Accelerating Blockchain Applications on IoT Architecture Models—Solutions and Drawbacks

ROLAND KROMES, Delft University of Technology, the Netherlands
FRANÇOIS VERDIER, Université Côte d'Azur, CNRS, LEAT, France

More and more IoT use cases require trustworthy computing from cloud/back-end services, which cannot necessarily provide a fully trusted execution environment, data immutability, and traceability. The integration of IoT with the blockchain technology is one of the most promising solutions to achieve the previously mentioned features in the IoT networks. Researchers are also interested in integration solutions, and several solutions are already present in the scientific literature. However, there are still some uncertainties in establishing a direct and effective interaction between an IoT device and the given blockchain. In this work, we propose the first IoT hardware architecture model designed to accelerate time-consuming operations of IoT-Blockchain. The proposed IoT hardware architecture model is programmed in SystemC-TLM and can provide a significant reduction in execution time, 53% and 18% when running Hyperledger Sawtooth and Ethereum applications, respectively.

CCS Concepts: • **Computer systems organization** → **Embedded software**; **Peer-to-peer architectures;**

Additional Key Words and Phrases: IoT, blockchain, embedded systems, hardware modelling, SystemC

ACM Reference format:
Roland Kromes and François Verdier. 2024. Accelerating Blockchain Applications on IoT Architecture Models—Solutions and Drawbacks. *Distrib. Ledger Technol.* 3, 2, Article 14 (June 2024), 24 pages.
https://doi.org/10.1145/3626200

1 INTRODUCTION

Blockchain technology is a peer-to-peer network in which all members of the network participants contain the same data. As every participant has the same data, blockchain technology can also be called a database or distributed ledger. Thanks to several cryptographic primitives (e.g., hash, digital signature), the technology provides the immutability of the data and the traceability of every event happening in the Blockchain. The execution of digital codes, the so-called smart contracts that can automatize operations in the Blockchain system, can also be executed in the new generation of blockchains (since blockchain 2.0—Ethereum blockchain [49] is an excellent example). The data contained by the blockchain is immutable. In addition to this unchangeable data feature, all blockchain events are stored and visible to all blockchain network participants. It can be noted that adding new data and interaction with smart contract can be done via transactions. The data adding and smart

R. Kromes and F. Verdier contributed equally to this research.

This work has been supported by the French government, through the ANR-19-CE25-0008, ANR-15-IDEX-0001, ANR-17-EURE-000 and by European Union's Horizon 2020 research and innovation programme under grant agreement No. 101021727 (IRIS).

Authors' addresses: R. Kromes (Corresponding author), Delft University of Technology, Delft, the Netherlands; e-mail: r.g.kromes@tudelft.nl; F. Verdier, Université Côte d'Azur, CNRS, LEAT, Sophia-Antipolis, France; e-mail: francois.verdier@univ-cotedazur.fr.

2769-6472/2024/6-ART14
https://doi.org/10.1145/3626200

contract interactions can be done according to a common agreement of the network participants, which also make the overall system more trustworthy.

The **Internet of Things (IoT)** technology has several definitions. The IoT is the connection and communication of resource-constrained devices (devices with limited computational power, limited memory, limited battery life) following specific automation without human intervention. In general, the intelligence of the automation is provided by a central entity (e.g., server, cloud). The central unit can also be considered a third-party application on which the overall IoT architecture relies. The central unit of such a complex system can also be considered a drawback, because all components and IoT users/owners must trust that given central unit.

The replacement of this third-party (central unit) with a Blockchain can make the overall IoT network structure a more trustworthy environment. According to References [31] and [39], the combination of IoT with blockchain technology can provide data trustworthiness, which cannot be provided by a typical "centralized" IoT network structure. However, integrating IoT-constrained devices with Blockchain technology can introduce specific requirements, such as the computation of resource-intensive cryptographic and the valid transaction creation tasks in these embedded IoT devices. For this reason, we believe that specific IoT-Blockchain APIs and new IoT hardware architectures should be developed to compute these specific operations efficiently in terms of execution time and energy consumption.

According to Reference [21], developing a new architecture or modifying an existing one is based on five primary phases: Requirements, Design, Build, Test, and Operation. The mentioned study demonstrates the cost ratio of fixing an error in the mentioned phases and highlights that error detection and fixing in Design can be from 5 to 20 times less expensive than in the Build, Test, and Operation phases. For the IoT architecture designer teams it is thus a better option to have a methodology that could help to create models of IoT architectures that can be tested in the first early phases. Thanks to this methodology, fewer errors would be produced in the Build phase. Also, fixing an error at the Design phase's cost is more economical than at the end of the overall architecture development.

The novelty of our study is that it proposes two software applications that enable interaction with the Hyperledger Sawtooth and Ethereum blockchains and proposes an IoT hardware architecture model that enables the optimal execution of the previously mentioned applications in terms of execution time. The study also provides dedicated hardware acceleration methods and design modifications. We believe that this proposed hardware architecture model is a generic solution for optimizing the execution time of IoT-blockchain related applications and, thus, other blockchain applications than Hyperledger Sawtooth and Ethereum can also be accelerated. To the best of our knowledge, this is the first work that provides a dedicated IoT hardware model for blockchain-related applications, to achieve a more optimal execution time. The main research objective is to find a hardware acceleration solution to IoT-blockchain-related applications, the simulation of the proposed hardware model, and finally the execution of the APIs on the model.

1.1 Challenges of Integrating IoT with Blockchain

One of the main goals in our point of view of IoT integration with a given blockchain network is establishing direct interaction between the IoT device and the given blockchain. One of the basic elements of valid transaction creation is the digital signature of the transaction. The digital signature (e.g., Elliptic Curve Digital Signature) on the transaction should be produced locally in the IoT device to allow the device's authentication in the blockchain network.

The transaction and payload creation requires applying also cryptographic schemes such as digital signatures and hash algorithms. Executing the required cryptographic schemes on resource-constrained IoT devices can be considered a challenge, because they require high computational power, and their execution time can be significant. The significant amount of time taken by the transaction and payload creation for smart contracts can also be considered problematic to be solved. The longer the time taken for the transaction and payload creation due to cryptographic schemes, the greater the power consumption of the given IoT device. Most IoT devices have limited battery life. To increase their battery lifetime, the procedure of transaction and payload creation should be accelerated. The acceleration possibilities can be achieved at software and hardware levels. Thus, this work focuses on the hardware acceleration possibilities of the essential cryptographic schemes applied in the given IoT-Blockchain applications.

Another challenge in IoT blockchain integration is the amount of data to store on the blockchain. Typical IoT structures collect a high amount of data that must be stored. Storing a significant amount of data on the blockchain causes two main problems. First, data must be duplicated on each peers of the blockchain, which is not economical. For example, today, Bitcoin blockchain contains more than 350 GBytes of data [3], and one Bitcoin transaction takes around 500 Bytes. We can imagine how the size would increase if the simple IoT transaction contain KBytes or MBytes of data. The second is that the quickly increasing data size to store in the blockchain can risk the explosion of the whole infrastructure hosting the blockchain. Results of the related works will be demonstrated that investigate resolving the rapid growth of data size in an IoT-Blockchain context.

1.2 Contributions

This work has two main contributions. The first provides API implementations designed for IoT devices to allow interactions with Ethereum and Hyperledger Sawtooth smart contracts. The proposed APIs are deployed in C/C++ programming languages, because these languages are more optimal for constrained IoT devices. The proposed APIs meet all of the requirements of Ethereum and Hyperledger Sawtooth blockchains to create valid transactions. These requirements include the formatting and encoding of the transaction and payload used by smart contracts. As one of the main goals is establishing direct communication between the IoT device and the given blockchain, the APIs must include several cryptography-related functionalities.

The second main contribution provides a design of a specific IoT hardware architecture model for IoT-Blockchain applications to achieve a better performance when executing the APIs. For achieving a more acceptable performance, our study presents an analysis of the most resource-intensive functions of the APIs related to the cryptographic primitives to highlight which part of these functions can be hardware-accelerated.

It should be noted that this work proposes a model of architecture and not the prototype of real hardware. The specific IoT hardware architecture model is simulated, thanks to a virtual platform combining the QEMU CPU-architecture emulator and SystemC-TLM high-level hardware description language (the tools for modelling the architecture are detailed in Section 6). Executing the API on top of the simulated IoT hardware model highlights the possible performances that can be achieved in a real hardware architecture that would have been constructed according to the model. The study also highlights the main advantages and interests in high-level architecture modeling and virtual platforms.

1.3 Structure of the Article

The article is organized as follows: In Section 2, we briefly discuss the blockchain and IoT technologies, their implementation possibilities, the related work, and the time constraints of valid blockchain transaction creation. Moreover, we introduce the basic requirements of the developed hardware architecture model. Section 4 demonstrates our proposed blockchain-IoT APIs allowing the interaction with Ethereum and Hyperledger Fabric blockchains. Moreover, this section also provides an analysis of the mentioned APIs. Section 5 explains the possibilities of hardware acceleration of blockchain-related cryptographic functions. In this section, we also present existing hardware accelerator modules and explain our choice of hardware accelerator modules that will be used in the proposed IoT architecture model. In Section 6, we present our proposed IoT architecture model, and in Section 7, we show the results obtained when executing the blockchain APIs on the proposed architecture. Finally, we discuss and conclude this article.

2 BACKGROUND

2.1 Blockchain

Blockchain is a peer-to-peer network in which the participants are connected and authenticated in the network. A member can be identified to the network by using its public key. The cryptographic digital signatures are used to verify the provenance of the given transaction containing the data or the amount of the crypto to transfer.

Another particularity of blockchain technology is that all network members contain the exact copy of the same blockchain data (all data and the events on the blockchain). This data is readable to every participant; thus, the technology also provides full visibility and traceability. Data can be added to the blockchain via transactions that are first verified. After the verification, the transactions are put in the newly created block. The blocks are connected between them with their hashes providing an immutable data structure. In most blockchains (for example, in Reference [47]) smart contracts can also be executed that are intended to describe a business logic and automatize particular tasks.

2.2 IoT

Most of the Internet of Things network architectures are based on IoT devices' connections among them and with a third central party application/back-end or entity. IoT includes diverse types of devices, such as constrained embedded systems, edge devices, smartphones (run Linux-based or other lightweight operating systems), and less-constrained devices. According to Reference [12], one of the primary goals of an IoT is the decision-making and control of the IoT environment, which can be done according to a logic applied on the data provided by the devices and sent to the central unit (or back-end service). This dependence can also be considered a weak point of the system because of some possible main issues. First, the data management logic and process can be altered, which can lead to false results. The second weak point is that an untrustworthy member can alter or delete the data stored on the central unit [5]. Another weak point is that the central unit does the IoT device authentication, which can also be problematic, because the authentication process can also be altered.

3 RELATED WORK

IoT-blockchain structures can be applied in supply-chain management use cases. The Agri-Food project from Caro et al. [10] aims to trace agricultural production using blockchain. With the help of blockchain, each phase of the production (planting, growing, packaging, transporting) can be recorded and later audited by the consumer (buyer) of the product. The IoT devices in this scenario can send GPS and environmental information (e.g., temperature while delivering watering information during the lifetime of the plant) about the product.

In another project from Müller et al. [32], the goal is to prove that the products have been delivered from A to B according to the requirements described in the agreement between the actors. To prove that delivery conditions were respected, IoTs send information about the products' environment. **Service Level Agreements (SLA)** can also be deployed, thanks to specific smart contracts in this use case.

IoT-Blockchain structures are also used in smart city applications and use cases. Pincheira et al. [36] propose a blockchain-IoT-based water management system in precision agriculture. In this work, the Ethereum public blockchain replaces the centralized unit (cloud service) of the IoT-network architecture. In this architecture, the IoT device can be considered a blockchain member that produces water consumption measurements and logs these measurements, thanks to an API that sends logs directly to a specific smart contract. The authors also developed smart contracts to realize business logic related to water consumption.

The automotive sector also showed a high interest in applying blockchain-IoT structures in specific use-cases. For example, the car manufacturers Renault [29] and BMW [1] aim to develop digital vehicle passports in which the car's history can be stored (e.g., car's maintenance, information about car's selling). Thanks to blockchain technology, this information can be immutable and auditable by the participants (or future car owners).

Renault car manufacturer also investigates to solve odometer-reading fraud problems. Samuel et al. [40] developed a possible solution in which the mileage readings information of the car (which contains an IoT device) are sent to smart contracts of Ethereum blockchain when the vehicle's engine starts and stops. According to the author's conclusion, the proposed solution can support 11 million vehicles making four trips per day.

Renault also imagined a car accident use case that could also impact other sectors than automotive. A complete ecosystem is based on blockchain and IoT technologies in that use case. The ecosystem contains multiple actors such as Renault car manufacturer, an insurance company, car mechanics, the police. The idea is that the vehicle fleet of Renault will be able to interact with the blockchain-based ecosystem when an accident between the vehicles occurs. In the use case, the vehicles are equipped with IoT devices that can measure the car's environment (speed, photos/videos about events around the car, etc.) and the driver's behaviors (detection of drowsiness, respect of safety distance, etc.) before the accident occurs. Thanks to these data, the insurance companies can accelerate the faulty party detection and reimbursement procedure by using dedicated smart contracts. The car mechanics can also be faster informed of which components are needed to repair the car.

In related work [27] Renault's use case was realized. The Hyperledger Sawtooth blockchain was the core of the system. The IoT devices can directly interact with the smart contract deployed on the blockchain in this application. However, the data measurements related to the car's environment, such as 360-degree photos, are all sent to the blockchain. In this work, we concluded that the proposed IoT-Blockchain network architecture is functional; however, the data related to accidents can take a high amount of size. The large size of the data per transaction can be seen as a disadvantage, because the size of the blockchain data will grow too fast, resulting in the crash of the infrastructure containing the blockchain.

In another study [17], Hyperledger Sawtooth blockchain-IoT structure was combined with an **InterPlanetary File System (IPFS)** [8] to minimize the quickly growing data size of the blockchain. IPFS is a peer-to-peer distributed network such as the blockchain, but contrarily to the blockchain, it does not use a consensus algorithm to add new data to the file system. IPFS is similar to a torrent file system in which not every peer contains the same data. Using this latter advantage, IPFS can be used to send important data sizes in the range of 1 KB–1 GB (car accident data) and use the blockchain to store the pointers of these data. The pointers of the data are the cryptographic hashes of the data (in general, taking 256 bits of size). Smart contracts were developed to ensure the data adding on IPFS network by filtering the messages coming from unregistered IoT devices. Sending only the hash of the IoT data to the blockchain is a more optimal choice in IoT blockchain structures, especially when the size of data to be sent is considerable. In this solution, the IoT device has to compute the hash of the data, which can significantly increase the execution time of the IoT API. A complete study on the car accident use case implementation possibilities in a cloud environment is provided in Reference [16].

3.1 Approach to Follow when Integrating IoT with Blockchain Technology

In realistic implementations, the IoT device should not contain the copy of the blockchain because, first, the required storage placed in the IoT device is insufficient or it can be filled too quickly. Another problem is that when an endpoint (whether it is a PC or IoT device with enough hardware resource) contains the copy of the blockchain, it has to use a client application provided by the given blockchain. The majority of client applications require their execution on top of an operating system. On the contrary, a so-called offline or off-chain solution should be applied in a more realistic implementation. The basics of the off-chain approach for integrating IoT with blockchain technology are presented in Figure 1.

As Figure 1 presents in this integration approach, the IoT device does not contain the copy of the blockchain. However, it uses an API containing all the necessary tools to create valid blockchain transactions. In this figure, it can be noticed that the IoT API also contains a key, which is the private key used for blockchain transaction digital signatures. The blockchain possesses the public key pair of this private key. Thanks to this public key, the signature can be verified if it comes from a blockchain member (a member signs the transaction; thus, the

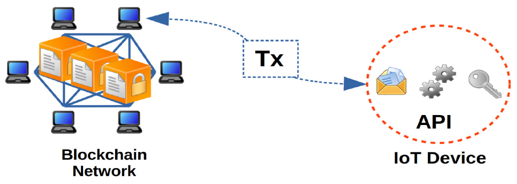

Fig. 1. Principles of the offline/off -chain approach for integrating IoT with blockchain technology.

IoT device's private key was used for signature) or if the transaction was manipulated after its sending. It can be noted also that the registration-enrollment process of new members to the blockchain is done according to the given blockchain policies. In several cases, the blockchain generates the public-private key pairs, but they can also be generated by the new member and added to the blockchain by the admin.

The digital signature creation is thus an essential operation in the offline approach. However, this operation is resource-intensive, and it can take a significant amount of time. The time constraints of digital signature will be detailed in Section 4.

The main advantage of the off-chain or offline approach is that the IoT device has to be connected to the blockchain network only when it aims to send the transactions, which means that the transaction can be sent at any time (obviously, when the networks connection is stable). If the IoT device had contained the copy of the blockchain (the so-called on-chain approach), then sending the transaction would have been possible only when the blockchain data on the device was the same as in the rest of the blockchain network. The same data on the IoT device that we found on the blockchain could be achieved with synchronization requiring a quasi-continuous connection of the device to the blockchain. In IoT applications, the number of communications processed by an IoT device should be minimized as much as possible, because the data (package) sent by activating the device's radio module is expensive in terms of energy consumption.

In the offline approach, the device has to be connected to the blockchain only during the transaction sending, which also means that the number of communications is less important than in the on-chain approach. Thus the offline approach is a more efficient approach for IoT devices.

3.2 Timing Constraints of Valid Blockchain Transaction Creation

This section highlights the possible timing constraints when a given IoT device executes an API intended to create a valid blockchain transaction. These timing constraints are retrieved from our previous studies and the related works found in the literature.

The authors of Reference [35] aim for IoT devices to be are "*direct actors*" on the blockchain network, which also means that the devices sign the blockchain transactions locally in the device. According to the authors, this approach assures the "*root of trust*" for the sensed data by the given IoT devices. This work gave us a great amount of inspiration, because they also used the offline approach to achieve a more secure authentication of devices in IoT-Blockchain network structures.

The authors decided to use Ethereum public blockchain, one of the most popular platforms among users and developers. For achieving the interaction between the device and the blockchain, the authors developed an IoT API written in C language. One of the main contributions of their work is the execution time of the different tasks while running the API on different IoT platforms such as STM32L031K6T6 (Cortex-M0+) and STM32L452 (Cortex-M4).The results highlight that the execution of the transaction signing operation takes more than 95% of the total execution time on both constrained architectures.

The transaction is digitally signed by applying the **Elliptic Curve Digital Signature Algorithm (ECDSA)** [23, 38] used in most blockchain technologies. According to these results, the authors highlighted that the most resource-intensive task of transaction creation is the digital signature process, which should be improved to execute it faster or more efficiently.

It should be noted that the digital signature is performed on the hash value of the data. The hash operation's execution time is dependent on the data size to be hashed. The higher the data size, the longer is the execution. Authors of Reference [27] demonstrated that one-third of the execution time can be taken by the hash operation of a IoT-blockchain API when the data to send exceeds the 1MByte size.

4 PROPOSED IOT-BLOCKCHAIN INTEGRATION

The section is structured as follows: First, it explains the essential requirements of an IoT API development for IoT-Blockchain use cases when the offline integration approach is applied. A second part demonstrates the Hyperledger Sawtooth and Ethereum APIs that we developed and the analysis of the APIs to determine their most-called functions.

4.1 Essential Requirements

In the first initiatives for integrating IoT with a given blockchain, the IoT devices were not the direct "data source" of the blockchain network, because the IoT device did not sign the blockchain transactions. In one of the first such implementations [33], the IoT devices send their sensed data and their identifications via RPC protocol to a dedicated computer hosting an Ethereum blockchain node and an API, converting RPC messages into transactions. In this Proof-of-Concept, it is the PC that signs the transactions and not the IoT devices themselves.

In Reference [20], the authors developed an IoT API in which the transaction's payload only is formed and signed by the IoT device locally. A dedicated smart contract could produce the verification of this payload signature. However, the transaction's signature must be done in a dedicated Ethereum client run on a PC.

In our implementations, we were interested to create all of the transaction components and the transaction signatures by the IoT device. The blockchain network can then verify the transaction's validity, and the IoT device can be authenticated without necessarily running a smart contract. The requirements of valid blockchain transaction creation can differ for each type of blockchain. However, three basic requirements can be considered as expected for most blockchains. All blockchains require a specific type of transaction structure or format to be respected. For example, the Ethereum blockchain uses the **Recursive Length Prefix (RLP)** encoding to differentiate the transactions structure fields. In several cases, the transaction payload should also be encoded or formulated in a specific way to be able to interact with the given smart contract (e.g., Application Binary Interface must be used for interacting with Ethereum blockchain's smart contracts).

All of the IoT APIs must include the cryptographic hash primitive, which is required by the given blockchain (e.g., Ethereum applies the Keccak hash function producing a hash value of 256 bits). The hash value is an essential component of the digital signature, and its importance will be explained in Section 4.5.

The third main component of valid blockchain transaction creation is the digital signature of the transaction. As a heritage of Bitcoin blockchain, most blockchains apply the ECDSA [23]. The particularity of this type of signature algorithm is that the size of its operands is less significant than other traditionally used algorithms like RSA, for example (256/1,024 bits of public key size according to ECDSA/RSA). Depending on the blockchain, the elliptic curve on which the operations are computed can be different (e.g., Ethereum uses the Secp256k1 curve, Substrate blockchain framework applies edwards25519).

We describe in the following the Hyperledger Substrate and Ethereum blockchains and demonstrate the results that we could obtain after analyzing the proposed APIs. A brief description of the analysis is also provided below.

4.2 API Analysis Description

The analysis of the proposed APIs consists of the identification of the most-called functions by the APIs. The analysis results will also highlight the percentage taken by a given function compared to the total execution time of the API. Another advantage of this analysis is that it also highlights the most solicited components of the most-called functions. This information can provide an idea of which parts of these most-called functions can/should be software- or hardware-accelerated.

For the analysis of the APIs, we used a specific software called Callgrind taking part in Valgrind dynamic analysis tool. Thanks to Callgrind profiling tool, we can obtain the call trees or call graphs of the called functions of an executable software running in a Linux environment. When running the given executable, Callgrind catches all memory accesses for tracing the call graphs. Callgrind results can be studied interactively using KCachegrind [48], as this tool allows the visualization (via an interface) of the call graphs that can be browsed afterward.

It must be noted that the analysis studies the valid transaction creation without the sending phase of the transaction to the given blockchain. The sending phase is not measured, because the time taken by sending the transaction can differ according to the communication protocol applied, the transaction validation rate of the blockchain, and the implementation allowing the interaction between the blockchain node and the external application. We have already mentioned previously that according to related works, the transaction's payload size can influence the portion of time taken by the hash computation relative to the total execution time. In the analysis, the transaction size is fixed to 32 bytes, which means that hashing the payload requires only one computation. However, the hash functions are not only used for the payload hashing, as it will be clarified in the following sections.

4.3 Hyperledger Sawtooth API

This subsection briefly introduces the Hyperledger Sawtooth blockchain [25] and explains the obtained result analysis. Hyperledger Sawtooth blockchain is a member of the Hyperledger open-source blockchain family. This blockchain can be implemented as a private or consortium blockchain. The primary purpose of this blockchain is its application in ecosystems or enterprise use cases, because it is based on a modular structure that allows developers to adapt to specific use case requirements easily.

The structural modifications on Hyperledger Sawtooth can also increase the performance in terms of transaction validation rate. For example, this latter can be achieved by changing the Consensus Engine (i.e., applying PBFT or PoET consensus rules will result in different transaction validation rates). The Hyperledger Sawtooth network comprises validator nodes participating in transaction validation and block committing process. A significant advantage of this blockchain is that the interaction from a so-called Client to the blockchain can be established via a REST API integrated into the blockchain structure. The Client application does not have to contain the copy of the blockchain but the basic requirements that as we mentioned on page 7.

Hyperledger Sawtooth provides official Client SDKs in Python, JavaScript, and Rust programming languages. However, there is still no official C/C++ implementation that is more optimal for IoT devices. In this work, we propose a C/C++ open-source API[1] allowing the interaction with Hyperledger Sawtooth's Transaction Processors (Smart Contracts). Smart contracts can be written in almost any language. The only important element is that the transaction payload's encoding format must be the same in both API and Transaction Processor sides (e.g., CBOR).

Hyperledger Sawtooth provides an official Google protocol buffer for the data structure of the transactions and batches. These protocol buffers can be easily converted into several programming languages. In our case, the conversion results in five C++ classes (*Batch*, *BatchHeader*, *BatchList*, *Transaction*, and *TransactionHeader*). In Hyperledger Sawtooth, a batch can contain multiple transactions depending on each other, but it has to contain at least one transaction (in our case, the batch contains only one transaction).

4.3.1 The API's Most-called Functions. The second row of Table 1 shows the results determined from the call graph of a valid transaction creation for Hyperledger Sawtooth, with a transaction payload of 32 bytes. For

[1]Hyperledger Sawtooth optimized API project is available at: https:://github.com/KRolander/HyperledgerSawtooth-cpp-client-optimized

Table 1. Percentages Taken by the Different Functions in the Total Execution Time of the APIs (Hyperledger Sawtooth and Ethereum)

	main()	Setup ContractData()	build signature() / create Transaction()	buildkey Address()	sha256 Data()	sha512 Data()	keccak 256()	SignTx()/ SignTrezor()	ecdsa sign digest()
Sawtooth	100%	X	100%	2.96%	2.52%	4.52%	X	90.87%	90.87%
Ethereum	100%	5.25%	91.93%	X	<1%	X	2.08%	79.29%	79.29%

The symbol "X" specifies that the function in question was not contained in the given blockchain API. Function names are separated by the symbol "/" meaning that the name on the left side is related to Hyperledger Sawtooth, and the right side to Ethereum.

interacting with the Transaction Processor (smart contract), the IntKey transaction processor officially provided by Hyperledger Sawtooth was slightly modified to allow to initialize a variable of 32 bytes size.

The first row of the table contains the main functions that are required for a valid transaction creation in both Hyperledger Sawtooth and Ethereum blockchain. In reality, the *build_signature* function takes 99.37% of the main function, representing 100% of the total execution time of the API. As the *build_signature* function occupies almost the totality of the main function, we selected it as the father function (blue column), taking the totality of the API's execution time. The results show that almost 91% of the total API execution is occupied by the *SignTx* function calling the *ecdsa_sign_digest* function of trezor-crypto[2] open-source C library. The *ecdsa_sign_digest* function realizes the digital signature of the transaction (first signature) and the batch (second signature). This two signatures procedure is specific to Hyperledger Sawtooth, and the secp256k1 elliptic curve is applied. The results also show that the *sha512Data* function corresponding to the SHA-512 cryptographic hash function computation takes 4.52% of the total execution time. This hash algorithm is used to calculate the payload's hash value and create the address encoding needed to access the ledger state variables (address encoding can also be performed by using SHA-256). The *sha256Data* function realizes an SHA-256 hash function computation and is called for performing the hash value of the transaction and the batch. These hash values are one of the main components of the digital signature (i.e., the hash value of the data that is signed not the data itself). The two SHA hash functions above are implemented in Crypto++[3] free and open-source C++ library.

4.4 Ethereum API

Ethereum was the first blockchain that allowed the deployment of smart contracts describing complete business logic not necessarily related to cryptocurrencies [49]. Ethereum is one of the most popular public blockchains among developers, and numerous research works are interested in this distributed ledger. Ethereum eases the development of decentralized applications, because the results of the smart contracts can contain logs for which the application can subscribe. Ethereum's structure is based on miner and client nodes. The miner nodes participate in the consensus rule, which were by default the Proof-of-Work but has been changed for a Proof-of-Stake in June 2022. In this consensus, the miners resolve mathematical problems, and the fastest miner is rewarded. As the miners must be rewarded, the transactions containing an amount of Ether (cryptocurrency of Ethereum) or want to change the state (variable's value in the smart contract) cannot be freely sent (a gas value must be paid). The transaction is also composed of other elements, such as the nonce (number of transactions sent by the account), gas limit (maximum quantity of gas available for the transaction), gas price (amount of crypto paid for each gas unit), smart contract or account address, value of Ether to send, and data for the smart contract.

Ethereum also proposes a JSON-RPC interface to establish the interaction between the smart contracts and the given API. Web3.js is an official JavaScript API allowing the interaction with an Ethereum node. This interface

[2]Trezor-crypto available at: https://github.com/trezor/trezor-crypto
[3]Crypto++ is available at: https://www.cryptopp.com/

Table 2. Percentages Taken by the Different Functions in the Total Execution Time of the Signature Process

ecdsa sign digest	init_rfc6979	scalar_multiply	bn_inverse
100%	3.09%	86.01%	19.79%

is up to date and also easy to use. However, it is not optimal for IoT applications. For this purpose, we developed again a C/C++ open-source IoT API[4] allowing the interaction with Ethereum smart contracts.

Ethereum uses **Ethereum Virtual Machine (EVM)** to run the smart contract written in Solidity language. The transaction's payload can call a specific function in the smart contracts and also provide the arguments for the functions. The function calls and arguments must be encoded following the **Application Binary Interface (ABI)** encoding. The ABI-encoded payload is contained by a transaction that must be RLP encoded and signed. Our proposed API includes the RLP and ABI encoding functionalities and almost every variable types encoding for smart contracts can be applied (e.g., uint8, uint16, uint32, uint256, uint arrays, bytes, and string). Moreover, the API implementation allows signing the transaction and the payload. The payload signature can be useful for creating multi-signatures. Reference [20] inspired us to add this feature to our implementation. In our git repository, we also provide a smart contract that can verify the payload signature.

4.4.1 The API's Most-called Functions. Table 1 summarizes the results of the call graph for a single Ethereum valid transaction creation process when the transaction's payload size is 32 bytes. The payload calls a smart contract function to set the value of a 32-byte variable in the ledger state. The API was tested with the Ganache[5] tool providing a one-click Ethereum blockchain.

The main function of Ethereum API takes 100% of the API's total execution time. All the functions taking less than 2% of the execution time are not presented. The main function is divided into two branches. The first branch starts with the *SetupContractData* function (5.25% of the total execution time), in which the payload will be ABI encoded for the smart contract calling. The second, consists of the *createTransaction* function taking 91.93% of the total execution time. This latter function is composed of three other functions. First, the *RLPEncode* (4%) is called to RLP encode all of the transaction components (nonce, payload, gas price, gas limit, etc.); this function is not presented in the table. After the RLP encoding, the *keccak_256* function (Keccak cryptographic hash function) is called to hash the RLP encoded elements. This function takes around 2% of the total execution time. Contrarily to Hyperledger Sawtooth, for Ethereum transaction signature, the Keccak hash function is applied instead of SHA-256. The third sub-function is the *SignTrezor*, which signs the hash value of the transaction. This function uses the *ecdsa_sign_digest* function of the trezor-crypro library to perform the signature, such as the case in Hyperledger Sawtooth. This signature procedure takes more than 79% of the total execution time, which is a significant portion of the total execution.

We can note that in the case of Ethereum transaction creation, the digital signature procedure takes less time of the total execution compared to the Hyperledger Sawtooth API. However, in Hyperledger Sawtooth API, a double signature is created, one for the batch and one for the transaction contained by the batch.

4.5 Elliptic Curve Digital Signature

To better understand the components of the digital signature function (*ecdsa_sign_digest*) used in the proposed APIs, we provide a table (see Table 2) with the results of call graph of the mentioned function.

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

[5]Ganache one-click blockchain is available at: https://trufflesuite.com/ganache/index.html

The *ecdsa_sign_digest* function is considered taking 100% of the total execution time. We can observe that the major part of this function is taken by the *scalar_multiply* function (85%), which is the point multiplication operation on the elliptic curve. This process is the core operation of the ECDSA digital signature algorithm (more details on how the elliptic curve point multiplication appears in the ECDSA algorithm are given in Section 4.5). As the **elliptic curve point multiplication (ECPM)** operation takes a significant 85% of the *ecdsa_sign_digest* function, it is clear evidence that this is the ECPM operation that must be accelerated/optimized to achieve better performance.

The function *init_rfc6979* takes 3% of the signature function, and it is based on the HMAC algorithm to create a nonce for the signature process.

4.5.1 Conclusion. In this part of this study, we analyzed the proposed IoT APIs, and we could observe that the most resource-intensive part of the APIs is due to the digital signature algorithm. We also highlighted that elliptic curve point multiplication is the core operation of the digital signature algorithm. To achieve better performance, this operation should be optimized by using a software or hardware approach.

In the following section, we collect the existing ECPM hardware accelerators found in the literature, which can eventually be applied in new IoT hardware architectures. We also compare them and select the most optimal hardware accelerator for ECPM. The selected design will be used in our proposed IoT hardware model.

The call graph analysis also highlighted that the hash functions take a small portion of the total execution time. However, it should be noted that the time taken by the hash function can be different when the payload size increases (it was also demonstrated in Reference [17]). The following section also lists some of the existing hardware accelerators for hash function families that can be embedded in new hardware architectures. Our proposed IoT hardware model also contains hardware acceleration for hash functions.

5 HARDWARE ACCELERATION OF IOT ARCHITECTURE

One of the main objectives of this section is to list the existing ECPM and hash hardware designs that can be found in the literature. Another objective is the selection of the most optimal design that can be embedded in our proposed IoT architecture model.

5.1 Elliptic Curve Point Multiplication (ECPM) Hardware Designs

This part of the article provides an introduction about the background of Elliptic Curve Digital Signature algorithm and lists the existing ECPM hardware accelerators with highly detailed description. The conclusion of this part highlights which is the most optimal choice of ECPM hardware accelerator to integrate into the proposed IoT hardware design.

5.1.1 Elliptic Curve Digital Signature Background. In the previous section, we demonstrated that the execution of the elliptic curve digital signature can take almost at least 80% of the total execution time of the given blockchain API. We also demonstrated that the elliptic curve point multiplication operation takes 85% of the signature's execution time. However, we did not detail how the elliptic curve multiplication appears in the signature algorithm. In the following, a brief description is given about the main steps of the ECDSA. We assume that the description of ECDSA is illustrated similarly as in References [23, 34, 46] and RFC-6979 standard is detailed in Reference [38].

The following algorithm demonstrates the ECDSA. The message that has to be signed is denoted as *msg*.

The first step of the algorithm is the computation of the hash of the message to sign (e.g., the SHA-256 hash algorithm). The second step is the secret integer generation (k) derived from the hmac (hash-based algorithm) [26] of the message digest and the private key of the message signer. The private key is also an integer in the range of $[0, \ldots, n-1]$ with n, the order of the generator point G (with x, y coordinates) lying on the given elliptic curve E.

ALGORITHM 1: ECDSA Sign Algorithm

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

This integer is used in the third step for calculating a random point on the given elliptic curve (E). The random point calculation is performed by the elliptic curve point multiplication of the generator point by the secret integer (k). This operation corresponds to adding the point G to itself k times.

Ethereum and Hyperledger Sawtooth apply secp256k1 curve, a Weierstrass form curve that can be defined over the finite field or prime field $\mathbb{F}_p$, with $p > 3$, and can be described by the following equation:

$$E : y^2 = x^3 + ax + b \pmod p, \tag{1}$$

with an imaginary point of infinity ϑ, and with $a, b \in \mathbb{F}_p$, and a condition to respect: $4a^3 + 27b^2 \neq 0 \pmod p$. The fourth step uses the x coordinate of the random point ($R(x, y)$) calculated in the previous step to determine the signature proof (s). The final step represents the signature composed by the x coordinate of the random point ($R(x, y)$) and the signature proof (s). When using the secp256k1 curve, the signature is 64 bytes long.

5.1.2 List of ECPM Hardware Designs. This subsection presents the ECPM designs that we can find in related works. We also provide a Table 3 summarizing the designs' features regarding the type of the implementation, operation latency, the frequency at the design works, and the speedup that can be achieved against the ECPM executed on BCM2837 (Raspberry Pi 3 B+) ARM-based architecture.[6]

It must be specified that the ECPM hardware accelerator designs must allow working with the secp256k1 curve (required by Ethereum and Hyperledger Sawtooth). This requirement also implies that the given design must operate over 256-bits prime field $\mathbb{F}_p$ ($p \leq 256$) and not over the binary field $\mathbb{F}_{2^n}$. The 256-bits prime field is required because the secp256k1 curve is designed to be used with 256-bits operands. Hence, the goal is to find designs compatible with the secp256k1 curve and eventually with other Weierstrass form curves. It must also be noted that secp256k1 is not a NIST-recommended curve, which implies that hardware accelerator designs for NIST-recommended curves cannot be an option in our research.

Most of the related works present FPGA implementations of the accelerator designs, which can also achieve significant speedup and low latency. However, the ECPM hardware accelerators should be implemented in ASIC in future IoT hardware architectures. The design implementation in ASIC will provide at least the same latency and, theoretically, even lower than the FPGA implementation can provide.

Shah et al. [42] mention that the major part of today's designs can work only with NIST-recommended curves, which is a significant drawback. To address this problem, the authors propose an architecture compatible with any of the Weierstrass form curves defined over the general prime field of 256 or 512-bits ($\mathbb{F}_p$ with $256 \leq p \leq 512$). The design implementation on Virtex-6 FPGA operating on curves defined over 256-bits prime field can achieve a significant 0.65 ms of latency at a relatively low frequency of 144.5 MHz (see *Design 1* in Table 3).

The authors of Reference [4] propose an ECC processor for accelerating the ECPM operation. This design is programmable and can be used with any Weierstrass curves (the secp256k1 curve is included) defined over the prime field of 256 bits. The design includes a **pipelined Montgomery Modular Multiplier (pMMM)** to accelerate the operations and a BRAM to store the given curve's parameters. As the curve's parameters can be stored, only the k random integer (see Algorithm 1) must be set before the point multiplication operation. The authors implemented the design on Virtex-7 and XC72020 FPGA boards. The Virtex-7 implementation provides

[6]Specification of Raspberry 3B+: ARM Cortex A53 (quad core), @1.4 GHz, 1 GB RAM.

Table 3. Comparison of ECC Point Multiplication Hardware Accelerators

Design	Impl. Type	Prime p 256-bit / Curve	Latency (ms)	Freq. (MHz)	Speedup	Reference
Design 1	Virtex-6	Any	0.65	221	2.5	[42]
Design 2	XC7Z020	Any	0.206	156.8	4.36	[4]
Design 2	Virtex-7	Any	0.158	204.2	4.74	[4]
Design 3	Virtex-6	Any	2.01	95	1.08	[22]
Design 4	Virtex-US+	secp256k1	0.176	181.8	4.59	[30]
Design 4	Virtex-7	secp256k1	0.25	125	4.06	[30]
Design 5	ASIC	Any	23.06	100	–	[37]
Design 6	ASIC	Any	1.01	556	1.85	[11]

Here, *Any* means any curves of **Weierstrass** form. In orange, the design that is chosen to be used in our IoT architecture model.

a latency of 0.158 ms at 204.2 MHz. The implementation on XC72020 can achieve a slightly higher latency of 0.206 ms but at a less high frequency of 156.8 MHz (see *Design 2* in Table 3).

The hardware design proposed by the authors of Reference [22] allows the ECPM operation on all of the "*secure elliptic curves*" [9] (including the secp256k1 curve) studied and tested by the team of Bernstein. The design implementation on the Virtex-6 FPGA board can provide a latency of 2.01 ms at 95 MHz. Compared to the two previous designs, this design is slower and its latency is higher (see *Design 3* in Table 3).

The authors of Reference [30] propose a secp256k1 curve-specific ECPM design, which means that the design can operate only on this curve. This design is intended to operate on **Residue Number System (RNS)**, which also means that the secret integer (k) and the result (coordinates of the random point R) are in RNS format. Moreover, the coordinates are Jacobian (x,y,z) and not finite coordinates (x,y). The authors also adapted the RNS architecture to multiple methods of ECPM and found that **GLV (Gallant-Lambert-Vanstone)** method of computing scalar multiplication can provide the fastest operation. The design implementation on Virtex-7 FPGA achieves 0.25 ms of latency at 125 MHz of frequency. The design on a more recent board Virtex-UltraScale+ provides a latency of 0.176 ms at 181.8 MHz of frequency (see *Design 4* in Table 3).

Only one ASIC implementation was found among the recent related works. This design was proposed in Reference [37]. This EC co-processor allows using the curves defined over the general field. The latency of this design is high: 23.06 ms at a clock frequency of 100 MHz, which is almost 100 times higher than the other recent FPGA implementations presented above. This high latency is due to the architecture design being resistant against Side-channel attacks. However, another recent work [24] achieved a successful Side-channel attack on the mentioned design (see *Design 5* in Table 3).

In less recent works, the only ASIC design found was published in Reference [11]. The design has a latency of 1.01 ms at a significant 556 MHz. That high frequency cannot be considered useful in IoT architectures (see *Design 6* in Table 3). Table 3 represents the ECPM hardware accelerator designs we found in related works presented above.

The table also contains the possible speedup that can be achieved against the execution of ECPM on BCM2837 (Raspberry Pi 3 B+) ARM-based architecture. The digital signature execution time of ECDSA on the BCM2837 is 2.6 ms. Thanks to the analysis results (see Table 2), we know that 85% of the execution time of the digital signature (T_{total}) on the hash of the message to sign (*ecdsa_sign_digest*) is taken by the ECPM operation. The hardware accelerator is intended to accelerate this 85% of the *ecdsa_sign_digest* function, and its execution time ($T_{HardwareAccelerator}$) corresponds to the design's latency. The 15% left is not accelerated. The speedup can be calculated, thanks to the equation below.

$$Speedup = \frac{T_{total}}{T_{total}'} = \frac{T_{total}}{(T_{total} * 0.15) + T_{HardwareAccelerator}}, \tag{2}$$

with T_{total} the total execution time before the acceleration, and T'_{total} the total execution time after acceleration.

The specifications of the listed designs demonstrate that most of the designs can provide a speedup against BCM2837 quad-core architecture. The design that we aim to use in the IoT architecture model that we present later (Section 6) should provide a high speedup at a relatively low frequency. Comparing these designs is not evident, because the designs are implemented on different FPGA boards and as ASIC. However, we can compare them according to the designs' speedup, frequency, and other features.

We can state that *Designs 5* and *6* (ASIC implementations) are slower than the other designs. It is also evident that *Design 3* cannot be chosen because of its low speedup. It can also be declared that *Design 2* and *Design 4* beat *Design 1* in terms of speedup (almost twice higher) and frequency usage (lower). We can observe that *Design 4* Virtex-Ultrascale+ implementation can slightly beat *Design 2* XC7Z020 implementation in terms of speedup and the *Design 2* Virtex-7 implementation in terms of frequency usage. However, *Design 4* has three drawbacks compared to *Design 2*. First, the design uses Jacobian coordinates. When a software aims to use these results, it must convert them to affine coordinates. Second, the coordinates must also be in RNS representation, which again must be converted on the software side and which can take some additive time. The last drawback is that this design can only be used with the secp256k1 curve. The *Design 2* can be used with any Weierstrass form curve of 256 bits, and this feature can also be considered a significant advantage. In future IoT-Blockchain use cases where the given blockchain uses a different Weierstrass curve than the secp256k1, the design can still be used (only its reprogramming is needed). We chose to use the XC7Z020 implementation of *Design 2* as an ECPM hardware accelerator in our IoT architecture model (see Section 6), because it uses almost 50 MHz less frequency than the Virtex-7 implementation but almost provides the same high speedup. In the following, we consider that this is *Design 2*, which will be used in the proposed IoT architecture model.

5.2 Cryptographic Hash Hardware Designs

The message to be signed in ECDSA must be hashed first. In Section 4, it was also highlighted that the most-used cryptographic functions in Ethereum and Hyperledger Sawtooth blockchains are SHA-256, SHA-512, and Keccak hash functions. To reduce the time needed for the hash creation procedure, we look for existing hardware accelerator designs for SHA and Keccak hash families in the literature. We also compare the designs that we found and select those that can be efficiently used in the proposed IoT hardware architecture model.

5.2.1 Cryptographic Hash Function—Background. The hash function provides a fixed size output value (e.g., 256 bits), which is the unique representation of the input message. The hash function comprises two main components: *Padding* and *Compression Function.*

As the message (m) can have arbitrary size, it must be divided into chunks ($m_1, m_2 ... m_n$) of fixed size (e.g., 512-bits chunks in case of SHA-256). The padding is needed to pad the chunks or last chunk with some additive information. Each of the message chunks is consumed by the *Compression Function,* one after the other. The *Compression Function* provides the internal hash (H_i) of the given message chunk; this hash value is used for producing the next chunk's hash value (H_{i+1}). The hash of the last chunk corresponds to the hash of the message. The most resource-intensive part of the hash algorithms is the Compression Function. Therefore, the hash hardware accelerator designs realize this part of the algorithms.

5.2.2 List of Cryptographic Hash Designs. In this part of our work, we list, compare, and choose the hash hardware accelerators that could be implemented in the proposed IoT architecture model. The following designs were found in the literature, and most of them are implemented as IPs in ASIC. The comparison of the designs is based on their latency, throughput, frequency, and speedup can be achieved against the execution time when the hash is computed on a BCM2837 ARM-based architecture (Raspberry Pi 3 B+).

Table 4. Comparison of ASIC Implementation of Cryptographic Hash Functions

Design	Function	Tp (Gbit/s)	Latency (ns)	Freq. (MHz)	Speedup	Reference
Design 1	SHA-256	4.502	112.64	294.55	24.5	[19]
Design 2	SHA-256	20.9125	24.48	794	112.75	[28]
Design 2	SHA-512	31.836	32	746	293.18	[28]
Design 3	SHA-512	10.18	101	250	97.35	[41]
Design 1	Keccak	44.01	35.2	485.44	890	[19]
Design 4	Keccak	37.345	43	284	715	[43]

The latency and Throughput (Tp) were estimated for 40 nm CMOS technology. The design marked in orange is chosen for accelerating SHA-256 hash function, the design marked in green is chosen for accelerating SHA-512 hash function, and the design marked in yellow is chosen for accelerating Keccak hash function in our IoT architecture model.

These ASIC implementations are based on different CMOS transistor technologies, making it difficult to compare them. Therefore, the CMOS scaling estimation method is applied to estimate the latencies corresponding to the same designs but implemented on 40 nm CMOS technology [44].

The hardware accelerator proposed in Reference [19] allows accelerating the SHA-256, Keccak-256, and Blake hash functions on a single chip. The Verilog source code[7] of the design is still open-source and can be synthesized as ASIC. The design provides a latency of 112.64 ns for one SHA-256 internal hash creation, which is 24.5 times faster (speedup) than the hash creation measured in the BCM2837 ARM-based quad-core architecture. This significant speedup can be achieved when the design works at 294.55 MHz of frequency (see *Design 1* in Table 4).

The hardware accelerator design for SHA-256 hash function published in Reference [28] (*Design 2*) can achieve a significant 112.75 of speedup, which is more significant than the speedup of *Design 1*. However, this design works at 794 MHz, which cannot be considered an optimal solution for IoT architectures. It is worth noting that if the frequency of *Design 1* was the same as *Design 2*, then its latency would have been only half that of *Design 2*. We can conclude that *Design 1* provides a high enough acceleration (speedup) at an acceptable frequency.

The *Design 2* can also accelerate the SHA-512 hash function by achieving a significant 293.18 times speedup against the ARM-based architecture's execution time (9832 ns). However, a high 756 MHz of frequency is required for such a performance. The design (see *Design 3* in Table 3) proposed in this article can provide a less significant speedup than *Design 2*. However, a speedup of 97.35 can still be considered a high enough speedup at the frequency of 250 MHz, almost three times less than the frequency required by *Design 2*. As *Design 2* requires too significant frequency, its implementation in IoT architecture cannot be considered optimal.

Design 1 implements the Keccak hash function's acceleration, and it provides a significant speedup of 890 at a relatively high frequency of 485.44 MHz (execution on ARM-based architecture takes 30,768 ns). Another design (see *Design 4* in Table 4) found in Reference [43] also provide a significant speedup of 715 but at a lower (284 MHz) frequency than *Design 1*. *Design 4* is chosen for our IoT hardware architecture model, thanks to its high performance at an acceptable frequency.

The latencies of the hash function executed on the ARM-based architecture are all implemented in CryptoPP C++ library. In the following parts of this article, we consider using *Design 1* for accelerating the SHA-256 hash function, *Design 3* for accelerating the SHA-512 hash function, and *Design 4* for the Keccak hash function.

6 PROPOSED IOT ARCHITECTURE HARDWARE MODEL

The main objective of this part is to present our IoT architecture model, including an ARM Cortex-A9 CPU architecture presented below and the hardware accelerators selected in the previous section. Our proposed

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

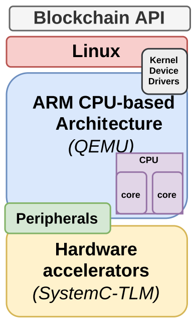
Blockchain API
Linux
Kernel Device Drivers
ARM CPU-based Architecture (QEMU)
CPU
core
core
Peripherals
Hardware accelerators (SystemC-TLM)

Fig. 2. The basic architecture and the requirements.

IoT architecture is modeled with SystemC-TLM high-level modeling tools, detailed briefly in the following subsections.

6.1 Requirements of the Proposed IoT Architecture

As mentioned earlier, most IoT devices have several constraints in computational power, limited memory for storing the data, and limited battery life and applied CPU frequency. One of the first requirements of the architecture that we model is that the CPU frequency should be around 600 MHz. The purpose of the hardware accelerators is to accelerate certain operations. The frequency of these IPs should not exceed 300 MHz, which is the second main requirement of the proposed architecture.

In this article, we propose an IoT architecture model based on an ARM Cortex-A9 Dual-core CPU architecture. Figure 2 also presents the specific peripherals around the CPU architecture that could probably be interconnected through an AMBA BUS model. The maximum frequency that can be used by this CPU architecture is around 600 MHz, which is a constraint compared to today's CPU architectures applied in PCs (2–3 GHz). It should be noted also that the frequency used by the CPU affects the architecture's overall energy consumption, because the power consumption is highly dependent on the applied frequency. Another critical requirement is that the architecture should also be able to run a Linux operating system. This requirement also involves the need for Kernel device drivers' development allowing the interactions with the specific hardware accelerators that can run specific functions faster.

6.2 High-level Hardware Modeling

The main advantage of high-level hardware modeling is the simplicity of the development. In the early phases of high-level modeling, modelling each single bit operation of the architecture is not necessary. The main idea is to model and verify the main functionalities of the required architecture. Another advantage is that the software-level development for the architecture (blockchain IoT-APIs) can be tested in this early phase of the architecture model development. Hence, the developers do not build but model the architecture, so detecting and fixing errors is less expensive than in low-level architecture modeling.

6.2.1 SystemC-TLM. The hardware accelerators identified in the previous section are modeled in SystemC-TLM high-level description language. System-C is an open-source C++ library standardized by IEEE 1666. This tool is usually used in the industrial domain in the early phase development of hardware architectures. The main elements of SystemC are that modules are similar to C++ classes. The modules can be instantiated as objects in C++ language and can contain SC_THREADs or SC_METHODs. The SC_THREADs are executed only one time and can be triggered, thanks to events that are notified according to the logic that the user develops. The SC_METHODS are executed multiple times and triggered when a specific input signal is present. In SystemC,

Table 5. Comparison of the Co-simulation Platforms

	Synchronization's Master	Applying SystemC-TLM *wait(time)*	Functional Model Realization	Timed Model Realization	Open-source Code
Hiventive Platform	*QEMU*	✗	✓	✗	✓
QBox	*SystemC Kernel*	✓	✓	✓	✗
TLM Co-Sim (Xilinx)	*QEMU*	✓	✓	✓	✓

two different time domains are distinguished. The wall clock is the time taken on the host machine while the SystemC simulation is executing. The simulated time is the time viewed by the modules, which is modeled in the design. This also means that timed modeling is possible with SystemC. The time can be modeled by calling the wait() function accepting a femtosecond to second time quantity. In our model, we also used the **Transaction Level Modeling (TLM)** [18] blocking interface ("b_transport") to enable communication between the modules. The TLM blocking interface (**TLM LT** or **Loosely-Timed**) is not the only one used in TLM; the other is the non-blocking interface (**TLM AT** or **Approximate-Timed**), which is more refined and adapted to some specific bus protocols like AXI, for example. This refinement from LT to AT could express the specific out-of-order execution or the sending of data in pipeline before receiving the responses.

6.2.2 QEMU. **QEMU** or **Quick EMUlator** [7] is an open-source CPU architecture emulator including the emulation of peripherals and other components implemented near to the processor. Today, QEMU allows the emulation of more than 200 different architectures such as ARM, SPARK, x86, and many others. The emulation of an architecture (guest machine) is performed by the host machine by running one-by-one on the guest machine's instructions. The "-icount n" option can set the time taken for emulating one instruction (2^n nanoseconds per instruction).

6.2.3 Virtual Platforms. If we look at Figure 2, then we can see that QEMU emulates the ARM CPU architecture, and the hardware accelerators are modeled in SystemC-TLM. SystemC-TLM and QEMU work on different time environments, making it hard to establish communication between the SystemC-TLM and QEMU instances. These instances must be synchronized to achieve a joint simulation; for this purpose, a virtual platform can be used. We have summarized the different features of the three solutions that exist today in Table 5.

The Hiventive Platform [14] emulates a quad-core ARM Cortex-A53 CPU by a QEMU instance, and the peripherals (UART, etc.) are modeled in SystemC-TLM. This platform allows functional modeling; however, when a SystemC instance applies a *wait()* function, the QEMU instance cannot handle the synchronization of the wait time between QEMU and the SystemC instances, which makes the overall simulation stall. Because of this issue, this platform cannot provide the time modeling we were expected.

In QBox platform [6, 13, 15], the master is the SystemC kernel, and the QEMU is treated as a SystemC module. This also means that QEMU and SystemC modules are synchronized, thanks to the global quantum. A synchronization must occur after the initiator module local time reaches the multiple of the quantum value. Thanks to this feature, a wait time applied in a SystemC module can wait in the QEMU without stalling the overall simulation. Unfortunately, this platform is not open-source anymore.

SystemC-TLM 2.0 Co-Simulation (Xilinx) [2, 50] is an open-source project in which a Xilinx modified QEMU is the master of the simulation, but thanks to *libSystemCTLM-SoC* and the Remote-Ports, the QEMU instance can be treated as a SystemC module. The basic idea of this virtual platform is to emulate the **Processing System (PS)** of Xilinx boards (e.g., Zynq, Versal ACAP) and deploy architectures in the **Programmable Logic (PL)** by modules written in SystemC-TLM. The SystemC-TLM 2.0 wrapper encapsulates the PL to enable the Remote Port connection with the PS. When memory transactions or wire updates occur in QEMU, the emulator sends timestamps to the SystemC-TLM instance for synchronization. The periodic synchronizations also occur when the global quantum is achieved. This virtual platform allows the complete time modeling when using the *icount*

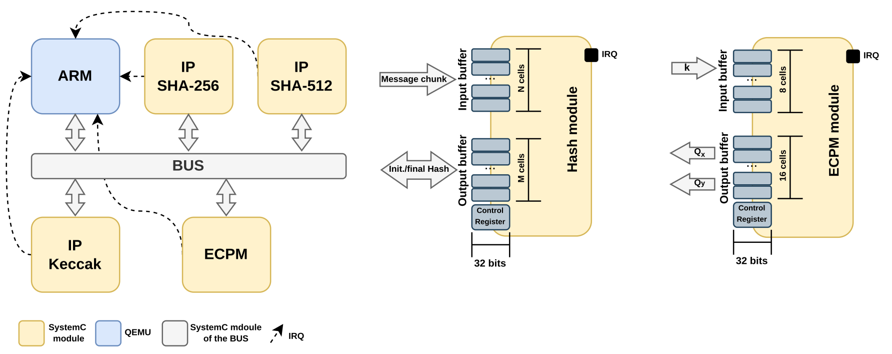

Fig. 3. The proposed IoT architecture model on the left side. The Generic representation of the hash and ECPM modules are on the right side. The number of input buffer cells of the Hash modules is calculated as: N = message chunk's bit length/32 bits; the number of output buffer cells is calculated as: M = hash digest's bit length/32 bits. In the case of ECPM modules, k, Q_x, Q_y are represented in 256-bit.

options at the QEMU side by using the *wait()* function in SystemC modules. Note that the Xilinx QEMU guide recommends setting the '*icount*' parameter to 7. According to our experiments, this parameter can significantly increase the boot time of a Linux operating system. When emulating the Zynq-7000 board running a Linux/arm 5.4.0 Kernel with the *icount* parameter set to 7, the boot time is around 15 minutes. Applying an *icount* equal to 1, the boot time will result in around 30 seconds.

In our opinion, this platform is the most optimal choice for modeling a high-level IoT architecture, because it enables time modeling, and it is an open-source platform with an active community. In the rest of our work, we consider using this virtual platform.

6.3 The Proposed IoT Architecture Model

Figure 3 depicts the proposed IoT architecture model, which contains a QEMU-emulated ARM Cortex-A9 dual-core 32-bit processor with peripherals such as I2C, SPI, GPIO, and AMBA Interconnect.

The hardware accelerators of the cryptographic primitives identified in Section 5 are modeled in SystemC-TLM (the motif **"DE"** refers to **Design Element**). We can also observe that the hardware accelerators and the ARM CPU are connected to a BUS module, which is also implemented in SystemC-TLM and provided by the Virtual Platform to facilitate the communication between the QEMU instance the SystemC modules. The BUS is also connected to the libSystemCTLM-SoC to allow the communication between the QEMU instance and the SystemC-TLM modules via the blocking *b_transport* communication.

Previously, we mentioned that the emulated CPU architecture has to run a Linux operating system to execute the blockchain-IoT APIs. The objective is to run the identified cryptographic primitives in the hardware accelerator modules rather than the CPU (software execution). When the cryptographic primitive must be run, the CPU has to call the corresponding hardware accelerator module by writing the input data to its *Input register*. When the hardware accelerator finishes its computation, the CPU can read back the results. However, the CPU must be informed about the finishing of the hardware accelerator's computation (when it can read back the results). Each hardware accelerator is connected to the CPU by an **Interrupt ReQuest (IRQ)** to indicate that it finished the computation. The results are saved in the *Output register*. The CPU can read back the results to the API by using these registers. This protocol of the CPU communication with the accelerators is also depicted in Figure 4.

The CPU first writes the data to the hardware accelerator's (IP) *Input register* after it writes to the IP's *Control register* to specify that the *Input register* is complete and the IP can start its computation. The thread of the computation in the SystemC-TLM module waits for the writing notification to the *Control register*. As we can

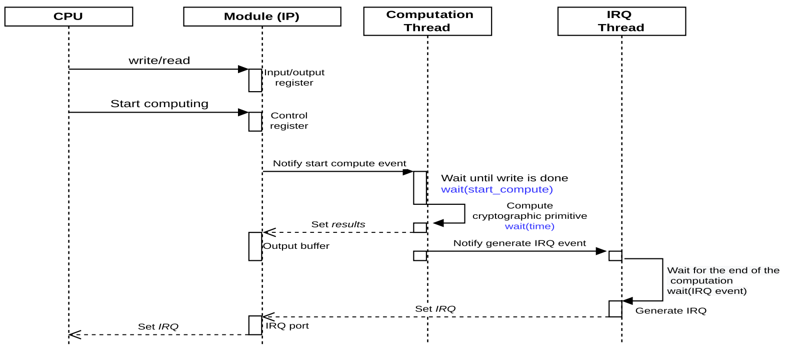

Fig. 4. Basics of modules functioning. *The computation Thread* and the *IRQ Thread* take part in the Module (SystemC-TLM). IP denotes Intellectual Property.

create a timed modeled design, the time required for the computation (see latencies in Table 3 and Table 4) can be simulated by calling the *wait(time)* function. After the computation, the results are written to the output buffer, and an *IRQ* signal can be generated to inform the CPU that it can read back the results.

The generic representation of the ECPM and hash hardware accelerator models are shown in Figure 3. The hash accelerator model has a bidirectional *Output buffer* that allows setting the hash values of the Initial hash value (H_0).

6.4 Specific Device Driver for Hardware Accelerators

Previously, we described that the CPU writes/reads the registers and receives IRQs from the hardware accelerators when one of the cryptographic primitives must be run in the API. However, we did not mention that calling the hardware accelerator IPs from the API cannot be done directly (from the "user space"), because the CPU also executes a Linux operating system. The API must call Linux kernel device drivers to access the hardware IPs. In our architecture model implementation, we developed specific kernel device drivers that can handle the write/read access to/from the hardware devices (e.g., writes message chunks and reads the final hash value of the hash IPs). The kernel device driver also subscribes to the IP's IRQ. When the driver writes the input data to the IP's *Input register* and writes to the *Control register* specifying that the IP can compute, the API will run the read procedure. The driver, at this point, will set the task of the read procedure to sleep mode. This also means that the API must wait until the IRQ does not arrive in the CPU. When the IRQ arrives, the device driver wakes up the task, and it can read back the data from the IP's *Output register* and provide it to the API.

It must be noted that suspending and waking up a task will introduce a certain time penalty (around 100 us), because Linux is not a real-time OS. The negative effect of using device drivers will be presented in the results section.

7 RESULTS

This section will present the results obtained after running the Ethereum and Hyperledger Sawtooth APIs on top of the IoT architecture model presented in the previous section. The results consist of measuring the APIs execution time and the execution time reduction when using the hardware accelerators. Three acceleration logics were applied. First is only the ECPM operation, which is hardware-accelerated. Second, in addition to the ECPM hardware acceleration, the hash functions (SHA, Keccak) are also hardware-accelerated. In the third, the ECPM and the hash primitives are all hardware-accelerated, but the hash hardware accelerators are equipped with a wide buffer to be able to compute 10 internal hash creation in a row. The importance of this hardware design modification is explained later. Table 6 represents the results obtained when running Hyperledger Sawtooth and

Table 6. Overall Total Execution Time of the Proposed Architecture when Running Hyperledger Sawtooth and Ethereum APIs

Acceleration logic	Hyperledger Sawtooth Total Execution Time	Hyperledger Sawtooth Total Time Reduction	Ethereum Total Execution Time	Ethereum Total Time Reduction
No Acceleration	12.4 ms	0%	14.5 ms	0%
ECPM only	5.9 ms	53%	12 ms	18%
ECPM+Keccak	–	–	13.2 ms	9%
ECPM+Keccak wide buffer	–	–	12.2 ms	16%
ECPM+SHA	12 ms	3.5%	–	–
ECPM+SHA wide buffer	9.5 ms	23.5%	–	–

The payload size is 32 bytes. Three different acceleration logics were applied.
"-" denotes that, in the given API, the Keccak or SHA hash functions are not accelerated.

Ethereum APIs applying the different acceleration logics. It must be noted that the transaction payload size is 32 bytes.

After observing the results, we can conclude that in the case of Hyperledger Sawtooth, a significant 53% of total time reduction can be achieved when accelerating only the ECPM operation. However, when accelerating only the ECPM operation, an 18% reduction of the total execution time can be achieved in the execution of Ethereum API. However, it must be noted that accelerating the ECPM operation and the hash computing (SHA or Keccak) in both APIs provides a less significant reduction of the overall execution time than accelerating the ECPM operation alone.

The significant difference between the wide buffer and simple buffer hardware accelerators can also be observed. The performance of the API's acceleration decreases when accelerating the hash-related functions, because Linux is not a real-time OS. The IRQ waiting in the Linux kernel device driver introduces around 100 us of waiting every time the task must wake up or be set in sleep mode. This is due to the Linux tasks' scheduling time. The first results on the basic hash hardware accelerators highlighted that the buffer's size should be increased to avoid as many IRQ waiting as possible. We increased the buffer size to 10, which avoids 9 IRQ waitings. The results clearly show the evolution between the modified and the basic hash designs. However, it is also clear that we cannot significantly increase the number of buffers, because it would affect the overall IoT architecture's size and energy consumption.

It is worth noting that accelerating only the ECPM operation results in a more efficient execution time than accelerating both the ECPM and the hash operations. However, in these results, the payload size was only 32 bytes.

The following table (see Table 7) represents the execution time of Hyperledger Sawtooth API when the payload size is 32 Kbytes (1,024 times more) and when the different acceleration logics are applied. The SHA hash accelerators are equipped with large buffers.

In the related work cited in Reference [17] it was also demonstrated that the time taken by the total execution time could increase significantly when the payload size increases. One-third of the total execution time can be taken by the hash operation when the payload is grater than one megabyte. The results of Table 7 show that even if the size of the payload is significant and the hash accelerators avoid 9 IRQ waiting in a row, the acceleration of ECPM operation alone is still more efficient in terms of execution time than accelerating both ECPM and SHA operations.

8 DISCUSSION

According to the results of the previous sections, we can state that the acceleration of ECPM operation alone seems a more efficient hardware acceleration logic than accelerating the ECPM and the hash operations together. The results also showed that this latter conclusion does not depend on the payload size. However, the hash hardware accelerators were equipped with a wide buffer to store 10 input data for 10 internal hash creations in a row. We assumed that increasing the hash IPs' buffer size will negatively influence the IoT architecture's overall

Table 7. Overall Total Execution Time of the Proposed Architecture when Running Hyperledger Sawtooth API when the Size of the Payload is 32 KBytes

Acceleration logic	**Total Execution Time**	**Total Time Reduction**
No Acceleration	20.89 ms	0%
ECPM only	16.27 ms	22.1%
ECPM+SHA	18.64 ms	10.8%

The results help to compare the hardware acceleration strategies: when the SHA functions are accelerated by hardware and executed by software. The SHA hardware accelerators are equipped with large buffers.

execution time. In future works, we should determine the size of the buffer, which can provide a more efficient execution time than the execution of the ECMP operation alone.

Implementation and System limits: As a potential implementation issue, we have introduced in our work a simulation model for the proposed IoT hardware architecture model. In this model, we have used an "elementary" BUS between the CPU and the potential hardware accelerators. However, in a concrete physical implementation of the model, it would be more sophisticated to apply a more complex physical interconnection BUS such as AXI interconnect or Network-On-Chip. In our simulation model, we have applied 22 nm CMOS technology to adapt all the IPs to the same CMOS technology. Nevertheless, technological properties (e.g., frequency and timings) of the CPU have only been estimated for older CMOS technology. This aspect must be taken into account for future physical implementation.

In this work, we have elaborated on the execution time and acceleration of the blockchain-related operations, however, a more in-depth study is needed on the overall energy consumption of our proposed solution. Obviously, reducing execution time can lead to a more efficient energy consumption. However, a concrete analysis is needed as part of future work. In our work, we have only emulated a CPU that allows frequency scaling; in an extensive future work, we will also have to employ CPUs that allow voltage scaling. The application of frequency and voltage scaling techniques will provide a broader view on the energy consumption that can be achieved.

Most of today's blockchains apply the ECDSA primitive with the secp256k1 curve for signing the transactions. According to this fact, our work is focused on finding hardware accelerators of ECPM operation that are compatible with Weierstrass curves (including the secp256k1 curve). However, there is also a tendency to apply different curves and different signing algorithms in newer versions of blockchain technology. In addition to the new trends in signature algorithms and curves used in blockchain technology, there is also a new tendency to use **Zero-Knowledge-Proof (ZKP)** techniques [45]. Although, if ZKP techniques are also based on elliptic curve operations and point multiplications on the elliptic curves, then further research is required to determine whether our solution can accommodate these specific operations.

Another limitation of our solution is related to real-time functionality. In our work, we run a Linux operating system on our simulation model, and, since a default Linux operating system cannot provide real-time functionality, our approach cannot provide real-time execution of blockchain-related operations on the proposed hardware architecture. However, it is possible to run other real-time operating systems that can provide real-time execution of blockchain transaction generation on the proposed architecture model. Nevertheless, blockchain technology cannot provide real-time execution, so real-time interaction between an IoT device and a blockchain cannot be ensured.

9 CONCLUSION

The main objective of our research is thus completed, since we were able to propose a complete model of IoT hardware architecture adapted to the acceleration of blockchain-related operations. Through the analysis of the most requested functions of the proposed Ethereum and Hyperledger Sawtooth APIs, we were able to identify that the most resource-intensive functions are related to cryptographic operations. We believe that most existing and future IoT applications that allow direct communication with the given blockchain will be constrained because of these same operations. Hardware accelerator designs were listed that can accelerate the mentioned cryptographic operations and were chosen to be implemented in the proposed model. The previously mentioned APIs were executed on the top of the proposed IoT architecture model containing the hardware accelerators and an ARM-based CPU. The results showed that a significant reduction in execution time could be achieved by speeding up the ECMP operation: 53% and 18% for Hyperledger Sawtooth and Ethereum, respectively. The results also highlighted that more studies are needed about the modification of buffer size of the hash design to obtain a significant time reduction without affecting the size and the energy consumption of the design.

REFERENCES

[1] Dr. Andre Luckow, Online, BMW's website. 2001. *How Blockchain Automotive Solutions Can Help Drivers.* Retrieved from https://www.bmw.com/en/innovation/blockchain-automotive.html

[2] Joe Komlodi, Xilinx Wiki, Online. 2017. *Co-simulation.* Retrieved from https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/862421112/Co-simulation

[3] Raynor de Best, Statista (Online). 2021. *Size of the Bitcoin Blockchain from January 2009 to May 20, 2021.* Retrieved from https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/

[4] Asep Muhamad Awaludin, Harashta Tatimma Larasati, and Howon Kim. 2021. High-speed and unified ECC processor for generic Weierstrass curves over GF(p) on FPGA. *Sensors* 21, 4 (2021). DOI: https://doi.org/10.3390/s21041451

[5] Farag Azzedin and Mustafa Ghaleb. 2019. Internet-of-things and information fusion: Trust perspective survey. *Sensors* 19, 8 (2019). DOI: https://doi.org/10.3390/s19081929

[6] Calypso Barnes. 2017. *Verification and Validation of Wireless Sensor Network Protocol Properties through the System's Emulation.* Ph. D. Dissertation. Université Côte d'Azur. Retrieved from https://tel.archives-ouvertes.fr/tel-01618142

[7] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference.* 41–46.

[8] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. arXiv:1407.3561 [cs.NI].

[9] Daniel J. Bernstein and Tanja Lange. 2017. *Safe Curves: Choosing Safe Curves for Elliptic-curve Cryptography.* Retrieved from http://safecurves.cr.yp.to/rigid.html

[10] M. P. Caro, M. S. Ali, M. Vecchio, and R. Giaffreda. 2018. Blockchain-based traceability in agri-food supply chain management: A practical implementation. In *IoT Vertical and Topical Summit on Agriculture—Tuscany (IOT Tuscany'18).* 1–4. DOI: https://doi.org/10.1109/IOT-TUSCANY.2018.8373021

[11] Gang Chen, Guoqiang Bai, and Hongyi Chen. 2007. A high-performance elliptic curve cryptographic processor for general curves over GF(p) based on a systolic arithmetic unit. *IEEE Trans. Circ. Syst. II: Expr. Briefs* 54, 5 (2007), 412–416. DOI: https://doi.org/10.1109/TCSII.2006.889459

[12] Sonia Chhabra, Parveen Mor, Hussain Falih Mahdi, and Tanupriya Choudhury. 2021. *Block Chain and IoT Architecture.* Springer International Publishing, Cham, 15–27. DOI: https://doi.org/10.1007/978-3-030-65691-1_2

[13] Guillaume Delbergue. 2017. *Advances in SystemC/TLM Virtual Platforms: Configuration, Communication and Parallelism.* Ph. D. Dissertation. Université de Bordeaux.

[14] Guillaume Delbergue. 2017. *bcm2837.* Retrieved from https://github.com/hiventive/bcm2837

[15] Guillaume Delbergue, Mark Burton, Frederic Konrad, Bertrand Le Gal, and Christophe Jego. 2016. 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'16).* Retrieved from https://hal.archives-ouvertes.fr/hal-01292317

[16] Luc Gerrits, Edouard Kilimou, Roland Kromes, Lionel Faure, and François Verdier. 2021. A blockchain cloud architecture deployment for an industrial IoT use case. In *IEEE International Conference on Omni-layer Intelligent Systems (COINS'21).* 1–6. DOI: https://doi.org/10.1109/COINS51742.2021.9524264

[17] Luc Gerrits, Roland Kromes, and François Verdier. 2020. A true decentralized implementation based on IoT and blockchain: A vehicle accident use case. In *International Conference on Omni-layer Intelligent Systems (COINS'20).* 1–6. DOI: https://doi.org/10.1109/COINS49042.2020.9191405

[18] Frank Ghenassia. 2005. *Transaction-level Modeling with SystemC.* Vol. 2. Springer, Dordrecht, the Netherlands.

[19] Frank K. Gürkaynak, Kris Gaj, Beat Muheim, Ekawat Homsirikamol, Christoph Keller, Marcin Rogawski, Hubert Kaeslin, and Jens-Peter Kaps. 2012. Lessons learned from designing a 65nm ASIC for evaluating third round SHA-3 candidates. In *3rd SHA-3 Candidate Conference.*

[20] Mohamed Tahar Hammi, Badis Hammi, Patrick Bellot, and Ahmed Serhrouchni. 2018. Bubbles of trust: A decentralized blockchain-based authentication system for IoT. *Comput. Secur.* 78 (2018), 126–142. DOI: https://doi.org/10.1016/j.cose.2018.06.004

[21] Jonette M. Stecklein, Jim Dabney, Brandon Dick, Bill Haskins, Randy Lovell, and Gregory Moroney. 2004. 8.4.2 error cost escalation through the project life cycle. *INCOSE Int. Symp.* 14 (06 2004), 1723–1737. DOI: https://doi.org/10.1002/j.2334-5837.2004.tb00608.x

[22] Khalid Javeed, Xiaojun Wang, and Mike Scott. 2017. High performance hardware support for elliptic curve cryptography over general prime field. *Microprocess. Microsyst.* 51 (2017), 331–342. DOI: https://doi.org/10.1016/j.micpro.2016.12.005

[23] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The elliptic curve digital signature algorithm (ECDSA). *Int. J. Inf. Secur.* 1, 1 (01 Aug. 2001), 36–63. DOI: https://doi.org/10.1007/s102070100002

[24] Ievgen Kabin, Zoya Dyka, Dan Klann, Nele Mentens, Lejla Batina, and Peter Langendoerfer. 2020. Breaking a fully balanced ASIC coprocessor implementing complete addition formulas on Weierstrass elliptic curves. In *23rd Euromicro Conference on Digital System Design (DSD'20)*. 270–276. DOI: https://doi.org/10.1109/DSD51259.2020.00051

[25] Kelly Olson, Mic Bowman, James Mitchell, Shawn Amundson, Dan Middleton, and Cian Montgomery. 2018. *Sawtooth: An Introduction.* Technical Report. The Linux Foundation. Retrieved from https://www.hyperledger.org/wp-content/uploads/2018/01/Hyperledger_Sawtooth_WhitePaper.pdf

[26] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. 1997. HMAC: Keyed-Hashing for Message Authentication. RFC 2104. DOI: https://doi.org/10.17487/RFC2104

[27] Roland Kromes, Luc Gerrits, and François Verdier. 2019. Adaptation of an embedded architecture to run Hyperledger Sawtooth application. In *IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON'19)*. 0409–0415. DOI: https://doi.org/10.1109/IEMCON.2019.8936264

[28] Yong Ki Lee, Herwin Chan, and Ingrid Verbauwhede. 2007. Iteration bound analysis and throughput optimum architecture of SHA-256 (384,512) for hardware implementations. In *Information Security Applications*, Sehun Kim, Moti Yung, and Hyung-Woo Lee (Eds.). Springer Berlin, 102–114.

[29] Vanessa Loury. 2020. Groupe Renault tested a blockchain project to go further in the certification of vehicle compliance. Renault Group, Online: https://media.renaultgroup.com/groupe-renault-tested-a-blockchain-project-to-go-further-in-the-certification-of-vehicle-compliance/

[30] Mohamad Ali Mehrabi, Christophe Doche, and Alireza Jolfaei. 2020. Elliptic curve cryptography point multiplication core for hardware security module. *IEEE Trans. Comput.* 69, 11 (2020), 1707–1718. DOI: https://doi.org/10.1109/TC.2020.3013266

[31] Daniel Minoli and Benedict Occhiogrosso. 2018. Blockchain mechanisms for IoT security. *Internet Things* 1-2 (2018), 1–13. DOI: https://doi.org/10.1016/j.iot.2018.05.002

[32] M. MÃijller, S. R. Garzon, M. Westerkamp, and Z. A. Lux. 2019. HIDALS: A hybrid IoT-based decentralized application for logistics and supply chain management. In *IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON'19)*. 0802–0808. DOI: https://doi.org/10.1109/IEMCON.2019.8936305

[33] O. Novo. 2018. Blockchain meets IoT: An architecture for scalable access management in IoT. *IEEE Internet Things J.* 5, 2 (Apr. 2018), 1184–1195. DOI: https://doi.org/10.1109/JIOT.2018.2812239

[34] Christof Paar and Jan Pelzl. 2009. *Understanding Cryptography: A Textbook for Students and Practitioners.* Springer Science & Business Media.

[35] M. Pincheira and M. Vecchio. 2020. Towards trusted data on decentralized IoT applications: Integrating blockchain in constrained devices. In *IEEE International Conference on Communications Workshops (ICC Workshops'20)*. 1–6. DOI: https://doi.org/10.1109/ICCWorkshops49005.2020.9145328

[36] Miguel Pincheira, Massimo Vecchio, Raffaele Giaffreda, and Salil S. Kanhere. 2021. Cost-effective IoT devices as trustworthy data sources for a blockchain-based water management system in precision agriculture. *Comput. Electron. Agric.* 180 (2021), 105889. DOI: https://doi.org/10.1016/j.compag.2020.105889

[37] Niels Pirotte, Jo Vliegen, Lejla Batina, and Nele Mentens. 2018. Design of a fully balanced ASIC coprocessor implementing complete addition formulas on Weierstrass elliptic curves. In *21st Euromicro Conference on Digital System Design (DSD'18)*. 545–552. DOI: https://doi.org/10.1109/DSD.2018.00095

[38] Thomas Pornin. 2013. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979. DOI: https://doi.org/10.17487/RFC6979

[39] Ana Reyna, Cristian MartÃŋn, Jaime Chen, Enrique Soler, and Manuel Díaz. 2018. On blockchain and its integration with IoT. challenges and opportunities. *Fut. Gen. Comput. Syst.* 88 (2018), 173–190. DOI: https://doi.org/10.1016/j.future.2018.05.046

[40] C. N. Samuel, S. Glock, D. Bercovitz, F. Verdier, and P. Guitton-Ouhamou. 2020. Automotive data certification problem: A view on effective blockchain architectural solutions. In *11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON'20)*. 0167–0173. DOI: https://doi.org/10.1109/IEMCON51383.2020.9284886

[41] Akashi Satoh and Tadanobu Inoue. 2007. ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS. *Integration* 40, 1 (2007), 3–10. DOI: https://doi.org/10.1016/j.vlsi.2005.12.006

[42] Yasir Ali Shah, Khalid Javeed, Shoaib Azmat, and Xiaojun Wang. 2018. A high-speed RSD-based flexible ECC processor for arbitrary curves over general prime field. *Int. J. Circ. Theor. Applic.* 46, 10 (2018), 1858–1878. DOI: https://doi.org/10.1002/cta.2504

[43] Meeta Srivastav, Xu Guo, Sinan Huang, Dinesh Ganta, Michael B. Henry, Leyla Nazhandali, and Patrick Schaumont. 2013. Design and benchmarking of an ASIC with five SHA-3 finalist candidates. *Microprocess. Microsyst.* 37, 2 (2013), 246–257. DOI: https://doi.org/10.1016/j.micpro.2012.09.001

[44] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration* 58 (2017), 74–81. DOI: https://doi.org/10.1016/j.vlsi.2017.02.002

[45] Xiaoqiang Sun, F. Richard Yu, Peng Zhang, Zhiwei Sun, Weixin Xie, and Xiang Peng. 2021. A survey on zero-knowledge proof in blockchain. *IEEE Netw.* 35, 4 (2021), 198–205. DOI: https://doi.org/10.1109/MNET.011.2000473

[46] Nakov Svetlin. 2018. *Practical Cryptography for Developers.* MIT license, Sofia. Retrieved from https://cryptobook.nakov.com/

[47] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First Mond.* 2, 9 (Sep. 1997). DOI: https://doi.org/10.5210/fm.v2i9.548

[48] Josef Weidendorfer. 2017. *KCachegrind Call Graph Viewer.* Retrieved from https://kcachegrind.github.io/html/Documentation.html

[49] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ether. Proj. Yell. Pap.* 151 (2014).

[50] Xilinx. 2018. *Xilinx Quick Emulator User Guide (UG1169).* Technical Report. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1169-xilinx-qemu.pdf

Received 14 December 2022; revised 17 July 2023; accepted 22 September 2023