture. The hardware acceleration of the cryptographic primitives required by blockchain technology can reduce the overall execution time and optimize the overall energy consumption of the proposed IoT architecture. To the best of our knowledge, the idea of creating specific IoT hardware for blockchain-IoT use cases – in which the IoT does not contain the copy of the blockchain but can communicate with the blockchain in a secure way – is unique.

The proposed IoT hardware architecture is the first contribution that is a specific hardware architecture model at a system level for blockchain applications to the best of our knowledge. The proposed IoT architecture contains an ARM-based CPU architecture that is ”surrounded” by cryptographic hardware accelerators. It is important to note that this imagined IoT architecture is modeled (using a high-level modeling approach) and not fabricated. In order to ease the architecture development phase, QEMU (Quick Emulator) is used to emulate the ARM-based CPU (avoiding to the user to model the entire CPU architecture). The hardware accelerators are modeled in SystemC-TLM high-level description language. The complexity is that the QEMU and SystemC-TLM tools use different time domains. Thus they have to be synchronized, which can be done by using co-simulation virtual platforms. The platform used has to be ready to managed the time in SystemC-TLM and the time elapsed in QEMU in order to effectively co-simulate the entire hardware design (the cryptographc primitives, the CPU, the other I/O drivers etc...). The bottom of the figure represents the hardware architecture modeling, surrounded by a blue square, in which QEMU and SystemC-TLM were used as simulation tools.

Another main goal of the thesis is the modeling of a power-managed low-power consumption architecture. The power management (surrounded by a green square) is partially realized thanks to the PwClkARCH C++/SystemC-TLM library. However, when a Linux OS is present on top of the architecture, this deployment of the power management hides several complexities. PwClkARCH realizes a Power Management Unit (PMU) module in SystemC-TLM containing logic that can be commended from an API executed on a CPU. However, in this case, the API cannot access the PMU directly because of the Linux OS. This contribution proposes also for the first time a special device driver which is intended to orchestrate the power management logic and hide this process completely from the application.

The power management results highlighted many essential points that must be considered when hardware accelerates IoT for blockchain applications. The first is that, when the pay-

load size is small, a more optimal energy consumption and faster execution can be achieved when hardware accelerating only the Elliptic Curve Point Multiplication (ECPM) operation than accelerating both ECPM and hash functions. Secondly, when the payload size is important (e.g., when using a hybrid blockchain-IoT structure see 3.8.2 or whenever the internal hashes are in a significant number), it is vital to accelerate in hardware the hash operation. However, the preliminary results of the power management have demonstrated that cryptographic hash hardware designs found in the scientific literature must be slightly modified to achieve a really efficient energy consumption in Linux-based architecture modeled for blockchain use cases.

It must also be noted that in the proposed IoT architecture, Kernel device drivers are needed to be able to use the dedicated cryptographic hardware accelerators. The drivers preempt the write/read accesses when the given IP is computing. The read access can be done when the IP finishes its computation, and the end of the computation is specified with IRQ sending. Task preemption and IRQ waiting must also launch the task scheduling of Linux that is an expensive, time-consuming procedure. Equipping the hash IP designs with wide buffers (FIFOs) can reduce this wait time and considerably improve the overall energy consumption and total execution time on the proposed architecture. Thus, the contribution demonstrated the interest in high-level modeling because, thanks to the preliminary results, the high-level modeled architecture pointed out that the hash IP designs must be modified at a low-level to achieve the optimal overall energy consumption.

The contribution gives the first model of an IoT architecture that can optimize the overall energy consumption of the architecture while using it in an IoT-blockchain use case. The overall energy consumption of the proposed architecture is based on estimation of technological values such as capacitance and leakage resistance. However, the ASIC companies perfectly know these values, which also means that they can determine a more realistic energy consumption thanks to the proposed model. The proposed model must thus be taken into account by ASIC designers in order to improve the architecture. Taking over an existing architecture model is a more money and time saver solution than starting to create a new architecture from zero.

This thesis contribution provides a completely modular IoT hardware architecture model dedicated to IoT-blockchain use cases to optimize the architecture's overall energy consumption. The architecture is modular because all of the components are separated, such as the CPU or hardware accelerators, can be changed and modified. The power management orchestration is separated from the APIs, which could be comfortable for software developers when they aim to modify only the API but not the power management. Furthermore, and vice versa, when the power management logic has to be modified, the API can remain unchanged.

The main objective of this thesis was the development of an IoT architecture model. However, our contribution also includes new IoT-blockchain network architecture, including an InterPlanetary File System (IPFS). In this network architecture, the data storage of IoT devices on the IPFS and the blockchain is controlled by dedicated smart contracts (it is a novel approach). This proposed network has two main advantages. First, it allows the authentication of every IoT device taking part in a given ecosystem. On the other hand, only the hash of the data (fingerprint of the data) is stored on the blockchain; the raw data is sent to the IPFS system. This storage approach slowdown the quickly growing data size of a given blockchain. In our opinion, this network architecture and device authentication and control could be ideal for new ecosystems based on blockchain technology.

6.1 Perspectives

The perspectives of this thesis work are mainly related to the proposed IoT hardware architecture, the proposed blockchain APIs, and the proposed IoT-blockchain network architecture.

The proposed IoT hardware architecture contains a CPU architecture based on a multicore ARM Cortex-A9. This CPU could be changed for a more recent architecture in an improved version of the architecture, such as a multicore ARM Cortex-A53. Changing the architecture could not only provide more significant computational power, but it could also provide new ways of power management. The ARM Cortex-A9 architecture does not contain supply voltage regulators, which means that its supply voltage cannot be scaled. As the supply voltage cannot be modified, only the frequency scaling can modify the dynamic power consumption[1]. The multicore ARM Cortex-A53 allows the scaling of its supply voltage and frequency. A more optimal dynamic power consumption can be achieved by changing the frequency and the supply voltage simultaneously.

The proposed architecture CPU could also be changed to a more constrained CPU such as ARM Cortex-M3 or M0, which cannot execute a Linux OS but can run Real-Time OS or be used as bare-metal. This model could exhibit the energy consumption of a constrained architecture using the specific hardware accelerators for cryptographic primitives. Obviously, in this case, the power management orchestration has to be handled completely differently because the Linux Kernel device drivers could not be used.

The study of the overall energy consumption while a given blockchain API is executed was limited to the valid transaction creation. The transaction sending is dependent on the communication protocol is used. In a future study, the transaction sending phase could also be measured when using different communication protocols (e.g., Wi-Fi, Bluetooth, LoRaWAN, 4G, 5G). This future study could also provide power consumption optimization possibilities during the transaction sending phase. To go further, dedicated low power consumption radio modules could be applied in the future architecture.

Perspectives can also be done in the proposed IoT-blockchain network architecture. More secured communication can be achieved between the IoT and the blockchain and the third entity like IPFS. In all of the proposed IoT-blockchain network architectures, the data (payload included in a transaction) is sent to the blockchain or an IPFS-like system as cleartext. This also means that the payload can be read by man-in-the-middle attacks and also on the blockchain (blockchain is a transparent system). When a payload contains privacy-sensitive data, it cannot be possible to send it as cleartext. Therefore a data encryption cryptographic primitive should be applied. Probably the most logical is to use symmetric encryption, which uses one secret key to encrypt data. This key can be secretly shared (e.g., using Diffie-Hellman Key Exchange [112]) with a system member who aims to access the data.

This also means that the IoT API has to perform an encryption operation that is also expensive in terms of execution time and power consumption. The optimization of energy consumption and the acceleration of the execution can be possible by using dedicated hardware accelerator IPs for encryption primitives. Dalmasso *et al.* [153] demonstrate that PRIME cryptographic cipher can be accelerated more efficiently than the popular AES primitive [110]. The study also demonstrates that PRIME is a more secured primitive against side-channel attacks than AES.

It can also be concluded that the payload sent by the IoT device should be encrypted for

[1]Dynamic Power: $P_{dynamic} = \alpha * C * V^2 * F$, with V the supply voltage and F the applied frequency

achieving a more secure data sharing, and the future IoT architecture should be equipped with a dedicated encryption IP for achieving more efficient overall energy consumption.

Appendix A

Management-Electronics research aspects

At the beginning of this thesis work, it was mentioned that this thesis takes part in the Smart IoT for Mobility multidisciplinary project. It can be noted that synchronizing the research work between the disciplines is probably as challenging as synchronizing QEMU with SystemC-TLM. The main complexity of multidisciplinary research is the technical jargon that can have different meanings in each domain. Another difficulty is the representation of results. Should they be related more or less to one of the disciplines? In which conference could the results be accepted, conferences more related to technical results or more related to perspectives and propositions? These kinds of questions make the multidisciplinary research work complex.

In the case of this thesis contribution, the multidisciplinary research was done principally between the Management and Electronics domains as it was initiated in subsection 1.5.1. The goal of the study is to analyze the requirements of potential ecosystem members, and their inquietude about using blockchain technology. The study also provides technical solution to achieve stronger acceptability and confidence in the blockchain-based ecosystem for these future industrial members.

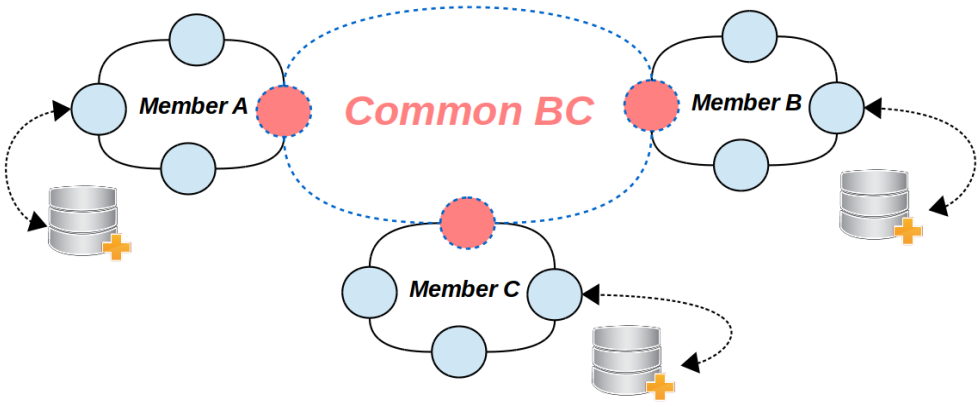


Figure A.1: The ecosystem's members have their own blockchain and local database, and they also connected to a so called common blockchain.

According to the interviews with car manufacturers, companies in the finance sector, and insurance companies the study demonstrated that each of the companies of a future blockchain-based ecosystem could have its own blockchain, but a common blockchain would also needed to bring together all of the members (represented by figure A.1). The common

blockchain can provide a more secure and transparent data sharing among the ecosystem members. The study also highlighted that a new smart contract deployment on the common blockchain has to be accepted by all participants, which could be realized with a voting system smart contract, for example. It was also figured out that the companies could deploy an economy for data sharing between each other and sell the data for crypto-currency.

The study also highlighted that certain data related to a given member could be stored in local storage (and not necessarily to a blockchain). However, it is evident that the hash value of a given data has to be stored on the member's own or on the common blockchain. The analysis has also shown that the common blockchain could also serve as a transparent entity of the overall ecosystem, in which every action that happens in the ecosystem can be stored in a transparent and immutable manner.

The study also encourages to make more studies on blockchains which could allow connecting different blockchains together. After preliminary analysis, the Polkadot blockchain [102] with Substrate [98] blockchain framework could provide interoperability between other blockchains. However, more studies are required to create the first deployment.

Appendix B

SHA-256 device driver

```
1
2
3  /*
       ////////////////////////////////////////////////////////////////////////////////
4  //                  Université Côte d'Azur (UCA) -
5  //                  Centa National de la Recherche Scientifique (CNRS)
6  //                  Copyright © 2020 UCA, CNRS - All Rights Reserved.
7  //
8  //      These computer program listings and specifications, herein, are
9  //      the property of Université Côte d'Azur and CNRS
10 //      shall not be reproduced or copied or used in whole or in part as
11 //      the basis for manufacture or sale of items without written permission
       .
12 //      For a license agreement, please contact:
13 //      <mailto: licensing@sattse.com>
14 //
15 //
16 //
17 //       Author: Roland Kromes, François Verdier
18 //       -- LEAT Laboratory - roland.kromes@univ-cotedazur.fr, francois.
       verdier@univ-cotedazur.fr
19 //
20 //      Description : Kernel Device Driver for SHA-256 IP SystemC-TLM Module
       with IRQs
21 //
22 //      Debug IP SystemC-TLM module (sha_256_IP.cc) generates on port
       SHA_256_IP.irq(zynq.pl2ps_irq[1]);
23 //      This Linux IRQ num is 48 => corresponds to GIC-0 62
24 //
25 ////////////////////////////////////////////////////////////////////////////////

26  */
27
28 #include <linux/module.h>
29 #include <linux/fs.h>
30 #include <linux/slab.h>
31 #include <linux/uaccess.h>
32 #include <linux/ioctl.h>
33 #include <linux/wait.h>
34 #include <linux/sched.h>
```

```c
#include <linux/ioport.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/ioport.h>
#include <linux/io.h>

// To deploy IRQs these libraries are needed
#include <linux/interrupt.h>
#include <linux/of.h>
#include <linux/of_irq.h>


#include <linux/types.h>
#include <linux/ioctl.h>

#define DEV_NAME "dev_sha_256_IP"
static int dev_major;

#define PHYSICAL_ADDR_IP_SHA_256 0x40000300

#define ADDR_IP_SHA_256 ((volatile unsigned int *)(0x40000300))

// DEBUG DEVICE IP REGISTER MAPING

#define SHA_256_DEV_CTRL_REG 0x0 // Control register
#define BUF_IN 0x4          // Input buffer's size is 64 bytes (0x40)
#define BUF_OUT 0x44        // Output buffer's size is 32 bytes (0x20)
#define BUF_BLOCK_LENGTH 0x70  // Lenght of input block 1 byte (size_t)

// DEBUG DEVICE IP CTRL_REG values

#define BUSY 2  // it's computing
#define IDLE 0  // cmputing is done
#define START 1 // command to start computing

#define IRQ_SHA_256 1
#define IRQ_PMU_LINUX 46

// Ioctl adrress, that is called when a Truncated Final Hash (Library
    CryptoPP) is created,
// thus the last block was hashed in an iterative hash function, to obtain
// an optimized energy consumption
#define FINAL_HASH_256_FLAG _IO('a', 'a')

// this variable is equal to one, when the Trunkated Final Hash is done
unsigned int lastBlockIsHashed = 0;

// If the program is in the dev_write() this variable is equal to 1
unsigned int inWriteProcess = 0;

static wait_queue_head_t sha_256_wait_queue;
static int sha_256_wait_queue_flag = 0;


```

```
89 // This variable is equal to 1 on when the first message block (512 bits) has
90 // to be computed, when the last block has been computed the API does a read
91 // in dev_read firstMessageBlock set to 1
92 unsigned int firstMessageBlock = 1;
93
94
95 //Interrupt handler for IRQ 47.
96 static irqreturn_t irq_handler(int irq, void *dev_id)
97 {
98   //unsigned int data = 0;
99   // printk("************ [dev_sha_256_IP] Shared IRQ 47 has Occurred ***************");
100
101   sha_256_wait_queue_flag = 1;
102   wake_up_interruptible(&sha_256_wait_queue);
103
104   //iowrite32(data, ((Virtual_ADDR) + IRQ_TEST));
105   return IRQ_HANDLED;
106 }
107
108
109 // Verify if all necessary parameter are set
110 unsigned int inputIsSet = 0;
111 unsigned int stateIsSet = 0;
112 unsigned int lenghtIsSet = 0;
113
114
115 void __iomem *Virtual_ADDR;
116
117 typedef struct
118 {
119
120   //int in_use;
121
122 } Device;
123
124 static int dev_open(struct inode *, struct file *);
125 static int dev_release(struct inode *, struct file *);
126 static ssize_t dev_read(struct file *, char __user *, size_t, loff_t *);
127 static ssize_t dev_write(struct file *, const char __user *, size_t, loff_t *);
128 static long dev_ioctl(struct file *file, unsigned int cmd, unsigned long arg);
129
130 struct file_operations fops = {
131   .read = dev_read,
132   .write = dev_write,
133   .open = dev_open,
134   .unlocked_ioctl = dev_ioctl,
135   .release = dev_release,
136 };
137
138 static ssize_t dev_read(struct file *f, char __user *buff, size_t len, loff_t *ptr)
```

```
{

  unsigned int *read_value;
  unsigned long err;
  int i;

  inWriteProcess = 0;
  // sha_256_wait_queue_flag = 0;
  //int k;
  // printk("[dev_sha_256_IP] Read Value\n");

  read_value = kmalloc((len / 4) * sizeof(int), GFP_KERNEL);

  // Wait SHA-256 IP's interruption (IRQ 47) indicating that SHA-256 IP has
     finished its computation
  wait_event_interruptible(sha_256_wait_queue, sha_256_wait_queue_flag != 0)
     ;
  sha_256_wait_queue_flag = 0;

  // printk("[dev_sha_256_IP] IP comuting is finished\n");
  // Read of buffer IN => it's a IN/OUT buffer

  for (i = 0; i < (len / 4); i++)
  {
    read_value[i] = ioread32(Virtual_ADDR + BUF_OUT + (i * 4));

    // printk("[dev_sha_256_IP] states[%d] = %d", i, read_value[i]);
  }

  err = copy_to_user(buff, read_value, (len / 4) * sizeof(int));
  firstMessageBlock = 1;


  kfree(read_value);

  return 0;
}

static ssize_t dev_write(struct file *f, const char __user *buff, size_t len
    , loff_t *ptr)
{
  unsigned int *write_value;
  unsigned long err;
  unsigned int i;

  printk("[dev_sha_256_IP] Write Value\n");


  unsigned int buff_len = len / 4;

  write_value = kmalloc(buff_len * sizeof(int), GFP_KERNEL);

  err = copy_from_user(write_value, buff, buff_len * sizeof(int));

  // printk("[dev_sha_256_IP] Error : %lu\n", err);
```

```
191
192   inWriteProcess = 1;
193   // Request a new OPP after the first write is occured
194   // => unnecessary to change OPP when lenght, data and state writing
       occures
195
196   // The data was sent (16*4 bytes => 512 bits, in_vector in SystemC module,
        input in sha.cpp)
197   if (buff_len == 16)
198   {
199
200     // The first message block => don't need to wait an iterruption
      indicating that the IP finished hashing a message block
201     if (firstMessageBlock == 1)
202     {
203       for (i = 0; i < 16; i++)
204       {
205         iowrite32(write_value[i], ((Virtual_ADDR) + BUF_IN + (i * 4)));
206       }
207       firstMessageBlock = 0;
208       inputIsSet = 1;
209     }
210     else if (firstMessageBlock == 0)
211     {
212       printk("[dev_sha_256_IP]  In write witing for IRQ");
213
214       // Wait SHA-256 IP's interruption (IRQ 47) indicating that SHA-256 IP
      has finished its computation
215       // until an itermediate hash has not been done let's wait the
      indicating that the IP finished hashing a message block
216       // after the IRQ we can add a new message blockc as a new input of our
       IP
217       wait_event_interruptible(sha_256_wait_queue, sha_256_wait_queue_flag
      != 0);
218       sha_256_wait_queue_flag = 0;
219
220       for (i = 0; i < 16; i++)
221       {
222         iowrite32(write_value[i], ((Virtual_ADDR) + BUF_IN + (i * 4)));
223       }
224
225       inputIsSet = 1;
226     }
227     else
228     {
229       printk("[dev_sha_256_IP]  Input vector lenght error 1 from API");
230     }
231     // printk("[dev_sha_256_IP] =>>>>>>> DATA");
232
233   }
234   else if ((buff_len) == 8)
235   {
236     // The buffer contains an itermedier hash (state), this value would
      initialize the internal state (hash value) of the IP
237     // used in secp256k1_rfc6979_hmac_sha256
```

```
238     // printk("[dev_sha_256_IP] Write states");
239
240     for (i = 0; i < 8; i++)
241     {
242       iowrite32(write_value[i], ((Virtual_ADDR) + BUF_OUT + (i * 4)));
243       // printk("[dev_sha_256_IP] init_state[%d] = %x", i, write_value[i]);
244
245     }
246     stateIsSet = 1;
247   }
248   else
249   {
250     printk("[dev_sha_256_IP]  Input vector lenght error from API");
251   }
252
253   // printk("[dev_sha_256_IP] BUF_IN is set");
254
255   // If state, lenght and input were set (write to the IP) the IP can start
       to compute
256   if ((inputIsSet == 1))
257   {
258
259     //Set IP to an active state
260     iowrite32(START, (Virtual_ADDR) + SHA_256_DEV_CTRL_REG);
261
262     // printk("[dev_sha_256_IP] SHA-256 IP is READY is ready");
263
264     inputIsSet = 0;
265     stateIsSet = 0;
266     lenghtIsSet = 0;
267   }
268
269   inWriteProcess = 0;
270   kfree(write_value);
271
272   return 0;
273 }
274
275 static int dev_open(struct inode *inode, struct file *file)
276 {
277
278   // printk("[dev_sha_256_IP] Device is open\n");
279   sha_256_wait_queue_flag = 0;
280   return 0;
281 }
282
283 static int dev_release(struct inode *inode, struct file *file)
284 {
285
286   // printk("[dev_sha_256_IP] Release of Device\n");
287
288   return 0;
289 }
290
291
```

```
292
293 /* Nécessaire pour tout module */
294 int init_module(void)
295 {
296   int err = 0;
297
298   // printk("Loading " DEV_NAME "\n");
299   err = register_chrdev(0, DEV_NAME, &fops);
300
301   int result;
302
303   struct device_node *np;
304
305   np = of_find_node_by_name(NULL, "rp_wires_in");
306
307   if (np == NULL)
308   {
309     // printk("[dev_sha_256_IP] Problem while looking for an IRQ");
310   }
311   else
312   {
313
314     // Take care of the number after np => 1 refers to SHA_256_IP.irq(zynq.
    pl2ps_irq[1]);
315     int irq = irq_of_parse_and_map(np, IRQ_SHA_256);
316
317     // printk("[dev_sha_256_IP] ====>>> IRQ num is : %d", irq);
318
319     result = request_irq(irq, (irq_handler_t)irq_handler, 0x81, DEV_NAME, (
    void *)(irq_handler));
320
321     if (result != 0)
322     {
323       // printk("[dev_sha_256_IP] IRQ %d cannot be registered", irq);
324       free_irq(irq, (void *)(irq_handler));
325     }
326     else
327     {
328       // printk("[dev_sha_256_IP] IRQ %d is done", irq);
329     }
330
331     // -- initialize the WAIT QUEUES head
332     init_waitqueue_head(&sha_256_wait_queue);
333
334     result = request_irq(IRQ_PMU_LINUX, (irq_handler_t)irq_handler_PMU, 0x81
    , DEV_NAME, (void *)(irq_handler_PMU));
335     if (result != 0)
336     {
337       // printk("[dev_sha_256_IP] IRQ %d cannot be registered",
    IRQ_PMU_LINUX);
338       free_irq(IRQ_PMU_LINUX, (void *)(irq_handler_PMU));
339     }
340     else
341     {
342       // printk("[dev_sha_256_IP] IRQ %d is done", IRQ_PMU_LINUX);
```

```
343     }
344     // -- initialize the WAIT QUEUES head
345     init_waitqueue_head(&PMU_wait_queue);
346 
347     Virtual_ADDR = ioremap(PHYSICAL_ADDR_IP_SHA_256, 0x200);
348     // printk("Loaded " DEV_NAME " %d\n", err);
349     dev_major = err;
350     return err;
351   }
352 }
353 
354 void cleanup_module(void)
355 {
356 
357   unregister_chrdev(dev_major, DEV_NAME);
358   // printk("Unloaded " DEV_NAME "\n");
359 }
360 MODULE_LICENSE("GPL");
```

Listing B.1: SHA-256 functional device driver.

Bibliography

[1] M. S. Ali, M. Vecchio, M. Pincheira, K. Dolui, F. Antonelli, and M. H. Rehmani, "Applications of Blockchains in the Internet of Things: A Comprehensive Survey," *IEEE Communications Surveys Tutorials*, vol. 21, no. 2, pp. 1676–1717, 2019. xix, 33

[2] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, "On blockchain and its integration with IoT. Challenges and opportunities," *Future Generation Computer Systems*, vol. 88, pp. 173–190, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17329205 xix, 13, 14, 20, 24, 28, 33

[3] O. Novo, "Blockchain Meets IoT: An Architecture for Scalable Access Management in IoT," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, April 2018. xix, 33, 34, 45

[4] M. T. Hammi, B. Hammi, P. Bellot, and A. Serhrouchni, "Bubbles of Trust: A decentralized blockchain-based authentication system for IoT," *Computers & Security*, vol. 78, pp. 126 – 142, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404818300890 xix, 35, 36, 65

[5] M. Pincheira, M. Vecchio, R. Giaffreda, and S. S. Kanhere, "Cost-effective IoT devices as trustworthy data sources for a blockchain-based water management system in precision agriculture," *Computers and Electronics in Agriculture*, vol. 180, p. 105889, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168169920330945 xix, xxiii, 36, 37, 65, 80, 96

[6] M. P. Caro, M. S. Ali, M. Vecchio, and R. Giaffreda, "Blockchain-based traceability in Agri-Food supply chain management: A practical implementation," in *2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany)*, 2018, pp. 1–4. xix, 37

[7] R. A. Michelin, A. Dorri, M. Steger, R. C. Lunardi, S. S. Kanhere, R. Jurdak, and A. F. Zorzo, "SpeedyChain: A Framework for Decoupling Data from Blockchain for Smart Cities," in *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, ser. MobiQuitous '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 145–154. [Online]. Available: https://doi.org/10.1145/3286978.3287019 xix, 38, 39

[8] C. N. Samuel, S. Glock, D. Bercovitz, F. Verdier, and P. Guitton-Ouhamou, "Automotive Data Certification Problem: A View on Effective Blockchain Architectural

Solutions," in *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2020, pp. 0167–0173. xix, 39, 40, 46, 47

[9] L. Gerrits, R. Kromes, and F. Verdier, "A True Decentralized Implementation Based on IoT and Blockchain: a Vehicle Accident Use Case," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020, pp. 1–6. xix, 52, 56, 65, 72, 76, 77, 79, 104

[10] K. Thirumalesu and R. Sakthivel, "An efficient hardware implementation of the elliptic curve cryptographic processor over prime field,," *International Journal of Circuit Theory and Applications*, vol. 48, no. 8, pp. 1256–1273, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cta.2759 xx, 91, 95

[11] A. M. Awaludin, H. T. Larasati, and H. Kim, "High-Speed and Unified ECC Processor for Generic Weierstrass Curves over GF(p) on FPGA," *Sensors*, vol. 21, no. 4, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/4/1451 xx, xxiii, 92, 93, 95, 96, 105, 124, 125, 142, 143

[12] M. A. Mehrabi, C. Doche, and A. Jolfaei, "Elliptic Curve Cryptography Point Multiplication Core for Hardware Security Module," *IEEE Transactions on Computers*, vol. 69, no. 11, pp. 1707–1718, 2020. xx, 86, 92, 93, 95, 105, 125, 142, 143

[13] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "Cryptographic Accelerators for Digital Signature Based on Ed25519," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 7, pp. 1297–1305, 2021. xx, 93, 94, 95

[14] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009. xx, 4, 8, 21, 84, 85, 86, 87, 97, 99, 101

[15] L. Osborne, J. Brummond, R. Hart, M. Zarean, P.E, and S. Conger. Systems Engineering Process. (2005, October). [Online]. Available: https://commons.wikimedia.org/wiki/File:Systems_Engineering_Process_II.svg xx, 107

[16] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the ground up*. Springer Science & Business Media, 2009, vol. 71. xx, 7, 108, 110

[17] R. Kromes, L. Gerrits, and F. Verdier, "Adaptation of an embedded architecture to run Hyperledger Sawtooth Application," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2019, pp. 0409–0415. xx, xxiii, 56, 65, 72, 73, 74, 102, 103, 104, 115, 116

[18] E. E. Iglesias. LibSystemCTLM-SoC. [Online]. Available: https://github.com/Xilinx/libsystemctlm-soc xx, 117

[19] R. Kromes and F. Verdier, "IoT Devices Hardware Modeling for Executing Blockchain and Smart Contracts Applications," in *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, 2019, pp. 1–6. xxiii, 31, 32, 101, 103, 104, 115

[20] https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide. 2, 13

[21] https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/. 2, 13

[22] NUCLEO-F446RE development card. [Online]. Available: https://www.st.com/en/evaluation-tools/nucleo-f446re.html 3

[23] Raspberry Pi 3B+. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ 3

[24] S. Chhabra, P. Mor, H. F. Mahdi, and T. Choudhury, *Block Chain and IoT Architecture*. Cham: Springer International Publishing, 2021, pp. 15–27. [Online]. Available: https://doi.org/10.1007/978-3-030-65691-1_2 3

[25] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008. 3, 5, 15, 19

[26] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, 2014. 4, 5, 19, 21, 46, 48

[27] V. Gramoli, "From blockchain consensus back to Byzantine consensus," *Future Generation Computer Systems*, vol. 107, pp. 760–769, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17320095 4, 19, 20

[28] F. Yang, W. Zhou, Q. Wu, R. Long, N. N. Xiong, and M. Zhou, "Delegated Proof of Stake With Downgrade: A Secure and Efficient Blockchain Consensus Algorithm With Downgrade Mechanism," *IEEE Access*, vol. 7, pp. 118 541–118 555, 2019. 5, 19, 21, 50

[29] C. Dwork and M. Naor, "Pricing via Processing or Combatting Junk Mail," in *Advances in Cryptology — CRYPTO' 92*, E. F. Brickell, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–147. 5, 19

[30] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, p. 398–461, Nov. 2002. [Online]. Available: https://doi.org/10.1145/571637.571640 5, 20

[31] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 30:1–30:15. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190538 5, 20

[32] N. Szabo, "Formalizing and Securing Relationships on Public Networks," *First Monday*, vol. 2, no. 9, Sep. 1997. [Online]. Available: https://firstmonday.org/ojs/index.php/fm/article/view/548 5, 21

[33] Statista. Size of the Bitcoin blockchain from January 2009 to May 20, 2021. [Online]. Available: https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/ 6, 74

[34] F. Bellard, "QEMU, a fast and portable dynamic translator." in *in USENIX Annual Technical Conference*, 2005, p. 41–46. 7, 108, 112

[35] Xilinx, "Xilinx Quick Emulator User Guide (UG1169)," Xilinx, Tech. Rep., 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1169-xilinx-qemu.pdf 7, 117

[36] J. Komlodi. Co-simulation. [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/862421112/Co-simulation 7, 117

[37] "IEEE Standard for Design and Verification of Low Power Integrated Circuits," *IEEE Std 1801-2009*, pp. 1–218, 2009. 7, 112

[38] O. Mbarek, A. Pegatoquet, and M. Auguin, "A Methodology for Power-Aware Transaction-Level Models of Systems-on-Chip Using UPF Standard Concepts," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*, J. L. Ayala, B. García-Cámara, M. Prieto, M. Ruggiero, and G. Sicard, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 226–236. 7, 113

[39] B. Preneel, "Cryptographic hash functions," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 431–448, 1994. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.4460050406 8, 21, 97

[40] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, Aug 2001. [Online]. Available: https://doi.org/10.1007/s102070100002 8, 85, 86, 87

[41] F. Verdier. Smart IoT For Mobility, A research project funded by French (ANR) agency. [Online]. Available: https://univ-cotedazur.eu/sim 9

[42] J.-M. Robert and E. Brangier, "What Is Prospective Ergonomics? A Reflection and a Position on the Future of Ergonomics," in *Ergonomics and Health Aspects of Work with Computers*, B.-T. Karsh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 162–169. 10

[43] H. Affes, A. B. Ameur, M. Auguin, F. Verdier, and C. Barnes, "An ESL framework for low power architecture design space exploration," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016, pp. 227–228. 10, 113

[44] D. Minoli and B. Occhiogrosso, "Blockchain mechanisms for IoT security," *Internet of Things*, vol. 1-2, pp. 1–13, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660518300167 13, 14

[45] Z. Zibin, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 4, pp. 352–375, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17329205 16, 18, 20

[46] L. Dalmasso, "De la vulnérabilité des nœuds capteurs à la certification des transactions sur le réseau, une approche de la sécurisation de l'Internet des Objets," Ph.D. dissertation, Université de Montpellier, 12 2020. 18, 31

[47] C. Cachin and M. Vukolić, "Blockchain Consensus Protocols in the Wild," *arXiv e-prints*, p. arXiv:1707.01873, Jul. 2017. 18

[48] L. Lamport, R. Shostak, and M. Pease, *The Byzantine Generals Problem*. New York, NY, USA: Association for Computing Machinery, 2019, p. 203–226. [Online]. Available: https://doi.org/10.1145/3335772.3335936 18

[49] M. Sadek Ferdous, M. Jabed Morshed Chowdhury, M. A. Hoque, and A. Colman, "Blockchain Consensus Algorithms: A Survey," *arXiv e-prints*, p. arXiv:2001.07091, Jan. 2020. 19, 20

[50] N. Massiera, "Projet Smart IoT for Mobility," Master's thesis, Université Côte d'Azur, 2018. 19, 29

[51] O. Kelly, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, and C. Montgomery, "Sawtooth: an introduction," The Linux Foundation, Tech. Rep., 2018. [Online]. Available: https://www.hyperledger.org/wp-content/uploads/2018/01/Hyperledger_Sawtooth_WhitePaper.pdf 20

[52] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi, "On Security Analysis of Proof-of-Elapsed-Time (PoET)," in *Stabilization, Safety, and Security of Distributed Systems*, P. Spirakis and P. Tsigas, Eds. Cham: Springer International Publishing, 2017, pp. 282–297. 20

[53] Intel. Hyperledger Sawtooth Documentation, PoET 1.0 Specification. (2018, August). [Online]. Available: https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html 20

[54] D. Larimer. Delegated Proof-of-Stake (DPOS). [Online]. Available: http://107.170.30.182/security/delegated-proof-of-stake.php 20

[55] C. Walter. Delegated Proof of Stake (DPoS). [Online]. Available: https://tokens-economy.gitbook.io/consensus/chain-based-proof-of-stake/delegated-proof-of-stake-dpos 21

[56] EOS V2.0 Developers. [Online]. Available: https://developers.eos.io/welcome/v2.0/index 21, 56

[57] B. Schneier and P. Sutherland, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. USA: John Wiley & Sons, Inc., 1995. 21

[58] I. Grigg, "The Ricardian contract," in *Proceedings. First IEEE International Workshop on Electronic Contracting, 2004.*, 2004, pp. 25–31. 23

[59] L. Gerrits, "Comparative study of EOS and IOTA blockchains in the context of Smart IoT for Mobility," Master's thesis, Université Côte d'Azur, 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02935207 23, 38

[60] Q. Lu and X. Xu, "Adaptable Blockchain-Based Systems: A Case Study for Product Traceability," *IEEE Software*, vol. 34, no. 6, pp. 21–27, 2017. 23

[61] S. Ghaffaripour and A. Miri, "Application of Blockchain to Patient-Centric Access Control in Medical Data Management Systems," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2019, pp. 0190–0196. 24

[62] e-estonia guide. [Online]. Available: https://e-estonia.com/wp-content/uploads/e-estonia-guide-210820.pdf 24

[63] K. Salah, M. H. U. Rehman, N. Nizamuddin, and A. Al-Fuqaha, "Blockchain for AI: Review and Open Research Challenges," *IEEE Access*, vol. 7, pp. 10 127–10 149, 2019. 24

[64] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. M. Leung, "Decentralized Applications: The Blockchain-Empowered Software System," *IEEE Access*, vol. 6, pp. 53 019–53 033, 2018. 24

[65] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," in *2017 IEEE International Congress on Big Data (BigData Congress)*, June 2017, pp. 557–564. 24

[66] V. Buterin. (2015) On Public and Private Blockchains. [Online]. Available: https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains 24

[67] F. M. Benčić and I. Podnar Žarko, "Distributed Ledger Technology: Blockchain Compared to Directed Acyclic Graph," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1569–1570. 25

[68] The tangle. White paper 1. [Online]. Available: http://www.descryptions.com/Iota.pdf 26

[69] B. C. Florea, "Blockchain and Internet of Things data provider for smart applications," in *2018 7th Mediterranean Conference on Embedded Computing (MECO)*, 2018, pp. 1–4. 26

[70] A. Raschendorfer, B. Mörzinger, E. Steinberger, P. Pelzmann, R. Oswald, M. Stadler, and F. Bleicher, "On IOTA as a potential enabler for an M2M economy in manufacturing," *Procedia CIRP*, vol. 79, pp. 379–384, 2019, 12th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 18-20 July 2018, Gulf of Naples, Italy. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827119302136 26

[71] P. C. Bartolomeu, E. Vieira, and J. Ferreira, "IOTA Feasibility and Perspectives for Enabling Vehicular Applications," in *2018 IEEE Globecom Workshops (GC Wkshps)*, 2018, pp. 1–7. 26

[72] A. Tesei, L. Di Mauro, M. Falcitelli, S. Noto, and P. Pagano, "IOTA-VPKI: A DLT-Based and Resource Efficient Vehicular Public Key Infrastructure," in *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, 2018, pp. 1–6. 26

[73] D. Strugar, R. Hussain, M. Mazzara, V. Rivera, J. Young Lee, and R. Mustafin, "On M2M Micropayments: A Case Study of Electric Autonomous Vehicles," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 1697–1700. 26

[74] A. Gągol, D. Leśniak, D. Straszak, and M. Świętek, "Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes," *arXiv e-prints*, p. arXiv:1908.05156, Aug. 2019. 26

[75] L. Baird, "Hashgraph consensus: fair, fast, byzantine fault tolerance," SWIRLDS, Tech. Rep., 2016. [Online]. Available: https://www.swirlds.com/wp-content/uploads/2016/06/2016-05-31-Swirlds-Consensus-Algorithm-TR-2016-01.pdf 26

[76] L. Baird and A. Luykx, "The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020, pp. 1–7. 27

[77] H. Sun, S. Hua, E. Zhou, B. Pi, J. Sun, and K. Yamashita, "Using Ethereum Blockchain in Internet of Things: A Solution for Electric Vehicle Battery Refueling," in *Blockchain – ICBC 2018*, S. Chen, H. Wang, and L.-J. Zhang, Eds. Cham: Springer International Publishing, 2018, pp. 3–17. 29, 30

[78] E. Fernando, Meyliana, and Surjandy, "Blockchain Technology Implementation In Raspberry Pi For Private Network," in *2019 International Conference on Sustainable Information Engineering and Technology (SIET)*, 2019, pp. 154–158. 29

[79] T. Tantidham and Y. N. Aung, "Emergency Service for Smart Home System Using Ethereum Blockchain: System and Architecture," in *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2019, pp. 888–893. 30

[80] L. Dalmasso, F. Bruguier, A. Lamlih, and P. Benoit, "Wallance, an Alternative to Blockchain for IoT," in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, 2020, pp. 1–6. 30, 43

[81] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A Call Graph Execution Profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, Jun. 1982. [Online]. Available: http://doi.acm.org/10.1145/872726.806987 31

[82] M. Pincheira and M. Vecchio, "Towards Trusted Data on Decentralized IoT Applications: Integrating Blockchain in Constrained Devices," in *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2020, pp. 1–6. 36

[83] M. Müller, S. R. Garzon, M. Westerkamp, and Z. A. Lux, "HIDALS: A Hybrid IoT-based Decentralized Application for Logistics and Supply Chain Management," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2019, pp. 0802–0808. 37

[84] V. Loury, "Groupe Renault tested a blockchain project to go further in the certification of vehicle compliance," 2020. 38

[85] How blockchain automotive solutions can help drivers. [Online]. Available: https://www.bmw.com/en/innovation/blockchain-automotive.html 38

[86] Porsche introduces blockchain to cars. [Online]. Available: https://www.porsche.com/usa/aboutporsche/pressreleases/pag/?id=479342&pool=international-de&lang=none 38

[87] Z. Yang, K. Yang, L. Lei, K. Zheng, and V. C. M. Leung, "Blockchain-Based Decentralized Trust Management in Vehicular Networks," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1495–1505, 2019. 39

[88] K. L. Brousmiche, T. Heno, C. Poulain, A. Dalmieres, and E. Ben Hamida, "Digitizing, Securing and Sharing Vehicles Life-cycle over a Consortium Blockchain: Lessons Learned," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018, pp. 1–5. 40, 41, 42

[89] M. Cebe, E. Erdin, K. Akkaya, H. Aksu, and S. Uluagac, "Block4Forensic: An Integrated Lightweight Blockchain Framework for Forensics Applications of Connected Vehicles," *IEEE Communications Magazine*, vol. 56, no. 10, pp. 50–57, 2018. 42

[90] Road Safety Initiatives. [Online]. Available: https://www.aig.sg/personal/car-insurance 43

[91] Y. Khacef, "Smart IoT For Mobility," Master's thesis, Université Côte d'Azur, 2020. 43, 72

[92] B. Ampel, M. Patton, and H. Chen, "Performance Modeling of Hyperledger Sawtooth Blockchain," in *2019 IEEE International Conference on Intelligence and Security Informatics (ISI)*. IEEE, 2019, pp. 59–61. 46, 52

[93] S. Benahmed, I. Pidikseev, R. Hussain, J. Lee, S. A. Kazmi, A. Oracevic, and F. Hussain, "A Comparative Analysis of Distributed Ledger Technologies for Smart Contract Development," in *2019 IEEE 30th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 2019, pp. 1–6. 46, 52

[94] Z. Shi, H. Zhou, Y. Hu, S. Jayachander, C. de Laat, and Z. Zhao, "Operating Permissioned Blockchain in Clouds: A Performance Study of Hyperledger Sawtooth," in *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2019, pp. 50–57. 46

[95] L. Gerrits, E. Kilimou, R. Kromes, L. Faure, and F. Verdier, "A Blockchain cloud architecture deployment for an industrial IoT use case," in *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*, 2021, pp. 1–6. 46

[96] Hyperledger Sawtooth Official Online Documentation. [Online]. Available: https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html 50

[97] Y. Huang, H. Wang, L. Wu, G. Tyson, X. Luo, R. Zhang, X. Liu, G. Huang, and X. Jiang, "Characterizing EOSIO Blockchain," *arXiv e-prints*, p. arXiv:2002.05369, Feb. 2020. 56

[98] Substrate Developer Hub. [Online]. Available: https://substrate.dev/ 58, 166

[99] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-Speed High-Security Signatures," in *Cryptographic Hardware and Embedded Systems – CHES 2011*, B. Preneel and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 124–142. 71, 89

[100] D. J. Bernstein and T. Lange, "Faster Addition and Doubling on Elliptic Curves," in *Advances in Cryptology – ASIACRYPT 2007*, K. Kurosawa, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 29–50. 71

[101] C. P. Schnorr, "Efficient Identification and Signatures for Smart Cards," in *Advances in Cryptology — CRYPTO' 89 Proceedings*, G. Brassard, Ed. New York, NY: Springer New York, 1990, pp. 239–252. 71

[102] Polkadot is live. [Online]. Available: https://polkadot.network/ 71, 166

[103] Y. K. Lee, H. Chan, and I. Verbauwhede, "Iteration Bound Analysis and Throughput Optimum Architecture of SHA-256 (384,512) for Hardware Implementations," in *Information Security Applications*, S. Kim, M. Yung, and H.-W. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 102–114. 73, 102, 103

[104] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm," *Integration*, vol. 58, pp. 74 – 81, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167926017300755 74, 101, 102, 103, 143

[105] S. Wang, Y. Zhang, and Y. Zhang, "A Blockchain-Based Framework for Data Sharing With Fine-Grained Access Control in Decentralized Storage Systems," *IEEE Access*, vol. 6, pp. 38 437–38 450, 2018. 75

[106] H. R. Hasan, K. Salah, R. Jayaraman, J. Arshad, I. Yaqoob, M. Omar, and S. Ellahham, "Blockchain-Based Solution for COVID-19 Digital Medical Passports and Immunity Certificates," *IEEE Access*, vol. 8, pp. 222 093–222 108, 2020. 75

[107] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System," 2014. 75

[108] M. S. Ali, K. Dolui, and F. Antonelli, "IoT Data Privacy via Blockchains and IPFS," in *Proceedings of the Seventh International Conference on the Internet of Things*, ser. IoT '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3131542.3131563 75

[109] S. Krejci, M. Sigwart, and S. Schulte, "Blockchain- and IPFS-Based Data Distribution for the Internet of Things," in *Service-Oriented and Cloud Computing*, A. Brogi, W. Zimmermann, and K. Kritikos, Eds. Cham: Springer International Publishing, 2020, pp. 177–191. 75, 76

[110] D. Joan and V. Rijmen, "AES proposal: Rijndael," 1999. 78, 162

[111] L. Wang, X. Shen, J. Li, J. Shao, and Y. Yang, "Cryptographic primitives in blockchains," *Journal of Network and Computer Applications*, vol. 127, pp. 43–58, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S108480451830362X 83

[112] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976. 84, 162

[113] M. E. Harold, "Bulletin of the American mathematical society," *IEEE Access*, vol. 44, no. 3, pp. 393–422, 2007. 86

[114] N. Svetlin, *Practical Cryptography for Developers*. Sofia: MIT license, 2018. [Online]. Available: https://cryptobook.nakov.com/ 87, 89

[115] T. Pornin, "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)," RFC 6979, Aug. 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc6979.txt 87

[116] D. H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Feb. 1997. [Online]. Available: https://rfc-editor.org/rfc/rfc2104.txt 87

[117] Bitcoin. Secp256k1. [Online]. Available: https://en.bitcoin.it/wiki/Secp256k1 88

[118] C. P. Schnorr, "Efficient signature generation by smart cards," *Journal of Cryptology*, vol. 4, no. 3, pp. 161–174, Jan 1991. [Online]. Available: https://doi.org/10.1007/BF00196725 89

[119] J. Elbahrawy, J. Lovejoy, A. Ouyang, and J. Perez, "Analysis of Bitcoin Improvement Proposal 340 — Schnorr Signatures," Master's thesis, Masaryk University, 2020. 89

[120] Federal Office for Information Security: Elliptic Curve Cryptography. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03111/BSI-TR-03111_V-2-1_pdf.pdf?__blob=publicationFile&v=2 89

[121] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)," RFC 8032, Jan. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8032.txt 89

[122] N. Smart, "A comparison of different finite fields for elliptic curve cryptosystems," *Computers & Mathematics with Applications*, vol. 42, no. 1, pp. 91–100, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S089812210100133X 91

[123] Y. A. Shah, K. Javeed, S. Azmat, and X. Wang, "A high-speed RSD-based flexible ECC processor for arbitrary curves over general prime field," *International Journal of Circuit Theory and Applications*, vol. 46, no. 10, pp. 1858–1878, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cta.2504 92, 95

[124] K. Javeed, X. Wang, and M. Scott, "High performance hardware support for elliptic curve cryptography over general prime field," *Microprocessors and Microsystems*, vol. 51, pp. 331–342, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933116304124 92, 95

[125] D. J. Bernstein and T. Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. (2013, August). [Online]. Available: http://safecurves.cr.yp.to/rigid.html 92

[126] N. Pirotte, J. Vliegen, L. Batina, and N. Mentens, "Design of a Fully Balanced ASIC Coprocessor Implementing Complete Addition Formulas on Weierstrass Elliptic Curves," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 545–552. 93, 95

[127] I. Kabin, Z. Dyka, D. Klann, N. Mentens, L. Batina, and P. Langendoerfer, "Breaking a fully Balanced ASIC Coprocessor Implementing Complete Addition Formulas on Weierstrass Elliptic Curves," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 270–276. 93

[128] S. S. Roy and D. Mukhopadhyay, "Implementation of PSEC-KEM (secp256r1 and secp256k1) on Hardware and Software Platforms," NTT Corporation, IIT Kharagpur, Tech. Rep., 2012. [Online]. Available: http://cse.iitkgp.ac.in/~debdeep/osscrypto/psec/psec/downloads/PSEC-KEM_prime.pdf 93, 95

[129] G. Chen, G. Bai, and H. Chen, "A High-Performance Elliptic Curve Cryptographic Processor for General Curves Over GF(p) Based on a Systolic Arithmetic Unit," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 5, pp. 412–416, 2007. 93, 95

[130] M. M. Islam, M. S. Hossain, M. K. Hasan, M. Shahjalal, and Y. M. Jang, "FPGA Implementation of High-Speed Area-Efficient Processor for Elliptic Curve Point Multiplication Over Prime Field," *IEEE Access*, vol. 7, pp. 178 811–178 826, 2019. 93, 95

[131] T. Hansen and D. E. E. 3rd, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)," RFC 6234, May 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6234.txt 100

[132] NIST. SHA-3 Selection Announcement. [Online]. Available: https://csrc.nist.rip/groups/ST/hash/sha-3/sha-3_selection_announcement.pdf 101

[133] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak sponge function family main document. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.419.2140&rep=rep1&type=pdf 101

[134] M.-J. O. Saarinen and J.-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)," RFC 7693, Nov. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7693.txt 101

[135] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "BLAKE2: Simpler, Smaller, Fast as MD5," in *Applied Cryptography and Network Security*, M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 119–135. 101

[136] F. K. Gürkaynak, K. Gaj, B. Muheim, E. Homsirikamol, C. Keller, M. Rogawski, H. Kaeslin, and J.-P. Kaps, "Lessons learned from designing a 65nm ASIC for evaluating third round SHA-3 candidates," in *Third SHA-3 Candidate Conference*, 2012. 101, 102, 103, 143

[137] A. Satoh and T. Inoue, "ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS," *Integration*, vol. 40, no. 1, pp. 3–10, 2007, embedded Cryptographic Hardware. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167926005000581 102, 103, 143

[138] M. Srivastav, X. Guo, S. Huang, D. Ganta, M. B. Henry, L. Nazhandali, and P. Schaumont, "Design and benchmarking of an ASIC with five SHA-3 finalist candidates," *Microprocessors and Microsystems*, vol. 37, no. 2, pp. 246–257, 2013, digital System Safety and Security. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933112001639 103, 143

[139] J. F. Rossetti and W. V. Ruggiero, "Hardware implementation for permutation function of multiplication-hardened sponge BlaMka," in *2017 IEEE 8th Latin American Symposium on Circuits Systems (LASCAS)*, 2017, pp. 1–4. 104, 143

[140] L. Shuping and P. Ling, "The Research of V Model in Testing Embedded Software," in *2008 International Conference on Computer Science and Information Technology*, 2008, pp. 463–466. 107

[141] G. Delbergue, "Advances in SystemC/TLM Virtual Platforms : Configuration, Communication and Parallelism," Ph.D. dissertation, Université de Bordeaux, 12 2017. 107, 116

[142] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718 108

[143] F. Ghenassia, *Transaction-level modeling with SystemC*. Dordrecht, The Netherlands: Springer, 2005, vol. 2. 111

[144] C. Barnes, "Verification and validation of wireless sensor network protocol properties through the system's emulation," Ph.D. dissertation, Université Côte d'Azur, 6 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01618142 112, 116

[145] H. Affes, M. Auguin, F. Verdier, and A. Pegatoquet, "A methodology for inserting clock-management strategies in transaction-level models of system-on- chips," in *2015 Forum on Specification and Design Languages (FDL)*, 2015, pp. 1–7. 113

[146] T. Kanamoto, K. Kasai, K. Hatakeyama, A. Kurokawa, T. Nagase, and M. Imai, "A simple yet precise capacitance estimation method for on-chip power delivery network towards EMC analysis," *IEICE Electronics Express*, vol. 17, no. 14, pp. 20 200 198–20 200 198, 2020. 113

[147] bcm2837. [Online]. Available: https://github.com/hiventive/bcm2837 115

[148] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, and C. Jego, "QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, Jan. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01292317 116, 120

[149] A. Oualha, "Modélisation et développement au niveau transactionnel d'une architecture d'un objet communicant avec contrôle de sa puissance dissipée," Master's thesis, Université Côte d'Azur, 2018. 117

[150] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. 126, 128

[151] Xilix. Xilinx Power Estimator (XPE). [Online]. Available: https://www.xilinx.com/products/technology/power/xpe.html 142

[152] G. K. Adam, N. Petrellis, and L. T. Doulos, "Performance Assessment of Linux Kernels with PREEMPT_RT on ARM-Based Embedded Devices," *Electronics*, vol. 10, no. 11, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/11/1331 146

[153] L. Dalmasso, F. Bruguier, P. Benoit, and L. Torres, "Evaluation of SPN-Based Lightweight Crypto-Ciphers," *IEEE Access*, vol. 7, pp. 10 559–10 567, 2019. 162