



PRIVATE PROCESSING OF INTERNET OF THINGS DATA IN
CLOUDS WITH SECURITY BY HARDWARE

Guilherme Araujo Thomaz

Undergraduate Project presented to Electronics
and Computer Engineering Escola Politécnica,
Universidade Federal do Rio de Janeiro, as part
of the requirements necessary to obtain the
degree of Engineer.

Advisor: Miguel Elias Mitre Campista

Rio de Janeiro – RJ, Brazil


February 2024

PROCESSAMENTO PRIVADO DE DADOS DA INTERNET DAS
COISAS EM NUVEM UTILIZANDO SEGURANÇA POR
HARDWARE

Guilherme Araujo Thomaz


PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO
DE ENGENHARIA ELETRÔNICA E DE COMPUTAÇÃO DA ESCOLA POLITÉCNICA
DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS
REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGEN-
HEIRO ELETRÔNICO E DE COMPUTAÇÃO

Autor:



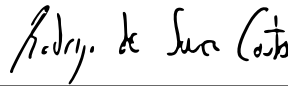
Guilherme Araujo Thomaz

Orientador:



Prof. Miguel Elias Mitre Campista, D. Sc.

Examinador:



Prof. Rodrigo de Souza Couto, D. Sc.

Examinadora:



Profa. Fernanda Duarte Vilela Reis de Oliveira, D. Sc.

Rio de Janeiro

Fevereiro de 2024

Declaração de Autoria e de Direitos

Eu, *Guilherme Araujo Thomaz* CPF 180.489.407-95, autor da monografia *Processamento privado de dados da internet das coisas em nuvem utilizando segurança por hardware*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.
2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e ideias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.
3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.
4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.
5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.
6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.
7. Por ser verdade, firmo a presente declaração.



Guilherme Araujo Thomaz

Araujo Thomaz, Guilherme

Private Processing of Internet of Things Data in Clouds with Security by Hardware/ Guilherme Araujo Thomaz. – Rio de Janeiro: UFRJ/ Escola Politecnica, 2023. XIX, 59 p. 29, 7cm.

Orientador: Miguel Elias Mitre Campista

Projeto de Graduação – UFRJ/ Escola Politécnica / Engenharia Eletrônica e de Computação, 2023

Referências Bibliográficas: p. 51 – 56.

1. Segurança em Redes de Computadores. 2. Computação Confiável. 3. Intel SGX. 4. Internet das Coisas (IoT). I. Elias Mitre Campista, Miguel. II. Universidade Federal do Rio de Janeiro, Escola Politécnica, Engenharia Eletrônica e de Computação. III. Private Processing of Internet of Things Data in Clouds with Security by Hardware

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

Escola Politécnica - Departamento de Eletrônica e de Computação

Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária

Rio de Janeiro - RJ CEP 21949-900

Este exemplar é de propriedade da Universidade Federal do Rio de Janeiro, que poderá incluí-lo em base de dados, armazenar em computador, microfilmear ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es).

À minha família e amigos.

AGRADECIMENTOS

Agradeço à minha família, em especial, meus pais, Gilberto Thomaz e Marcia Araujo, e avós por todo esforço que fizeram para me dar uma educação de qualidade. Além de terem me incentivado a sempre crescer academicamente, foram eles que me formaram como ser humano.

Ao professor Miguel Elias Mitre Campista, por me orientar nesses últimos dois anos e meio, e ao professor Otto Carlos Muniz Bandeira Duarte, que me orientou por um ano. O meu interesse pela pesquisa acadêmica e o meu aprendizado nessa área se deve aos ensinamentos deles. Apesar do professor Otto não estar mais aqui, esse trabalho carrega uma parte da sua paixão pela pesquisa.

Agradeço também à toda equipe de professores e alunos do Grupo de Teleinformática e Automação (GTA), por terem ajudado a tornar o ambiente da faculdade mais aconchegante, instigante e oportuno para meu crescimento pessoal e profissional. Agradeço aos amigos que me acompanharam nessa jornada, pelos momentos de aprendizado pessoal e de descontração.

Aos professores do Departamento de Engenharia Eletrônica e de Computação (DEL) e da Escola Politécnica, por terem me oferecido uma formação de excelência em Engenharia. Certamente, conheci professores que são exemplos de profissionalismo e competência que eu levarei para a minha carreira.

Por fim, agradeço aos brasileiros por terem investido na minha formação. Agradeço também ao CNPq, CAPES, FAPERJ e FAPESP por financiarem esse trabalho, e por todos que lutam e colaboram pela educação pública de qualidade e da pesquisa de alto nível no país. Tenho um compromisso de honrar a educação que eu recebi aos longos desses cinco anos através da minha atuação profissional na sociedade.

ABSTRACT

Internet of Things (IoT) devices collect sensitive data such as energy consumption patterns, biomedical signals, geolocation, and camera images. These data are commonly processed in the cloud due to the limited computing power of the sensors. As a result, clients lose control over how their data is used. The encryption and authentication mechanisms adopted in the industry fail to ensure security when cloud administrators themselves attempt to gain financial advantages from selling data. This work proposes an architecture to protect the processing and storage of IoT data in the cloud, even when an attacker obtains privileged server access. The architecture employs the Software Guard Extensions (SGX), available in specific Intel processors, to process the data within a trusted execution environment, isolated even from the operating system. Performance evaluations reveal that the architecture offers low computational overhead. Additionally, the work implements and documents a Software Development Kit (SDK) for developers to create secure IoT systems using the SGX architecture. An energy data management system is implemented using the proposed SDK's functionalities. The SDK enables a developer who is not a security expert to develop systems using databases, communication protocols, graphical user interfaces, and processing tasks typical in IoT scenarios.

Keywords: Networks Security, Trusted Computing, Intel SGX, Internet of Things.

RESUMO

Os dispositivos da Internet das Coisas (*Internet of Things* – IoT) coletam dados sensíveis como padrões de consumo de energia elétrica, sinais biomédicos, geolocalização e imagens de câmeras. Esses dados são comumente processados em nuvem, devido a restrição do poder de processamento dos sensores. Com isso, os clientes perdem o controle de como seus dados são utilizados. Os mecanismos de criptografia e autenticação empregados no mercado falham em garantir a segurança quando os próprios administradores em nuvem tentam obter vantagens financeiras com a venda de dados. Este trabalho propõe uma arquitetura para proteger o processamento e o armazenamento de dados de IoT em nuvem até mesmo no caso do atacante obter acesso privilegiado aos servidores. A arquitetura utiliza o *Software Guard Extensions* (SGX), disponível em alguns processadores da Intel, para processar os dados em um ambiente de execução confiável, isolado até mesmo do sistema operacional. As avaliações de desempenho revelam que a arquitetura oferece baixa sobrecarga computacional. Além disso, o trabalho também implementa e documenta um Kit de Desenvolvimento de *Software* (*Software Development Kit* – SDK) para que desenvolvedores criem sistemas IoT seguros utilizando a arquitetura. Um sistema de gerenciamento de dados de consumo de energia foi implementado utilizando as funcionalidades da SDK proposta. A SDK permite que um desenvolvedor que não seja especialista em segurança desenvolva sistemas utilizando bancos de dados, protocolos de comunicação, interfaces gráficas e tarefas de processamento típicas em cenários IoT.

Palavras-Chave: Segurança em Redes de Computadores, Computação Confiável, Intel SGX, Internet das Coisas (IoT).

SIGLAS

ACL - Access Control Lists

AES - Advanced Encryption Standard

AEX - Asynchronous Enclave Exit

AP - Access Point

CA - Certificate Authority

CACIC - Controle de Acesso Confiável à Dados em Nuvem da Internet das Coisas com Enclaves

CK - Communication Key

CoAP - Constrained Application Protocol

CPU - Central Processing Unit

CSPRNG - Cryptographically Strong Pseudo-Random Number Generator

DCAP - Data Center Attestation Primitives

DDoS - Distributed Denial of Service

DHKE - Diffie-Hellman Key Exchange

DRAM - Dynamic Random Access Memory

ECDHKE - Elliptic Curve Diffie-Hellman Key Exchange

ECDSA - Elliptic Curve Digital Signature Algorithm

EDL - Enclave Definition Language

EPC - Enclave Page Cache

EPCM - Enclave Page Cache Map

EPID - Enhanced Privacy ID

GCM - Galois/Counter Mode

GTA - Grupo de Teleinformatica e Automação

HMAC - Hash Message Authentication Code

HTTPS - Hypertext Transfer Protocol Secure

IAS - Intel Attestation Server

IDS - Intrusion Detection Systems

IPC - Inter-Process Communication

IoT - Internet of Things

IV - Initialization Vector

LAN - Local Area Network

MAC - Message Authentication Code

MEE - Memory Encryption Engine

MIT - Massachusetts Institute of Technology

MITM - Man-In-The-Middle

RAN - Radio Access Networks

OS - Operating System

OTP - One-Time Programmable

PKI - Public Key Infrastructure

PRM - Processor Reserved Memory

RSA - Rivest-Shamir-Adelman

RAM - Random Access Memory

SBRC - Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

SDK - Software Development Kit

SECS - SGX Enclave Control Structure

SGX - Software Guard Extensions

SHA - Secure Hashing Algorithm

SK - Storage Key

SoC - System-on-Chip

TCS - Thread Control Structure

TA - Trusted Authority

TEE - Trusted Execution Environment

TLS - Transport Layer Security

WLAN - Wireless Local Area Network

Contents

1	Introduction	1
1.1	Related Work	3
1.2	Publications	5
1.3	Text Organization	6
2	Technical Background	7
2.1	IoT Networks	7
2.2	Cryptography Fundamentals	8
2.2.1	Confidentiality	8
2.2.2	Integrity	9
2.2.3	Freshness	10
2.2.4	Authentication	11
2.2.5	Key Agreement	12
2.3	Security in IoT Clouds	13
2.4	Trusted Execution Environments	14
2.4.1	Memory Enclaves	15
2.4.2	Encryption and Attestation	17
2.4.3	Application Development	19
2.4.4	Performance Limitations	20
2.4.5	Security Limitations	20
3	CACIC Architecture	22
3.1	Trusted Protocols	22
3.1.1	Registration	22
3.1.2	Publication	23

3.1.3	Query	26
3.1.4	Revocation	28
3.2	Security Evaluation	28
3.2.1	File Manipulation	29
3.2.2	Packages Manipulation	30
3.2.3	Software Manipulation	30
3.3	Performance Evaluation	31
3.3.1	Scalability	32
3.3.2	Latency	35
3.3.3	Microbenchmarking	36
3.3.4	Resources Usage	37
3.3.5	Use Case: Aggregation	39
3.3.6	Validity Discussion	40
4	Software Development Kit and Demonstration	42
4.1	CACIC-DevKit Functionalities	42
4.2	Use Case and Demonstration	44
5	Conclusions	49
	Bibliography	51

List of Figures

2.1	Typical IoT scenario where devices and users send data to and collect data from the cloud. The cloud server must process and store an immense amount of data necessary to create a connected ecosystem for homes, cities, and factories.	8
2.2	The untrusted channel can be an Internet link spied by an attacker. Blue indicates a symmetric key, green is a public key, and red is a private key. The attacker does not understand the encrypted message, and ideally, the best way to retrieve the message without the key is by brute force.	9
2.3	The circle with C is a concatenator, while the circle with S is a splitter. The attacker is not modifying the message, but if it does, the MAC tag checker detects the tampered data. The message source can generate an already encrypted message to protect confidentiality.	10
2.4	An attacker sending messages impersonating the genuine message source will not be able to generate the signature, concatenated to the message (circle with C) because it does not have the source private key (in red).	11
2.5	Both sides compute k using the received public parameters ($g^A \bmod p$ and $g^B \bmod p$) and its corresponding private random number (A or B), without needing to know the private random number at the other side.	12
2.6	Memory separation with SGX. The lines represent threads. After creating an enclave, Application 3 (App 3) can call a function that executes inside the isolated memory (PRM). The red arrow represents the function call, while the blue arrow illustrates the return.	15

3.1	Components and interfaces in blue compose CACIC core, i.e., they do not depend on the use case. The architecture is agnostic to the processing task, the database, and the data sources.	23
3.2	Data in transit is encrypted with the Communication Key (CK), and stored data is encrypted with the Storage Key (SK). Only the enclave decrypts and processes the data in the cloud. The blue color highlights the security mechanisms introduced by the architecture.	24
3.3	The enclave reduces the maximum publication rate by 12%, but the system still processes thousands of publications per second.	33
3.4	The latency quickly increases when the system achieves its maximum publication rate since new messages are inserted into a queue.	33
3.5	This result is analogous to Figure 3.3. The reduction in message processing rate due to the enclave is more significant in that case, since queries are faster.	34
3.6	This result is analogous to Figure 3.4. The system processes more queries than publications per second.	35
3.7	The enclave adds an imperceptible latency overhead for the clients. The enclave does not perform any computation-intensive processing and the overhead is mainly due to the ECALL.	35
3.8	Percentage of CPU time for the server process, considering the sum of all CPUs. The resource usage is dominated by the HTTPS protocol implementation, as seen in gray.	38
3.9	The enclave overhead increases as more data is transferred to it. However, the time for reading the samples from the database is much higher than this enclave overhead.	39
4.1	Execution flow for a M[publication]. The functions in blue are implemented in the core, while functions in black must be programmed by the developer. The solid arrows represent a function call or return, while the dashed arrows indicate a network message. The lock represents enclave functions.	45

4.2	(a) Start menu. (b) Menu for configuring the IDs permitted to access each type. (c) Publication request menu. (d) Window with the result of a query.	46
4.3	The client LAN (left side) communicates with the CACIC server (right side) through the Internet, using CACIC messages, in green. The green key represents the client CK, shared with the server. Both the access point and the user equipment use it to encrypt the messages.	46
4.4	The publisher key (CK), the storage key (SK), and the decrypted publication request are only accessible inside the enclave. The content of the decrypted publication message is an SQLite query filtering the previous data that must be aggregated inside the enclave. The result is automatically encrypted and inserted into the database, without developer intervention.	48

List of Tables

2.1	Security requirements for IoT clouds. Issues can emerge if the solution is not effective against the threat model. An attacker who can read any file can read cryptographic keys from the disk and harm the system’s confidentiality, for example.	13
3.1	Publication and query latency microbenchmark results. The publication time is dominated by the disk writing and the query time is dominated by the enclave.	37
4.1	Functions in black must be programmed according to the use case, while functions in blue are part of the core of CACIC-DevKit. The documentation details each function’s arguments, as well as in which file it must be implemented.	44

Chapter 1

Introduction

The Internet of Things (IoT) allows devices to collect, store, process, and transmit data. This paradigm gains popularity as more tasks are automated by devices, including smart homes, factories, vehicles, and farms [1, 2, 3]. IoT devices, however, have restricted battery duration, storage, processing capacity, and bandwidth. Therefore, data collected by IoT sensors are sent to remote resource-rich infrastructures, such as the cloud.

Cloud computing offers low cost since multiple virtual machines can share the same physical substrate, i.e., a single machine. These virtual infrastructures are allocated on the fly according to the resource demand [4]. However, data collected by the devices reveal sensitive information such as biomedical signals, vehicle images, energy consumption patterns, and data gathered from critical industrial processes. Hence, a well-succeeded attack to the cloud impacts the security of millions of devices, justifying the importance of proposals enhancing the security of cloud-based IoT systems [5, 6].

Due to cryptography, the data sent to the cloud is protected while in transit and stored. At the same time, authentication mechanisms ensure clients' interaction with genuine servers before sending or requesting data. These mechanisms are already implemented in commercial systems, but do not guarantee data security while it is processed [7]. Cryptography and authentication assume that the operating system, the hypervisor, and the cloud administrator are trustworthy, i.e., the system denies access to plain data being processed in memory. However, this premise may not be realistic, as will be discussed.

Client data are valuable because they can be used to train machine learning models, leading to, for example, customized advertisements based on individuals' consumption profile [8, 9]. Hence, it is reasonable to assume that a cloud insider could maliciously access the client data during processing to obtain financial advantage [8, 9]. Several countries made legal efforts to enforce companies to follow data usage policies in a manner that keeps client's privacy. Despite this regulation, mechanisms to prevent access to data being processed by highly privileged software are not widespread. Trusted computing is a set of technologies that can fill this gap [7].

Unlike traditional security mechanisms, trusted computing relies on hardware resources. One of the most important trusted computing mechanisms is the Trusted Execution Environment (TEE). TEE protects data while processed, even if any other software component is controlled by an attacker. The Intel Software Guard Extensions (SGX) [10] is the most popular TEE implementation. Intel CPUs equipped with SGX have special instructions to create isolated memory regions called enclaves.

This work focus on TEE utilization to protect IoT data in clouds. Therefore, this Final Undergraduate Project proposes, implements, and evaluates an architecture that leverages enclaves to ensure that cloud servers follow data access policies defined by clients. The architecture is named CACIC, which is an acronym in Portuguese for Trusted Access Control for Internet of Things Data in Clouds using Enclaves [11, 12, 13].

Enclaves are state-of-the-art solutions to protect data collection and aggregation, manage cryptographic keys, and protect databases. Recent works employ TEE in IoT contexts [14, 15, 16, 17, 18]. However, these works do not allow clients to customize data access policies. In addition to data access customization, works typically propose case-specific solution for IoT, since they rely on particular data sources, databases, and processing tasks. This work fills these gaps by proposing a generic architecture for IoT data access control in clouds.

This work considers a scenario in which data is collected in a sensor network while the processing and storage are delegated to cloud servers. The proposal focuses on ensuring the confidentiality and integrity of the data and access policies during the transmission, storage, and processing, even if the attacker controls the whole

server software. Attacks that compromise client devices, attacks towards the server availability, and attacks that explore Intel CPU’s vulnerabilities are out of the scope. Therefore, the main goal is to develop an architecture to protect IoT data in clouds, assuming a threat model with privileged access to the server. The specific goals are:

1. formulate the threat model and the security, performance, and generalization requisites;
2. design an architecture that uses enclaves to follow the specified requisites, facing the proposed threat model and solving previous works limitations;
3. build the architecture;
4. evaluate security and performance;
5. document a Software Development Kit (SDK) for developers to build their own IoT systems with CACIC architecture, and;
6. demonstrate a system based on an energy consumption data management use case, implemented with the proposed SDK.

1.1 Related Work

This section discusses related work using trusted computing to secure IoT networks. Xiao *et al.* combine blockchain with TEE for access control [19]. A highlight of the work is the execution of selling contracts out of the chain, in an SGX enclave. The work focuses on data commercialization. Yang *et al.* use TEE to protect the publication of industrial sensor data, which may be compromised by physical attacks [14]. The proposal uses a blockchain to store data in a distributed manner and to train federated learning models using smart contracts. The proposal does not deal with cloud data security and does not ensure confidentiality since the distributed ledger is public.

Li *et al.* execute aggregation tasks, dynamic pricing, and load forecasting in enclaves [20]. This scenario obligates the users to send their data to remote servers because i) the sensors have low computational power, and ii) the tasks depend on the data from multiple clients. The architecture uses SGX on the server side and

on the client access point. The proposal focuses on securely sending data to energy companies and does not implement access control for authenticated users to query data. Silva *et al.* compares TEE with homomorphic encryption in terms of the time taken to aggregate measurements [21]. The time taken for aggregating with TEE is ten thousand times lower, confirming that trusted computing ensures security without a significant performance overhead.

Priebe *et al.* developed a database engine running within SGX enclaves [17]. Although the system is promising for IoT data in clouds, the proposal focuses solely on data storage. Valadares *et al.* expand the FIWARE platform, a development platform for smart sensor applications, to share sensitive data with authenticated users [15]. In their proposal, cloud enclaves store and distribute keys for encrypting produced data and decrypting consumed data. The architecture does not process data in the cloud and requires the consumer to have an SGX computer to process the data locally. The cloud enclave only implements a trusted publish-subscribe system.

Ayoade *et al.* propose trusted computing to process IoT data from different manufacturers in a shared cloud middleware [22]. The authors isolate the processing in enclaves and provide the data only to the device's company without customized access policies. The system uses SGX at the access point to protect against client-side attacks. However, this approach is unrealistic since SGX is commonly found only in high-performance processors¹. Additionally, the system initializes and attests the enclave for each publication, resulting in a significant performance overhead.

Anciaux *et al.* describe an architecture for personal data management with generic processing tasks in cloud [18]. The authors proposed the use of ARM TrustZone TEE on mobile devices and SGX on the server, but they did not provide an implementation. In another work, Carpentier *et al.* implement the architecture proposed by Anciaux *et al.* based on an application to reward employees using bicycles [23]. The system core implements access control policies using SGX and can be expanded with additional enclaves for more complex processing. The authors assume the use of smartphones with high computational power. Experiments are still needed to confirm whether the architecture meets the high throughput and low

¹SGX is found only in the 3rd Generation Xeon Scalable product line.

latency requirements necessary for some applications.

Unlike previous works, this paper proposes an architecture that enforces the server to adhere to client data access policies, even in the presence of attackers with privileged access. The performance evaluation demonstrates that the architecture meets the low-latency and high-throughput requirements of IoT networks. Another contribution of this work is the implementation of a development tool to build IoT systems based on the CACIC architecture. The paper demonstrates that the tool's APIs enable non-security developers to integrate enclave technology with commercial database solutions, user interfaces, and communication protocols.

1.2 Publications

The final project includes three publications carried out by the student during their undergraduate course, listed as follows:

- THOMAZ, G. A., GUERRA, M. B., SAMMARCO, M., et al. “CACIC: Controle de Acesso Confiável Usando Enclaves a Dados em Nuvem da Internet das Coisas”. In: Anais do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2022), pp. 573–586, Fortaleza, CE, Brasil.
- THOMAZ, G. A., GUERRA, M. B., SAMMARCO, M., et al. “Tamper-proof access control for IoT clouds using enclaves”, *Ad Hoc Networks*, v. 147, pp. 103191, 2023. ISSN: 1570-8705.
- THOMAZ, G., GUERRA, M., SAMMARCO, M., et al. “CACIC-DevKit: Construção de Sistemas IoT com Políticas de Acesso Customizáveis e Segurança por Hardware”. In: Anais Estendidos do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2023), pp. 1–8, Porto Alegre, RS, Brasil, 2023.

The student has participated in other research projects within the Grupo de Teleinformática e Automação (GTA) that have resulted in national and international publications in the areas of blockchain, federated learning, and Open Radio Access Networks (RAN):

- THOMAZ, G. A., CAMILO, G. F., DE SOUZA, L. A. C. et al., "Uma análise comparativa da arquitetura e desempenho de plataformas de blockchain permissionadas para contratos inteligentes". In: Anais do IV Workshop em Blockchain: Teoria, Tecnologias e Aplicações (WBlockchain 2021), pp. 114–127, 2021, Uberlândia, MG, Brasil.
- THOMAZ, G. A., CAMILO, G. F., DE SOUZA, L. A. C., et al., "Architecture and Performance Comparison of Permissioned Blockchain Platforms for Smart Contracts". In: 2021 IEEE Global Communications Conference (GLOBECOM), pp. 1–6, 2021, Madrid, Spain.
- REBELLO, G. A. F., CAMILO, G. F., GUIMARAES, L. C., et al., "A Security and Performance Analysis of Proof-Based Consensus Protocols", *Annals of Telecommunications*, pp. 1–21, 2021.
- CAMILO, G. F., REBELLO, G. A. F., DE SOUZA, L. A. C., et al., "Redes de Canais de Pagamento: Provendo Escalabilidade para Pagamentos em Criptomoedas". In: Minicursos do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2022), Fortaleza, CE, Brasil, 2022.
- MATTOS, D. M. F., DE MEDEIROS, D. S. V., DE SOUZA COUTO, R., et al., "Ameaças e Vulnerabilidades em Open RAN: Desafios e Soluções", In Minicursos do XXIII Simpósio Brasileira de Segurança da Informação e Sistemas Computacionais (SBSeg 2023), Juiz de Fora, MG, Brasil, 2023.

1.3 Text Organization

This work is divided into five chapters. Chapter 1 presents the state-of-the-art, discussing its current limitations. Chapter 2 overviews the background needed to fully understand the proposal, such as IoT architectures, security threats, and TEE. Chapter 3 proposes the architecture and evaluates its performance and security. Chapter 4 presents the SDK functionalities and demonstrates its functionalities with a practical use case based on an IoT system for energy consumption measurement. Finally, Chapter 5 concludes this work and draws future directions.

Chapter 2

Technical Background

This chapter presents the theory on IoT networks, security and trusted computing that is used for the rest of this work.

2.1 IoT Networks

Internet of Things (IoT) is a paradigm where watches, traffic lights, door locks, industrial controllers, and other devices are equipped with data collection, processing, storage, and transmission capabilities [1]. However, these devices usually present intermittent connectivity, are powered by batteries, offer a small-capacity storage medium, and are equipped with low-cost microcontrollers running very lightweight software. Due to these limitations, they send the collected data to be processed in remote servers. First, the devices send the data, usually using some wireless link, to an Access Point (AP), which acts as the Local Area Network (LAN) gateway. In some cases, the AP performs some computation, such as cryptographic operations or the first layers of a neural network [24]. Finally, the data is transmitted through the Internet to remote servers, usually instantiated as virtual machines in the cloud. The servers can store a massive amount of data and perform intensive processing tasks, like complex machine learning model training over a big amount of data. Companies and users' devices may query the stored data to automate processes and make decisions. Figure 2.1 illustrates a typical IoT infrastructure. The next section discuss cryptography tools that are typically used to provide security in IoT networks.

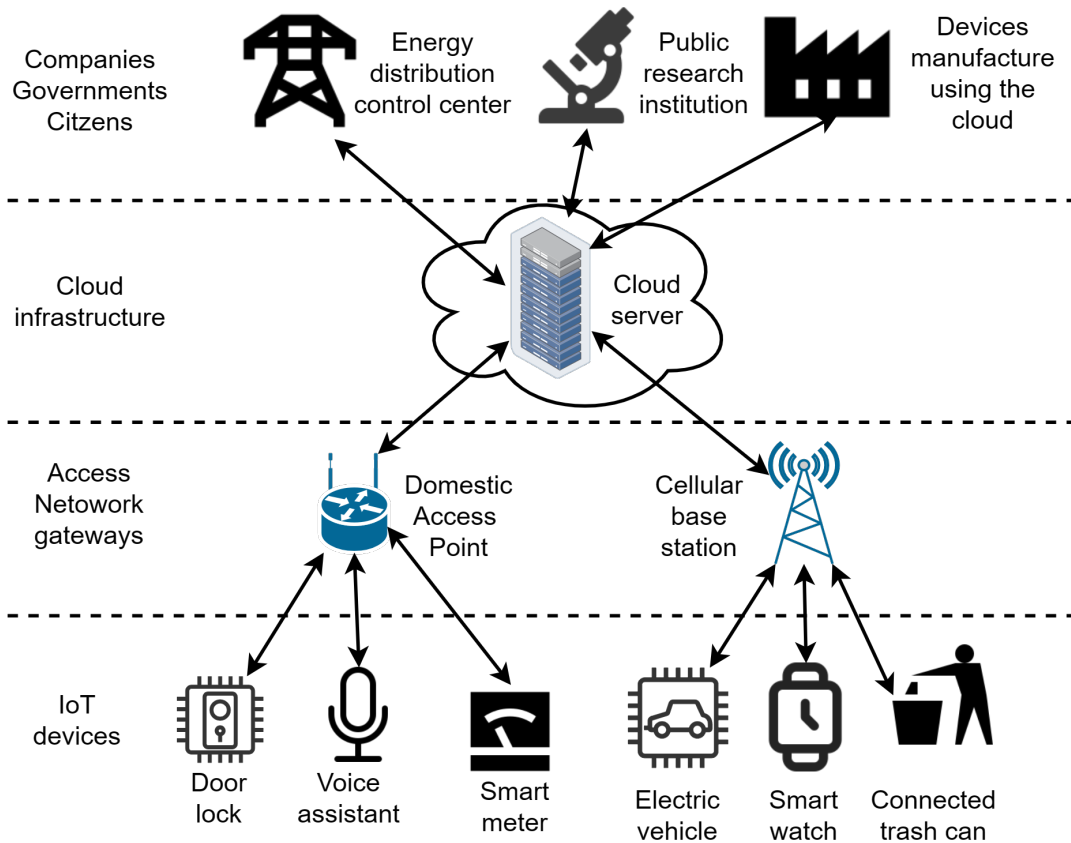


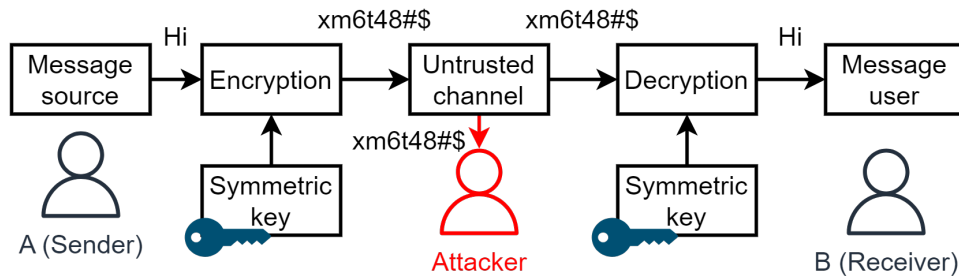
Figure 2.1: Typical IoT scenario where devices and users send data to and collect data from the cloud. The cloud server must process and store an immense amount of data necessary to create a connected ecosystem for homes, cities, and factories.

2.2 Cryptography Fundamentals

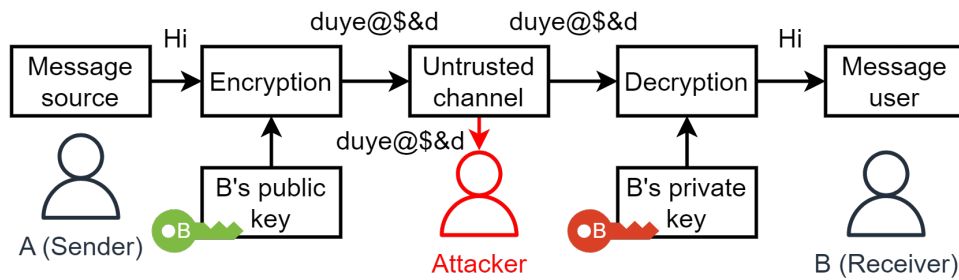
This section covers the cryptographic operations that will be used throughout this work.

2.2.1 Confidentiality

Data confidentiality is a property that ensures that the data going from A to B cannot be read by an adversary. This is achieved using an encryption algorithm to hide the message using an encryption key, and a decryption algorithm to recover the original message from the encrypted one using a decryption key, as illustrated in Figure 2.2. In symmetric-key block ciphers, such as the Advanced Encryption Standard (AES), the encryption and decryption keys are the same. In asymmetric-key block ciphers, such as the Rivest-Shamir-Adelman (RSA), there are two different



(a) Symmetric Cryptography



(b) Asymmetric Cryptography

Figure 2.2: The untrusted channel can be an Internet link spied by an attacker. Blue indicates a symmetric key, green is a public key, and red is a private key. The attacker does not understand the encrypted message, and ideally, the best way to retrieve the message without the key is by brute force.

keys: a public key and a private key. If a user A wants to send a message to a user B, he must encrypt it with B's public key. Only B will be able to decrypt the message using its private key. Communication from B to A must use A's key pair. Key generation algorithms use a Cryptographically Strong Pseudo-Random Number Generator (CSPRNG) [10].

2.2.2 Integrity

Data integrity is a property that ensures that if the data going from A to B is modified by an adversary, B will be able to detect it. This is achieved using secure hashing functions, such as the Secure Hashing Algorithm (SHA), which map an input of any size to a fixed-size output. These functions must be quick to compute, nearly impossible to reverse, and very difficult to generate repeated results for different inputs. It is common to say that the message was *hashed* and that the result is *the hash of the input*. An application of SHA is the Hash Message Authentication

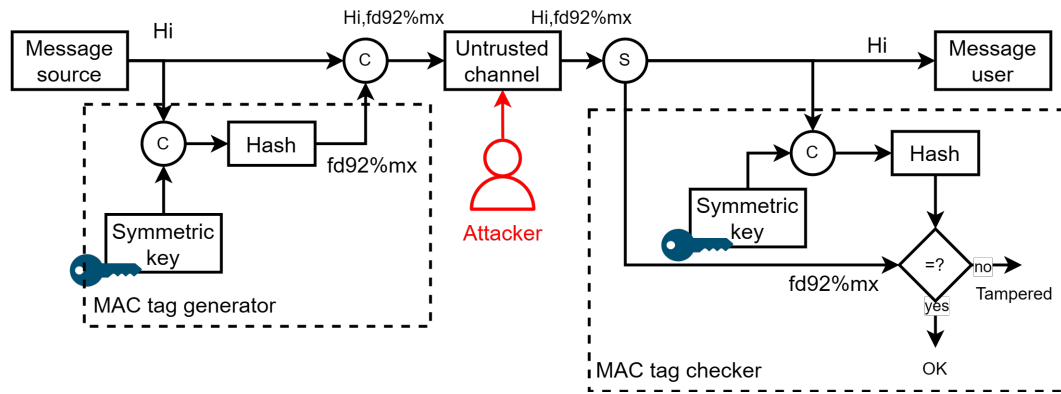


Figure 2.3: The circle with C is a concatenator, while the circle with S is a splitter. The attacker is not modifying the message, but if it does, the MAC tag checker detects the tampered data. The message source can generate an already encrypted message to protect confidentiality.

Code (HMAC), a scheme that produces an HMAC tag by *hashing* the message together with a secret symmetric key. Figure 2.3 shows that the receiver recomputes the HMAC and verifies if the result matches the received tag. The message can be encrypted with AES, before generating the HMAC, using the same secret key to achieve confidentiality and integrity. An alternative is using AES operating in Galois/Counter Mode (AES-GCM), which embeds the MAC tag generation and verification inside the AES block cipher [10].

2.2.3 Freshness

Data freshness is a property that ensures that every received data whose integrity was validated is not a copy of an old one being replayed by an attacker. This can be ensured by concatenating a single-use random number, named nonce, in each message before computing its HMAC. The destination stores the nonces, verifies if the received nonce is repeated, and discards the message if so. This prevents an attacker without the key from validating an old genuine message as a new one with the destination. AES-GCM encryption uses a number called Initialization Vector (IV), which can be retrieved in the decryption. The IV can be used as a nonce since the same data being encrypted with two different IVs leads to two completely different results [10].

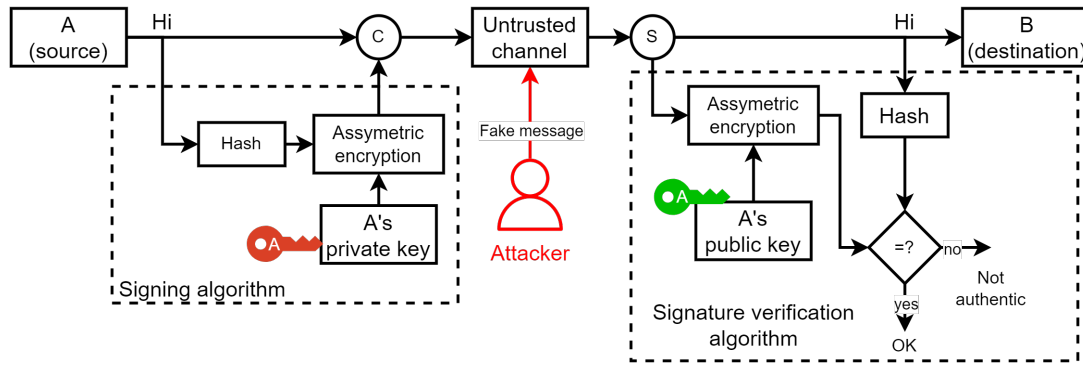


Figure 2.4: An attacker sending messages impersonating the genuine message source will not be able to generate the signature, concatenated to the message (circle with C) because it does not have the source private key (in red).

2.2.4 Authentication

Authentication is a mechanism to confirm that a user is who he claims to be, and that a message was generated by him. This can be achieved using a Public Key Infrastructure (PKI), in which each user is identified to others by its public key, while its private key remains confidential. Since anyone who wants to send a message to B can encrypt it with B's public key, a sender (A) must prove to B that the message was generated by himself and not by an attacker impersonating A. To authenticate the message, A must prove to B that it possesses A's private key. A signature is a mechanism to achieve message integrity and authenticity in PKIs, as illustrated in Figure 2.4. A encrypts the hash of the message with its private key, generating a signature. Once B receives the signature, it decrypts it with A's public key and verifies if the result corresponds to the received message's hash. Although RSA can be used in signing schemes, a more efficient signing algorithm is the Elliptic Curve Digital Signature Algorithm (ECDSA). In PKIs, a Certificate Authority (CA) is a trusted entity responsible for generating a certificate for each party's public key. Some fields contained in a certificate following X.509 format are the proprietary's identity, the proprietary's public key, the issuer CA public key, and a signature generated by the issuer CA. The goal of a certificate is to attach each party to an identity, such as a name, and to prove to other parties that this identity owns a public key [10].

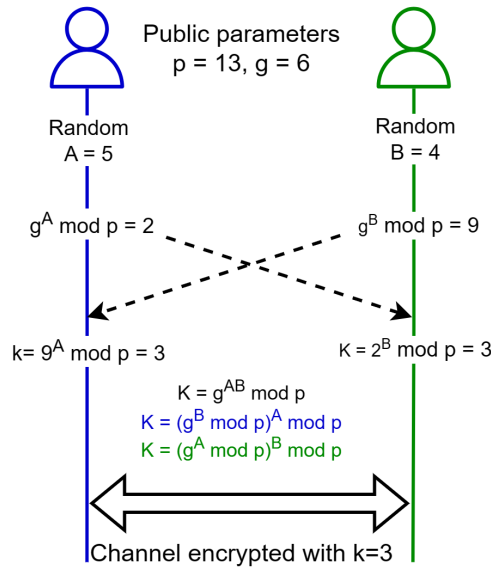


Figure 2.5: Both sides compute k using the received public parameters ($g^A \bmod p$ and $g^B \bmod p$) and its corresponding private random number (A or B), without needing to know the private random number at the other side.

2.2.5 Key Agreement

Whereas in asymmetric-key block ciphers the public key can be broadcasted in untrusted channels, in symmetric-key block ciphers, the key must be sent in a secure channel [10]. Diffie-Hellman Key Exchange (DHKE) is a symmetric key agreement protocol for two parties sharing a key, even in the presence of a man-in-the-middle (MITM) attacker reading and fabricating messages. Figure 2.5 shows that both ends privately generate random parameters A and B and end up with the same key $K = g^{AB} \bmod p$ in such a way that an attacker reading messages is not able to compute A or B nor generate K [10]. The Transport Layer Security (TLS) is a network protocol that uses DHKE to share a temporary secret session key between the client and the server. The server also authenticates itself to the client by sending a certificate and signing the last message in DHKE with its private key. The Hypertext Transfer Protocol Secure (HTTPS) uses TLS for authenticated key agreement, which is necessary to build an encrypted channel.

The cryptography tools presented in this section are used to meet security requirements in IoT networks, discussed in the next section.

Table 2.1: Security requirements for IoT clouds. Issues can emerge if the solution is not effective against the threat model. An attacker who can read any file can read cryptographic keys from the disk and harm the system’s confidentiality, for example.

Security requirement	Meaning	Issues	Possible solution
Confidentiality	Sensitive data cannot be revealed to third parties	Client data can be stolen and sold	Cryptography
Integrity	Data cannot be changed by third parties	Attacker can unlock house doors or control industrial actuators	Message Authentication Code (MAC)
Authentication	Clients must identify themselves to send and receive data	Attacker impersonates a genuine client to compromise confidentiality and integrity	Digital certificates Digital signatures Passwords/Tokens
Access Control	Clients must choose who has the access permission to their data and what can be done with it	The system allows the attacker to read or modify the data without the client’s consent, compromising confidentiality and integrity	Role-based Access Control Sticky Policies
Freshness	The same data cannot be sent more than once	Attacker can control actuators by replaying old valid messages	Cryptographic nonce

2.3 Security in IoT Clouds

This section has two goals: i) to define the requirements that the system must follow to ensure IoT data security in clouds, and ii) to define the set of actions the attacker can perform, known as the threat model. Table 2.1 presents the five security requirements that must be accomplished to ensure IoT data security in cloud servers. The table identifies some possible threats that can happen if the system does not meet the requirements. It also presents some typical security solutions whose efficacy depends on the threat model.

We consider a threat model where the attacker can control every application, operating system, and hypervisor in the cloud. Therefore, the attacker can perform the following actions:

1. read or write any file;
2. intercept, read, retransmit, or fabricate any network package;
3. execute or modify any application.

Nevertheless, the attacker cannot:

1. access the place where data is generated, like residences or factories where sensors, actuators, and local area network gateways are installed;
2. tamper with cryptographic primitives;
3. perform physical attacks on the CPU package, nor side-channel attacks.

This threat model defines a situation in which an attacker is someone who successfully performed privilege escalation to dominate the entire software stack or a malicious employee inside the cloud infrastructure with privileged access. These are realistic scenarios since i) an attacker has a financial incentive to steal client data, and ii) it is still a challenge to eliminate attacks from cloud managers [25]. The traditional solutions for protecting IoT data in clouds assume a weaker threat model, in which the super-user and the operating system are trustworthy [26, 27]. This is because a privileged attacker can read cryptographic keys, which are the basis of the security solutions presented in Table 2.1.

We define that *an IoT system is secure if, and only if, it meets the determined requirements under the defined threat model*. Since the solutions presented in Table 2.1 alone are not effective against our threat model, we combine them with trusted computing using enclaves, as described in the next section. Ensuring availability is outside the scope of this work, since it is achieved with orthogonal solutions, such as machine reapplication and Intrusion Detection Systems (IDS) [28].

2.4 Trusted Execution Environments

Trusted Execution Environments (TEE) use hardware resources to execute code sections securely without depending on the security of any other software component. The most popular TEE implementation in cloud computing is the Software Guard Extensions (SGX), an extension of the x86 instruction set available in most recent Intel Xeon processors. These new instructions can create and destroy trusted execution environments, called enclaves [10]. ARM also has its own TEE implementation, known as TrustZone, which is commonly used in mobile System-on-Chip (SoC) integrated circuits and cheap microcontrollers [29]. The rest of this section focuses on Intel SGX implementation because it is the technology used throughout

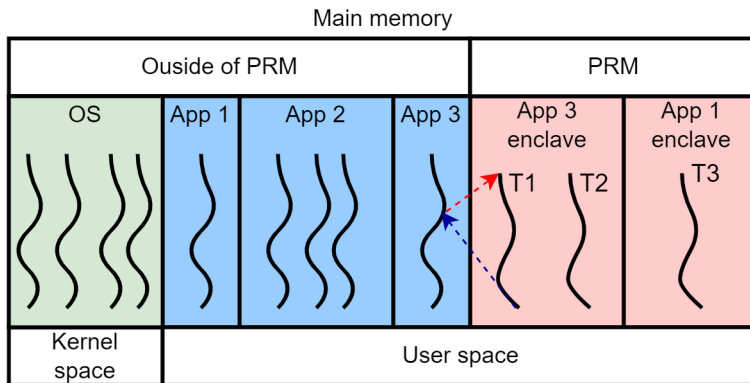


Figure 2.6: Memory separation with SGX. The lines represent threads. After creating an enclave, Application 3 (App 3) can call a function that executes inside the isolated memory (PRM). The red arrow represents the function call, while the blue arrow illustrates the return.

the rest of this work.

2.4.1 Memory Enclaves

Enclaves are isolated regions in the DRAM for protecting data confidentiality, as well as data and code integrity. In SGX-compatible CPUs, there is a subset of the DRAM address space where access is controlled by logic circuits called Processor Reserved Memory (PRM). Pages containing code and data from enclaves are allocated in a subset of the PRM, called the Enclave Page Cache (EPC) [10]. Figure 2.6 illustrates how the memory is organized in a computer with SGX.

When an application wants to process sensitive data, it asks the OS to create an enclave. Since the PRM access control hardware prohibits the OS from directly manipulating the PRM, the OS must use SGX instructions to create the enclave. The first instruction for creating the enclave, `ECREATE`, creates an SGX Enclave Control Structure (SECS), which can be considered the enclave identity. The SECS specifies various enclave metadata, such as the location of the enclave in memory. Then, the OS transfers data and code pages from outside the PRM to the EPC using the `EADD/EEXTEND` instructions. Each enclave page has metadata stored in a table called Enclave Page Cache Map (EPCM), such as its virtual address, its read/write/execute permissions, and a pointer to the SECS of the owner enclave. Last, the `EINIT` instruction puts the enclave in initialized mode, disallowing any posterior

instruction to modify the enclave code and metadata. There are instructions for destroying the enclave and freeing EPC pages, which is not discussed [10].

Now that the enclave is created, any application thread can call functions that reside inside the enclave. Thus, the application must execute `EENTER` instruction, which puts the processor in enclave mode and forces the instruction pointer register (RIP) to a predefined value, known as the thread entrypoint. SGX allows multiple threads to execute inside an enclave by creating a Thread Control Structure (TCS) for each thread, which keeps track of the thread stack, heap, data, registers, and RIP values. When an application thread finishes executing the function inside the enclave, it executes the `EEXIT` instruction. The system can also interrupt the thread execution because of a memory fault or a context switch, causing an Asynchronous Enclave Exit (AEX). In that case, the TCS is saved in the PRM and the processor returns to normal mode. The `ERESUME` instruction retrieves the previously stored TCS, puts the CPU in enclave mode, and resumes the execution [10].

Based on the example presented in Figure 2.6, there are five different cases in which SGX takes access control decisions:

1. Application thread in non-enclave mode tries to access data from thread T1 running in App 3 enclave: the hardware will verify if the address the application tries to access is inside the PRM range. Therefore, since the processor is in non-secure mode, the instruction will be aborted.
2. Application thread in enclave mode running in App 1 enclave tries to access data from thread T1 running in App 3 enclave: the first verification, presented in the previous case, will be successful since the processor is in enclave mode (`EENTER` was executed). Then, the instruction will access the metadata of the accessed page (EPCM). The EPCM indicates the location of the enclave's identity (SECS) who owns the page. Then, the instruction will verify if this retrieved identity matches the identity of the enclave that executed the instruction, passed as a parameter. Therefore, this verification fails, and the access is denied.
3. Application thread T1 in enclave mode running in App 3 enclave tries to access data from thread T1 running in App 3 enclave: the two first checks mentioned

above succeed. Then, the instruction verifies the read/write/execute permissions stored in the metadata of the accessed page (EPCM). If the instruction being executed is allowed for the page, the instruction will pass this verification. Last, it verifies if the virtual address used by the application's instruction matches the page's expected virtual address, stored in its metadata (EPCM). If so, the access is finally granted.

4. Application thread T2 in enclave mode running in App 3 enclave tries to access data from thread T1 running in App 3 enclave: the access is granted, since threads of the same process typically have shared memory.
5. OS thread tries to access thread T1 running in App 3 enclave: the OS always runs in non-secure mode, such that the access is denied as in the first case.

2.4.2 Encryption and Attestation

Each SGX CPU has unique One-Time Programmable (OTP) devices used to store a secret, which can only be used to derive keys, such as the sealing key. This key derivation is performed by the `EGETKEY` instruction. An application enclave must encrypt sensitive data with this sealing key before writing into the disk. Since the sealing key is unique to the enclave and cannot be accessed by other software, only the same enclave on the same processor can retrieve the sealed data [30].

All code, data, and metadata used during enclave creation (2.4.1) are hashed, creating a unique enclave measurement (`MRENCLAVE`). The measurement is performed by the enclave creation instructions and is stored in the enclave metadata, ensuring its integrity. If the untrusted OS modifies the code at the enclave creation, the measurement does not match the developer's expected value. This measurement is used to create a report, which can be signed by a privileged enclave developed by Intel, called quoting enclave.

Before sending its data to an enclave in the cloud, the client must ask the server to perform a remote attestation. The attestation proves that the server runs on a legitimate SGX-enabled platform with the expected code. Intel's Data Center Attestation Primitives (DCAP) is an attestation mechanism based on PKI [31]. In this mechanism, Intel is a trusted CA that provides the CPUs with a unique private

attestation key and emits certificates with the corresponding public key. A pre-made Intel enclave, called quoting enclave, uses a unique private key, in the format of the x509 certificate, to sign the enclave report. This signed measurement is forwarded to the client, and the enclave is considered trusted if the signature can be verified using the data center public key provided on the certificate. The client can also compare the enclave initial measurement, done at the initialization step, with an expected value to ensure that the enclave code loaded into the main memory is the expected one. The attestation protocol is based on Elliptic Curve Diffie-Hellman Key Exchange (ECDHKE) between the client and server enclave to transmit the signed report using symmetric encryption [32]. If the attestation is successful, the client can transmit confidential secrets over this encrypted channel so that only the enclave can decrypt them.

Intel also offers a service called Enhanced Privacy ID (EPID), in which the client needs to access an online centralized Intel Attestation Server (IAS) to verify the validity of the signature [30]. DCAP, described above, eliminates relying on a single verification point since Intel servers are only requested in the provisioning step. In this step, Intel servers provide the quoting enclave with the private key that never leaves this enclave used to sign the measurement, the corresponding public key, and the lists of revoked certificates. SGX platform offers a CSPRNG for generating secrets without depending on any other potentially malicious software components. This can be used to generate the ECDHKE secrets or as a seed to key generation algorithms.

SGX trust model assumes that the attacker cannot tamper with the circuitry inside the CPU package. However, attackers could still tamper with the buses that connect the CPU to the main memory. Therefore, SGX CPUs are equipped with a Memory Encryption Engine (MEE) responsible for encrypting data transferred from the cache to the main memory and decrypting data in the opposite direction. Data in the cache is plaintext to ensure fast access since an attacker cannot access the processor internals [10].

2.4.3 Application Development

Intel offers a Software Development Kit (SDK) with high-level C/C++ functions for using SGX functionalities. The developer must choose the functions residing inside the enclave and write them in separate files. These functions are compiled into a dynamic library (enclave.so, for example), which can be used by other applications to process sensitive data. Since system calls are untrusted OS code, they cannot be executed inside the enclave. Therefore, memory allocation, file descriptor operations, threads synchronization with semaphores, Inter-Process Communication (IPC), and child process creation must be outside the enclave [33].

A non-enclave function may want to call an enclave function passing a pointer to some buffer at the non-protected memory as an argument. However, the enclave can reveal sensitive information if it manipulates this buffer directly. Therefore, the application outside the enclave calls an intermediate function with the same name as the enclave function, called ECALL edge routine, which copies the buffer to the protected memory. Similarly, when an enclave function wants to call a non-enclave function, it calls an OCALL edge routine, which copies the buffers from the protected to the non-protected memory [33]. Intel offers the Edger8r (*edgerator*) tool for automatically generating C++ code containing edge routines. The developer must write an Enclave Definition Language (EDL) file for the edge routine generation, as illustrated in Code 2.1.

```
1 // Enclave function headers that may be
   // called by non-enclave code
2 trusted {
3 // public indicates that it can be called
   // by any non-enclave function.
   // sgx_status_t is the return type.
4 public sgx_status_t process_sensitive
5     ([in, size=in_size] int* buffer_in ,
6      int in_size ,
7      [out, size=16] int* buffer_out );
8 // The size field before each pointer
   // argument indicates the maximum number
   // of bytes of the buffer.
9 }
```

```

10 // Non-enclave function headers that may
    be called by enclave code
11 untrusted {
12 public sgx_status_t call_insec ();
13 }

```

Code 2.1: Enclave Definition Language (EDL) file used by Edger8r to generate .cpp and .h files containing an ECALL for `process_sensitive()` and an OCALL for `call_insec()`.

2.4.4 Performance Limitations

The main performance limitations brought by SGX are EPC page eviction, transition overhead, cache misses, enclave creation, and communication between enclaves. The EPC page eviction happens when the system runs out of EPC space and stores some pages outside the PRM. This degrades performance since the evicted pages must be encrypted when evicted and decrypted when restored. This problem is more evident in older SGX versions, found in Intel Skylake platforms, where the EPC was limited to 128MB, from which only 90MB were available to enclaves while the rest were metadata. The transition overhead introduced by ECALLs and OCALLs increases with the amount of data copied from buffers into/out of the PRM. With enclaves, cache misses are a more serious problem because when memory content is not cached in CPU, it must not only be retrieved from the main memory, as in untrusted computations, but also be decrypted. The biggest performance limitation is enclave creation since multiple memory pages must be copied and measured from the non-protected memory to the PRM [33]. In cloud computing, it is common to distribute processing in multiple computers to achieve parallelism. However, since enclaves do not trust each other, they must perform remote attestation for sharing a secret key. All data must be transmitted over this encrypted channel, introducing significant overhead [12].

2.4.5 Security Limitations

The main SGX security limitations are denial-of-service attacks, public code, unprotected peripherals, application code vulnerabilities, and side-channel attacks.

The need to trust Intel is an intrinsic security limitation when using SGX.

Enclaves do not protect against denial-of-service attacks since, for example, the untrusted OS can refuse to create an enclave. Also, while the data processed inside the enclave is confidential, the enclave code is not, since the untrusted OS knows the enclave code and initial data for creating the enclave. Moreover, clients aiming to use an enclave in the cloud must know the enclave code to verify if the measurement is correct. Silva *et al.* propose a tool to protect SGX code privacy by dynamic loading code inside the enclave whenever needed [34]. Enclaves do not offer any means for protecting peripheral’s data. The developer is responsible for writing secure codes, avoiding memory leakage and overflow issues, validating every input that comes from non-enclave code, and avoiding shared memory between threads to store secrets. Minimizing enclave code reduces the chance of vulnerabilities while processing sensitive data.

Side-channel attacks can happen through OS kernel manipulation to infer the application memory access pattern by measuring computation time [35]. The most common side-channel attack regarding SGX is the cache side-channel attack, in which the attacker uses a malicious non-enclave process to fill the cache used by a given core executing enclave code. Then, whenever enclave pages are accessed, they replace existing attacker pages in the cache. Periodically, the malicious process verifies if the time to access its data in a certain address is high. If so, the content there was evicted from the cache, and this location was accessed by the enclave [36]. Information obtained this way is reverse-engineered into actual data since the attacker has access to memory translation tables [37].

Chapter 3

CACIC Architecture

This chapter uses SGX to enhance the security of IoT data processing in clouds. Figure 3.1 illustrates the architecture composed of clients that send data to or receive data from a trusted cloud server. The communication uses HTTPS because it is a secure and widely adopted protocol in web servers. However, the sole deployment of HTTPS is not enough to meet the security requirements proposed in Section 2.3. Hence, the architecture leverages memory enclaves to equip the system with the four secure procedures: registration, publication, query, and revocation.

3.1 Trusted Protocols

This section describes the secure procedures followed by the clients and the server, illustrated in Figure 3.2.

3.1.1 Registration

Initially, the client access point must register in the platform (1) to share its symmetric Communication Key (CK) with the server, used to encrypt and decrypt data, as illustrated in Figure 3.2. During registration, the client access point must attest that the server is trustworthy (2), as presented in Subsection 2.4.2. The access point sends the CK to the enclave if the attestation is successful. The attestation procedure involves multiple messages in both directions for key exchange using Diffie-Hellman, as indicated by the double arrows in the diagram. The server seals the key before writing it on disk and associates it with the client ID: an eight-digit

hexadecimal string that identifies who is sending the messages.

3.1.2 Publication

When the access point receives data from sensors for publication (3), it assembles a message ($M[\text{publication}]$, 4) with the following content:

$$M[\text{publication}] = [\text{time} | \text{ID} | \text{type} | \text{size} | \text{nonce} | \text{CK}(\text{content} | \text{perm} | \text{nonce})].$$

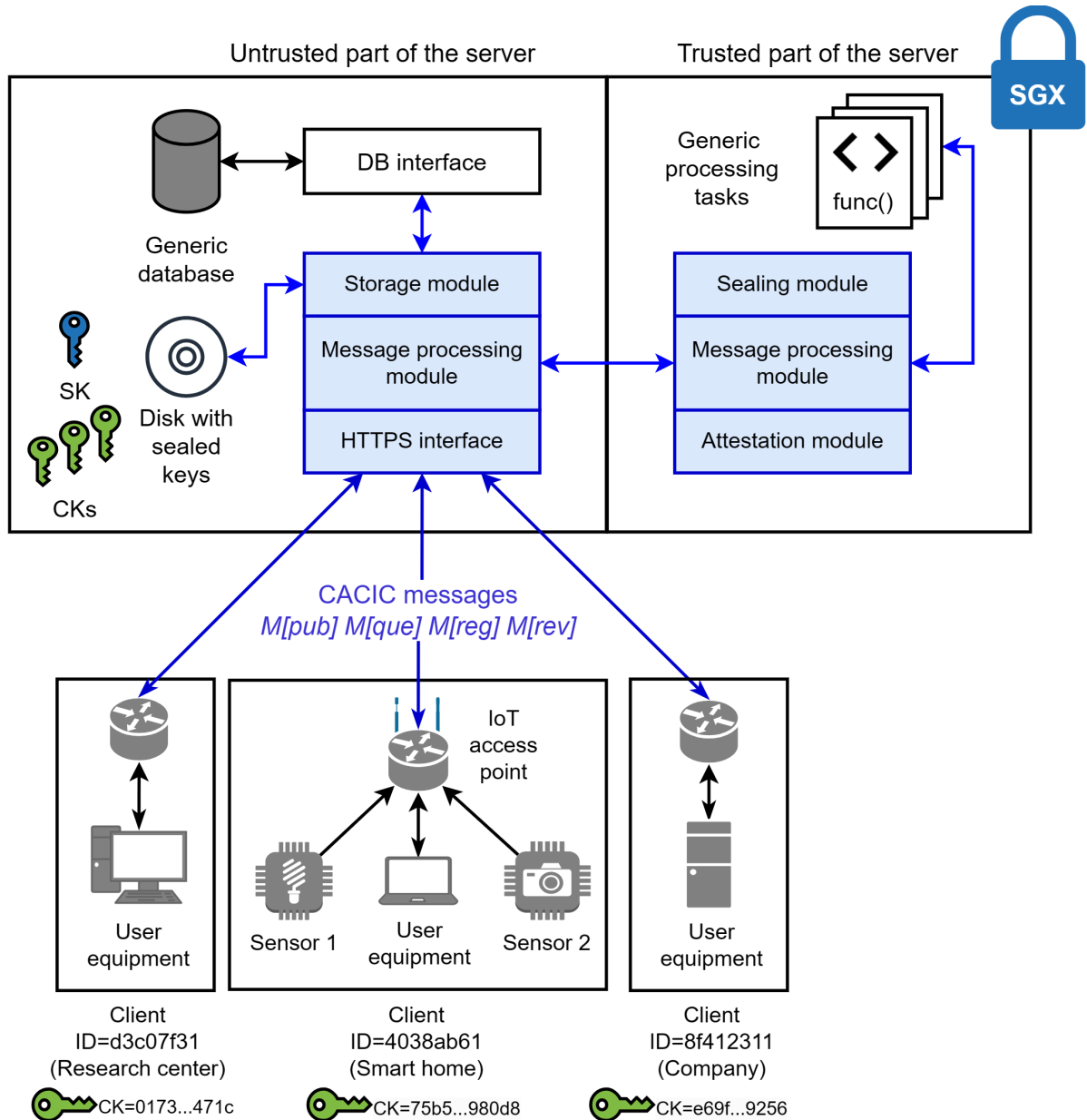


Figure 3.1: Components and interfaces in blue compose CACIC core, i.e., they do not depend on the use case. The architecture is agnostic to the processing task, the database, and the data sources.

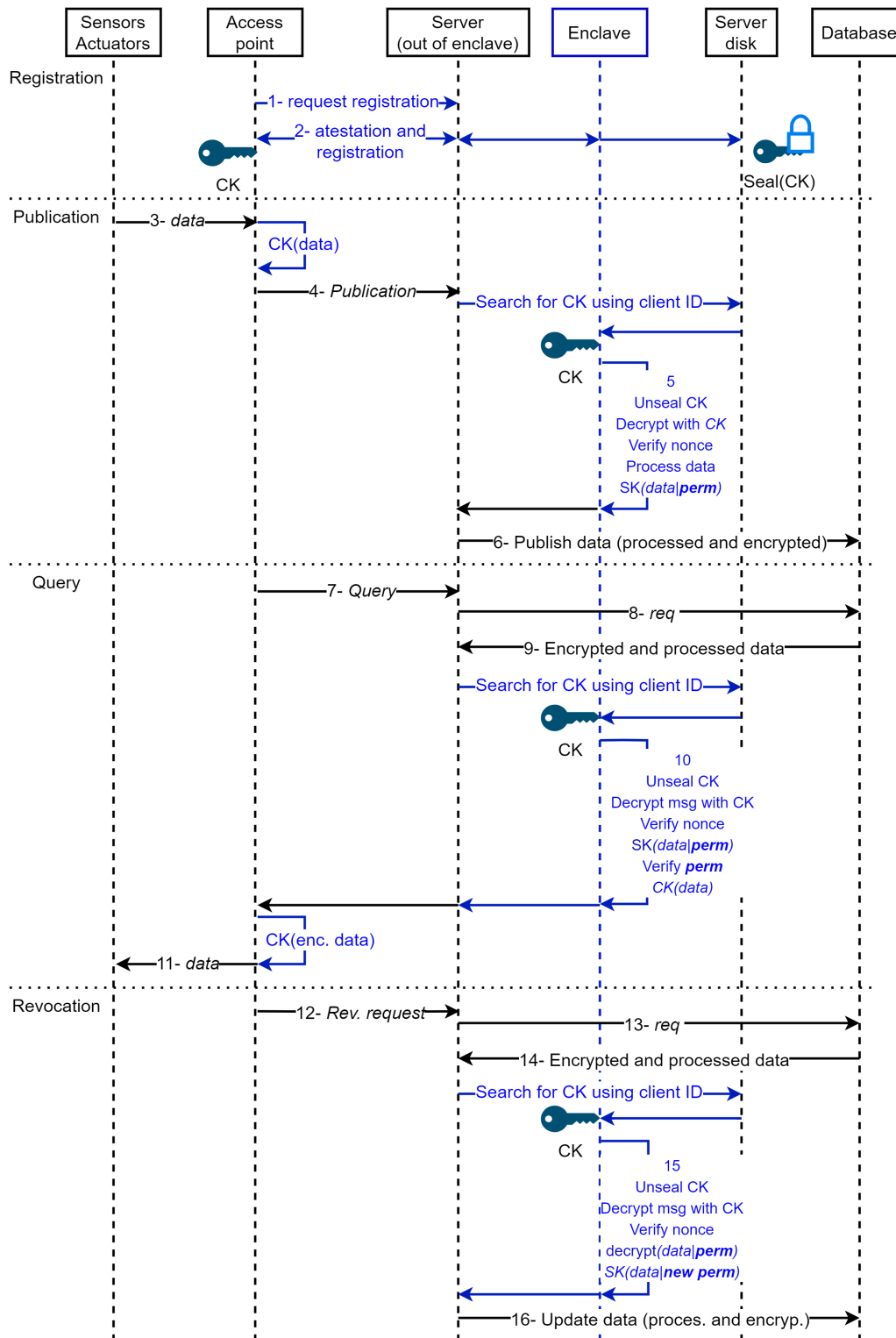


Figure 3.2: Data in transit is encrypted with the Communication Key (CK), and stored data is encrypted with the Storage Key (SK). Only the enclave decrypts and processes the data in the cloud. The blue color highlights the security mechanisms introduced by the architecture.

The symbol $|$ represents string concatenation. If the application needs, the `time` can be used as an index for post-lookup procedures at the database. The public ID identifies the client to the server, which locates the CK sealed on disk. The content `type` is used by the server to identify the appropriate processing task for each case. The sensor data is stored in the `content` field. Client access permissions (`perm`) permit the server to be aware of who can access the published data using a list of IDs. In addition, permissions are encrypted together with the data, as they are private information. The `nonce` is a random number sent to avoid replay attacks. Once the publication message is received, the server enclave retrieves the communication key, decrypts the data, verifies if the nonce is fresh, and applies some processing depending on the content `type` (5). The result and the permissions list are encrypted with the Storage Key (SK), which is sealed in the disk. The enclave also verifies if the encrypted nonce is consistent with the plaintext nonce.

The `M[publication]` message can also be used for requesting the server to apply some processing over previously published data, making this data available to other clients. In such a case, the `type` field identifies the processing task, while the `content` field can be used to specify processing task parameters. Some processing tasks may require the server to use previously published data, such as for building predictive models, for example. In such a case, the `content` indicates the data that should be used to perform the processing task. Applications running on clients' personal computers can send `M[publication]` to the server for this purpose.

To illustrate a common system application, let us consider the aggregation of a client's energy consumption data. The client sends a `M[publication]` requesting the server to aggregate energy consumption data using `content` and `type` fields. After receiving the message and identifying the type, the server reads from the database other energy consumption data according to the parameters received in the `content` field. Therefore, the enclave can decrypt the samples and calculate a sum, as described in Algorithm 1. With the proposed architecture, a control center can use this data to make decisions regarding energy distribution planning, for example, without accessing confidential consumption patterns. Other proposals use homomorphic cryptography for computing the sum with encrypted data. However, enclaves stand out for introducing a much smaller processing delay and allowing

Algorithm 1 Algorithm for aggregating data inside the enclave. The MAC and nonce checks are omitted. The enclave uses the result to build an encrypted sample for publication.

Input: ($size, enc_data[size], sealed_CK$)

Output: sum

$sum \leftarrow 0$

$CK \leftarrow \text{Unseal}(sealed_CK)$

$counter \leftarrow 0$

while $counter \neq size$ **do**

$data \leftarrow \text{Decrypt}(enc_data[counter], CK)$

$payload \leftarrow \text{GetPayload}(data)$

$sum \leftarrow sum + payload$

$counter \leftarrow counter + 1$

end while

arbitrarily complex operations on the data, such as filtering [21]. Before publication in the database, the data is encrypted with the access permissions and the nonce, using the SK.

3.1.3 Query

In the query procedure, the client access point sends a message $M[\text{query}]$ (7) with the following format:

$$M[\text{query}] = [\text{ID} | \text{index} | \text{command} | \text{size} | \text{nonce} | \text{CK}(\text{nonce})],$$

where `command` and `index` are used to locate and request data in the database. The request is forwarded to the database (8), and the response is the encrypted data (9). Then, the server enclave retrieves the client `CK`, decrypts the nonce with the `CK`, and verifies if the nonce is fresh and consistent with the plain-text nonce (10). Therefore, it decrypts the data received from the database and checks if the access permissions (`perm` stored with the data) allow access to the data by the interested client (10). If so, the enclave encrypts the result with the `CK` and sends it to the access point. Finally, the access point decrypts the received data using the `CK`, making it available to the interested device (11). The architecture is database-agnostic so that

the format of the `index` and `command` fields of the query message depends on the application. The cloud provider is free to deploy the database on the same machine as the server, on a separate server, or even in a distributed fashion.

An advantage of CACIC is that it does not impose any computational requirements or rigid message format on the sensor. This advantage is a consequence of delegating the key management and data encryption to the access point, regardless of the message format used by the sensors. The diversity of IoT devices in terms of protocols and performance specifications justifies this design choice, making the proposal flexible and device-agnostic [38]. However, if the sensor has cryptographic, networking, and key storage capabilities, it can directly send the messages: the access point would act as a simple network gateway. Still, delegating these operations to the access point is simpler because the `ID` and `CK` can be maintained in a single device. Additionally, the sensor can employ more lightweight network protocols, such as the Constrained Application Protocol (CoAP), since the access point can act as a proxy. The client can send messages using its own devices, such as a personal computer, to request the server to perform processing tasks (`M[publication]`) or to query data (`M[query]`).

A key feature of the architecture is the possibility for the client to configure access permissions. The `perm` field at the publication message (`M[publication]`) is used by the enclave to prevent or allow access when a client requests this data. A client can prevent the biomedical signals from their smartwatch from being made available to the watch manufacturer but can allow them to be accessed by a biomedical research institution, for example. This is only possible because the client has established a trustworthy relationship with the trusted execution environment in the cloud. Also, if a company sends a `M[publication]` requesting the server to perform some processing task over previously published biomedical data from other clients, the server can refuse to use data whose access permission list does not include the company `ID`.

The access control mechanism is based on sticky policies, first proposed by Karjoth *et al.* [39]. Sticky policies are stored and transmitted along with the data, providing the data producer with in-depth control over access rules for every single data unit [40]. This approach relies on a Trusted Authority (TA) for managing

decryption keys and enforcing access control, which is the enclave in this proposal. The system uses Access Control Lists (ACL) containing the identity of the clients allowed to access the data.

3.1.4 Revocation

The client can revoke the access permission or even completely remove the data, sending a revocation message in the following format:

$$M[\text{revocation}] = [\text{ID} | \text{index} | \text{command} | \text{size} | \text{nonce} | \text{CK}(\text{perm} | \text{nonce})].$$

The server first queries the data published together with the access permissions from the database using an `index` and a `command` (13, 14). The enclave verifies the nonce and unseals the data as described for other procedures. Therefore, it updates the access permissions of this data with the new permissions sent on the `perm` field (15). If this field in the message is null, this means that the data must be removed from the database. If not, it overwrites the database entry with new data containing new access permissions (16). Essentially, the revocation procedure combines querying old data and publishing a new one.

3.2 Security Evaluation

The architecture uses cryptographic primitives to achieve the first four security requirements, presented in Table 2.1. The system uses AES with 16-byte keys because this is the default symmetric encryption algorithm offered by SGX SDK. Therefore, an attacker without access to the cryptographic keys cannot access encrypted data, ensuring the confidentiality of encrypted data. The encrypted data are accompanied by a 16-byte MAC, computed using GCM, implemented in SGX SDK. This ensures that: i) the encrypted data was generated by someone with the expected key, and ii) the encrypted data was not modified. Therefore, authentication and integrity are ensured if an attacker cannot access the cryptographic keys. The encrypted data is also accompanied by a nonce, implemented using a 12-byte random IV for the AES-GCM algorithm. When data is decrypted, the system can verify if this value is unprecedented, such that an attacker cannot replay old messages, ensuring freshness. Therefore, it is safe to say that the first four properties

of Table 2.1 are ensured if: i) sensitive data is always in its encrypted form, except inside the trusted modules, ii) trusted modules must perform all AES-GCM operations as well as its corresponding security checks, and iii) the encryption keys are never accessed by an attacker following the attacker model defined in Section 2.3. Trusted modules are defined as systems components whose security does not depend on the attacker actions defined in Section 2.3, which, in this work, include the client LAN, the server enclave, the Intel system for provisioning keys and certificates, and the Intel CPU package. Thus, the server must access plaintext sensitive information only inside the enclave and prevent an attacker from accessing cryptographic keys. To ensure the last security requirement, access control, only trusted modules should modify access permissions and read access permissions to grant access to sensitive data.

Two trusted components must also be able to share the symmetric key through a channel controlled by the attacker. This is achieved using the ECDHKE (2.4.2), which is the key exchange algorithm implemented in SGX SDK. Given that the secret parameters are securely generated inside the trusted components and are never revealed outside them, the attacker cannot perform man-in-the-middle attacks on the communication channel between components. Therefore, keys must always be generated and exchanged between trusted components. Remote attestation is used by the client to ensure enclave authenticity.

Based on these principles, the next Subsections discuss how the mechanisms described in Section 3.1 make the system resilient to an attacker performing the actions described in Section 2.3. This work does not prove the system security, since formal methods are out of the scope. More information about formal methods for enclave security can be found in [41].

3.2.1 File Manipulation

An attacker who gains privileged access control over the server may read data from arbitrary files, including the database containing the client’s data and the access permissions. The database is encrypted with the **SK** using AES-GCM for ensuring confidentiality. Both the **SK** and the client’s **CK** are sealed on the disk so that only the enclave can decrypt these keys. It is assumed that no one can tamper

with the SGX hardware, making it impossible for the attacker to obtain the sealing key.

An attacker may also write to any file, modifying the client's data and access permissions in the database. Since AES-GCM ensures data integrity, the enclave cannot decrypt data fabricated by the attacker, thus denying the operation. Consequently, the attacker cannot tamper with the access control list stored with the data. This work does not deal with system availability issues such as forcing data to be unuseful by breaking its integrity, deleting contents from the disk, or simply shutting down the server by the cloud provider. Orthogonal solutions such as database replication for backup are outside the scope. Now, suppose a client publishes a boolean data identifying if a door lock in its house is open. The attacker could replicate old data in the database to force a stale state, putting client security at risk. However, the cryptographic nonce ensures data freshness, blocking replay attacks.

3.2.2 Packages Manipulation

The attacker can also intercept network packets to read or modify message contents. Since sensitive data and permissions in all messages are encrypted with the CK using AES-GCM, the attacker cannot tamper with its confidentiality, integrity, and freshness. The client generates the CK during the registration, without affecting the security of the proposal, as the attacker model assumes that the client is trustworthy. The client is responsible for maintaining its CK confidential. During attestation, the client shares the CK with the server enclave using ECDHKE securely, since both ends are trusted. After attestation, an attacker at the server will not be able to access the CK because it is sealed on the disk and can only be retrieved by the enclave. The attacker is not able to impersonate a client to send fake messages because all messages have an encrypted field that acts as proof that the sender possesses the CK. Availability attacks such as Distributed Denial of Service (DDoS) or packet drop attacks are not considered.

3.2.3 Software Manipulation

The attacks described in Subsections 3.2.1 and 3.2.2 do not involve modifying currently running applications or running a malicious application. This work goes

beyond most works in the literature by assuming an attacker can tamper with the existing server code to introduce malicious tasks, such as revealing client data in the clear. The OS is also not trusted, so an attacker can read, write, and execute code in every server memory region outside the one reserved for the enclave. To avoid these attacks, data is processed in clear inside the enclave, a trusted entity that guarantees its security at runtime. All sensible information, like CK, SK, data contents, access permissions lists, and encrypted message fields are only decrypted after entering the enclave and only encrypted before leaving the enclave, as shown in Figure 3.2. Therefore, all the integrity, freshness, and authenticity checks concerning AES-GCM-protected data are only performed inside the enclave.

The attestation protocol at the registration phase guarantees that the server does not falsify the authenticity of the code that runs inside the enclave. For this, the client checks if the signed measurement of the enclave code matches an expected value. As described in Section 2.4.2, the signature relies on a trusted system enclave owned by Intel that cannot be tampered with by any other component in the software stack. The client does not proceed with the communication if the code measurement is not the expected one, or if the signature cannot be verified, indicating that the server does not leverage a genuine enclave. Since only the enclave processes the access control rules, the client has complete control over who accesses its data. The SK is generated at the first enclave startup by a trusted software component, the SGX CSPRNG, and is sealed on the disk for future use.

3.3 Performance Evaluation

The performance evaluation aims to: i) verify if the architecture is scalable, ii) measure the processing latency introduced by enclaves, iii) identify the main performance bottlenecks, iv) evaluate the CPU and memory overhead, and v) verify how the architecture deals with a processing task that operates over a large amount of data instead of just publishing the received data. IoT cloud systems need to serve a large number of clients at the same time, given the growing number of devices. This challenge justifies evaluating the performance of security proposals with numerous of clients [42]. The experiments evaluate the number of requests the platform processes

per second and the time for publishing and querying data. The trusted server was implemented in a computer with an Intel i9-10900 CPU 2.80 GHz, 32 GB RAM, and 20 threads. This machine also sends messages for querying and publishing synthetic data in the *Ultralight* 2.0 format, a standard adopted by the FIWARE platform for developing intelligent applications used in other works [43, 15, 44, 45]. Still, the data format supported by the platform is generic, as described in Section 3.1. A Software Development Kit (SDK) for using the Intel SGX trusted instructions in C++ language was used¹. During the experiments, the EPC page eviction problem discussed in Subsection 2.4.4 did not occur.

3.3.1 Scalability

The first experiment evaluates the number of publication messages a server can process per second, which includes steps 4 to 6, as denoted in Figure 3.2. The experiment reproduces multiple clients sending publication messages at the same time with a constant rate of publications per second (pub/sec). The `wrk2` tool sends constant HTTP workloads, measures the server processing rate, and evaluates the performance statistics [46]. The experiment was repeated without the enclave (step 5 in Figure 3.2) to evaluate the overhead added by the proposed security mechanisms. Figure 3.3 shows that the publication rate increases rapidly with the sending rate as the system processes requests in parallel using multiple threads inside the same enclave. For less than 4k messages sent per second, the processing rate is similar to the sending rate, suggesting that, until this point, the server does not reach its processing limit yet. As the sending rate increases, the curve inclination decreases until the processing rate stabilizes at a maximum value, representing the server processing capacity limit. The maximum processing rate is 5,500 pub/sec for the secure server (with enclave) and 6,250 pub/sec for the insecure server (without enclave). This represents a 12% decrease in the maximum publication processing rate, which is small considering that CACIC architecture can still serve thousands of clients per second with a much stronger threat model.

Figure 3.4 reveals that the latency of a single publication message is in the

¹The repository with the CACIC core implementation for evaluating the performance is available at <https://github.com/GTA-UFRJ-team/TACIoT>.

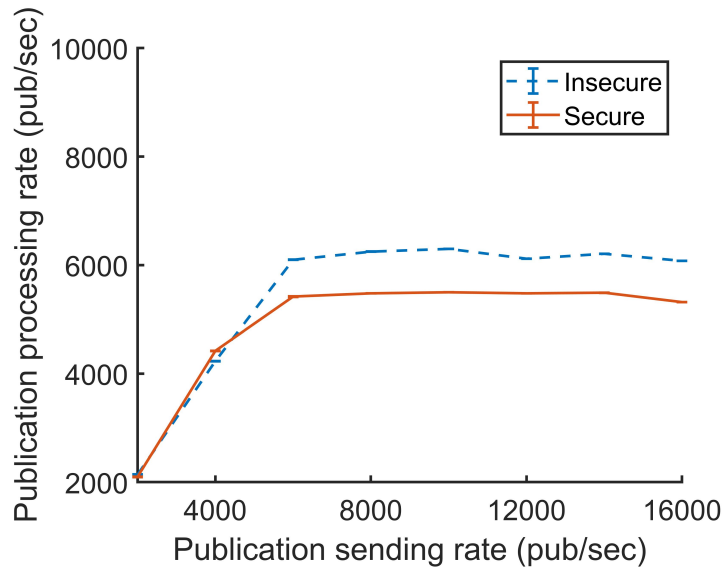


Figure 3.3: The enclave reduces the maximum publication rate by 12%, but the system still processes thousands of publications per second.

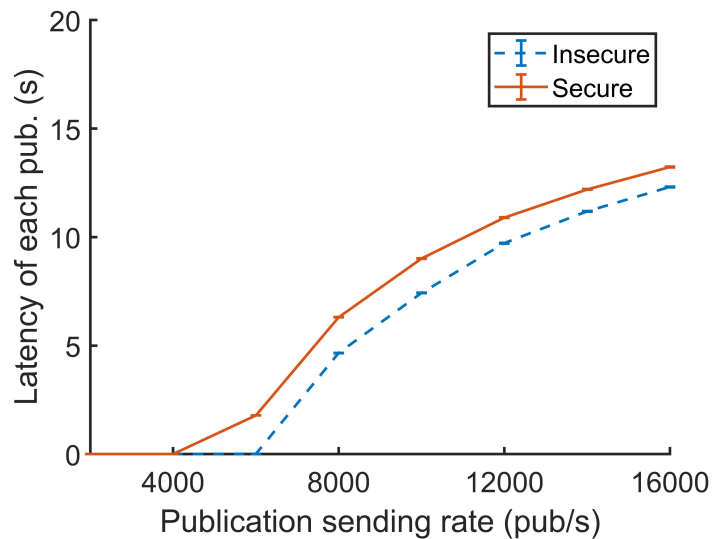


Figure 3.4: The latency quickly increases when the system achieves its maximum publication rate since new messages are inserted into a queue.

order of a few milliseconds when the system is below its processing rate limit. When the system achieves its maximum parallelism, every new message is inserted into a queue and is only processed when a thread becomes available. This is confirmed by the rapid growth in latency around the maximum processing rate. For the secure case, at 4K pub/sec the latency is negligible while at 6K pub/sec the latency is already significant, since the maximum processing rate is 5,5K pub/sec. The client

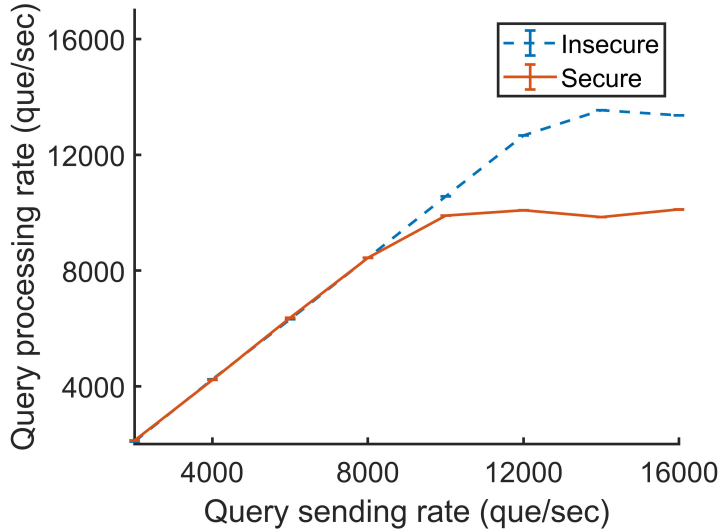


Figure 3.5: This result is analogous to Figure 3.3. The reduction in message processing rate due to the enclave is more significant in that case, since queries are faster.

will notice a response time in the order of seconds in both secure and insecure cases when the server is at its parallelism limit.

The second experiment evaluates the number of query messages processed per second (que/sec) by the server, which includes steps 7 to 10 in Figure 3.2. The methodology is the same as the one described for the first experiment. The results presented in Figure 3.5 are analogous to the first experiment, with a maximum rate equal to 13,540 que/sec without enclaves and 10,110 que/sec with enclaves. In both cases, the query rate is higher than the publication rate, suggesting that the query procedure is faster than the publication procedure. The overhead added by the enclave in the query is not that negligible, leading to a performance drop of 25% compared with the insecure case.

Figure 3.6 presents the latency of each query message, similarly to Figure 3.4, confirming that the latency increases rapidly when the server reaches its maximum processing rate. For the query messages, this maximum processing rate is much higher than for publication messages. As the sending rate increases, the curve for the secure server starts to grow before the curve for the insecure server. This happens because the maximum rate is much lower when using the enclave.

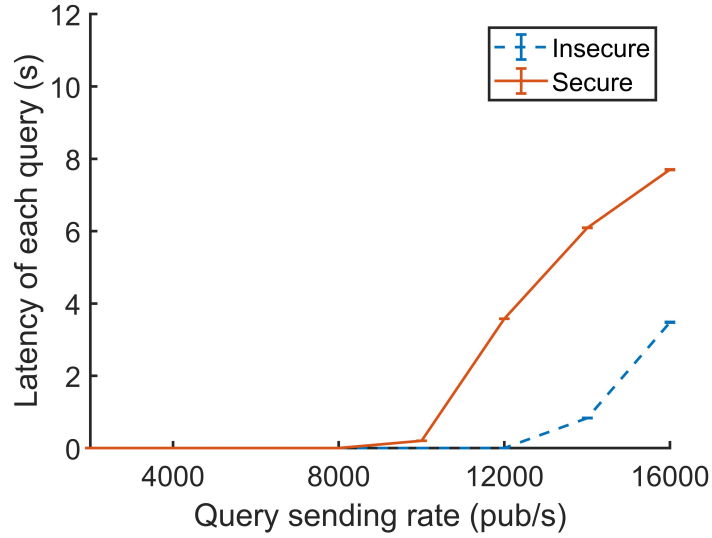


Figure 3.6: This result is analogous to Figure 3.4. The system processes more queries than publications per second.

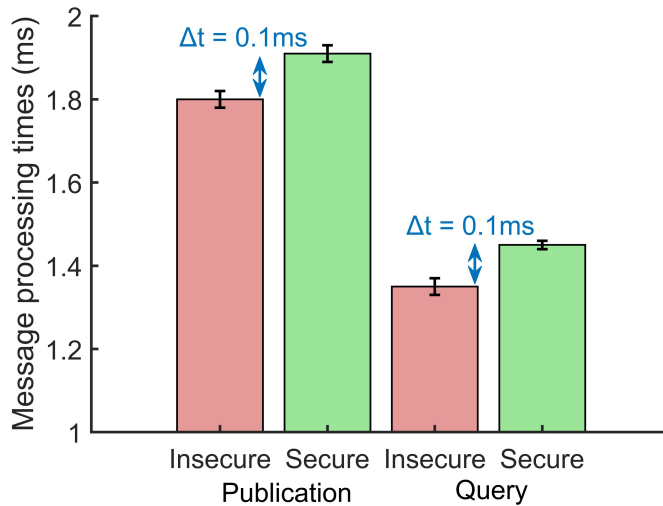


Figure 3.7: The enclave adds an imperceptible latency overhead for the clients. The enclave does not perform any computation-intensive processing and the overhead is mainly due to the ECALL.

3.3.2 Latency

Figures 3.4 and 3.6 also presented results concerning the processing time of messages, but considered a scenario with multiple concurrent messages being queued. The third experiment differs from the previous by evaluating the time between the client transmission of a single request and the reception of a successful response

using the `wrk2` tool. This experiment aims to verify if the enclave adds significant overhead in response time. Figure 3.7 shows that, in both cases, the enclave adds only 0.1 ms of delay, which is imperceptible for most applications. The enclave does not apply expansive computation over the published data, since the experiment only evaluates the publication/query messages processing times. The time introduced by the enclave processing is the same for publication and query because the system executes three sealing/unsealing and encryption/decryption operations in both cases, as presented in steps 5 and 10 of Figure 3.2. The experiment also confirms that publication time is approximately 25% higher than the query time for both secure and insecure servers. This result explains why the maximum query rate of the server is much higher than the maximum publication rate, as discussed before. The time added by the enclave has more influence on the total query time than on the total publication time since the query procedure is faster. This explains the significant reduction in the maximum query rate caused by the enclave, as discussed before. These experiments confirm an inverse relationship between the latency of a single request in seconds, and the maximum rate in requests per second. The performance of the revocation procedure must be analogous as it is a combination of querying old data and publishing it with updated access permissions.

3.3.3 Microbenchmarking

The previous experiments treated the server as a black box since the `wrk2` tool simulates clients interacting with the server through HTTP requests and measures the time and the rate by the responses. The fourth experiment analyses the system under the microscope, detailing how much each procedure contributes to the total overhead. To perform this analysis, this work develops a custom benchmarking tool based on high-precision timers, which individually measures the elapsed time of each critical code section and computes statistics, such as mean and variance, at the end of an experiment epoch, with 1 μ s resolution. The overhead added by HTTP processing was not evaluated. Table 1 presents the mean elapsed time in microseconds for each procedure. After repeatedly running the experiments, the error bar becomes negligible (lower than the resolution). The first row represents enclave initialization, which includes creating and transferring memory pages to

Table 3.1: Publication and query latency microbenchmark results. The publication time is dominated by the disk writing and the query time is dominated by the enclave.

Procedure	Elapsed time (μ s)
Enclave initialization	5848
Enclave publication message processing	105
Database write	387
Other procedures	42
Total publication time (not considering HTTP)	534
Enclave query message processing	105
Database read	56
Other procedures	74
Total query time (not considering HTTP)	235

the PRM, as described in Section 2.4.1. The following four rows relate to the publication procedure. In that case, the bottleneck is the time to write to the secondary storage, which does not depend on the proposed architecture. The last four rows relate to the query procedure. In that case, the smaller time to access the database led to much better performance. The bottleneck for the query becomes the enclave, explaining why removing the enclave from the query procedure results in a substantial performance improvement (Figures 3.5 and 3.6). Table 1 also reveals that just the enclave initialization time is 10 times higher than the total publication time and 25 times higher than the total query time, dominating the total overhead. However, the server takes advantage of multithreaded processing inside a single enclave to initialize the enclave only on the startup.

3.3.4 Resources Usage

The fifth experiment evaluates the CPU usage in terms of the percentage (%) of CPU time the kernel dedicated to the server process and of the physical memory usage in MB. The `psrecord` tool is used to monitor the server’s resources while receiving 2k HTTPS messages per second for 30 seconds. Figure 3.8 shows the CPU usage. The values can be greater than 100% because the result is the sum of all

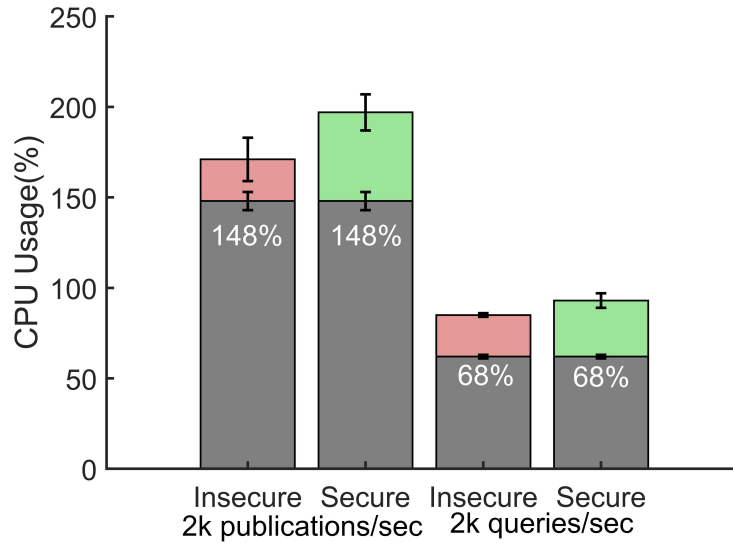


Figure 3.8: Percentage of CPU time for the server process, considering the sum of all CPUs. The resource usage is dominated by the HTTPS protocol implementation, as seen in gray.

percentages of time dedicated to the process by all CPUs. Thus, the maximum CPU usage for the entire 20-core machine is 2,000%. Initially, the publication and query functionalities were disabled to measure the resource usage by the HTTPS library alone, i.e., to create a baseline, presented in Figure 3.8 using gray bars. In that case, the mean CPU usage for receiving HTTPS messages is 148% for publications and 68% for queries.

Afterward, the system functionalities were re-enabled to measure the impact of the proposed publication and query protocols on resource usage. The CPU usage overhead is defined as the difference between the CPU usage while processing the requests and the CPU usage baseline, as defined before. For the insecure case, the CPU usage overhead is 24% for both publications and queries. For the secure case, on the other hand, the CPU usage overhead is 50% for publications and 32% for queries. The architecture does not impose significant CPU usage, considering that the overhead introduced by the security procedures is not very significant. Furthermore, it is evident that the HTTPS protocol dominates CPU usage and that the values are very distant from the maximum server capacity. The memory consumption maintained stable at around 7 MB for all experiments, indicating that the operations are not memory intensive. Repeating the experiment with rates

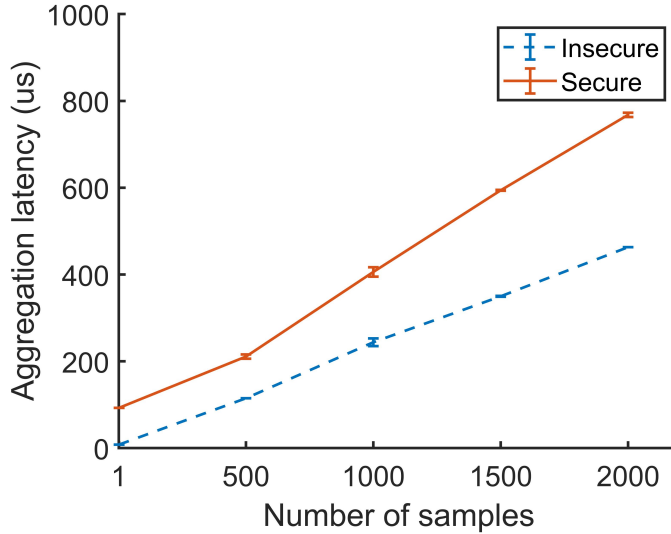


Figure 3.9: The enclave overhead increases as more data is transferred to it. However, the time for reading the samples from the database is much higher than this enclave overhead.

greater than 2k messages per second leads to similar results. These results confirm that high message processing rates do not require implementation with lots of CPU and memory resources.

3.3.5 Use Case: Aggregation

The sixth experiment evaluates the time taken to perform realistic data processing over fictitious IoT data samples, based on the use case of aggregation, described in Section 3.1. Figure 3.9 presents the time to aggregate the samples with and without enclaves. The experiment measures the time for aggregating 1; 500; 1,000; 1,500; and 2,000 samples to verify the influence of the number of samples. When the number of samples is 1, the system just decrypts the data and does not perform any sum. In both secure and insecure cases, the aggregation time grows with the number of data aggregated. This is expected since the asymptotic complexity of the Algorithm 1 is $O(n)$, where n is the number of samples, considering that it repeats the same efficient decryption and sum operations for every data sample.

However, the aggregation time with the enclave is always higher. This is a consequence of an additional step to execute an `ECALL` instruction to call the enclave and transfer the data buffer by reference before starting the described algorithm.

Other works demonstrated that the `ECALL` instruction is one of the main performance overheads introduced by enclave utilization, going from 4K clock cycles up to 20k clock cycles, depending on the amount of data transferred to the enclave [47, 33]. The difference between the secure and insecure curves represents the overhead added by entering the enclave. As the amount of data transferred to the enclave grows, the time taken to enter the enclave also increases, and the curves diverge. This is compatible with the results presented since the secure curve inclination is higher than the insecure case.

The time for reading a single sample from the secondary storage is approximately $56 \mu\text{s}$, as presented in Table 3.1. For thousands of samples, the total reading time is in the order of tens of milliseconds, which is much higher than the time for processing these samples. As the number of samples grows, the overhead introduced by the enclave increases, but the time for reading these samples from the database increases as well. This result suggests that the overhead introduced by the enclave can be imperceptible for the client if the application involves intensive I/O operations, as the bottleneck is on disk reading. These results are promising, considering that many applications, such as those requiring machine learning model training, are I/O intensive. Even though this is a straightforward conclusion, we still need more experiments to confirm it.

3.3.6 Validity Discussion

This section discusses some precautions taken to ensure the validity of the results and presents some limitations for generalizing the conclusions [48]. Experiments 1 to 5 aim to confirm the impact of the architecture and protocol, presented in Section 4, on scalability, latency, and resource usage. These experiments are direct measurements of two situations, contrasting only by the presence or not of security procedures, confirming the construct validity of the experiments. For internal validation, we repeated the experiments and performed statistical mean and standard deviation estimations to reduce stochastic errors. Also, we leverage widely adopted tools such as `wrk2`, C++ high-precision timers, and `psrecord` for conducting measurements to minimize the chance of systematic errors in instrumentation. Moreover, the obtained results lead to similar and coherent conclusions: i) publi-

cation performs worse than queries, ii) the enclave is the main bottleneck only for queries and iii) for publications, the time for writing into the secondary storage is the main bottleneck.

The validity of the results presents some limitations. Experiment 5 indicates that the system performs well in aggregating large amounts of data. However, more experiments must be run to ensure that the architecture does not hinder arbitrarily complex tasks. Some processing tasks may exit the enclave and re-enter it, while the models developed in Subsection 3.3.5 assume that the entire processing is performed inside the enclave once the thread enters the enclave. The proposed architecture is agnostic to the database, but the data is directly written into a file on the disk during the experiments. Therefore, in a production environment, the results depend on the overhead introduced by the deployed database. Still, the experiments confirm that even with the security procedures proposed, the system is scalable and presents low latency. The developer is responsible for choosing a scalable and resource-efficient database, and for programming the processing tasks in an optimized way to achieve high-performance requirements. Experiments run on a single computer, whereas cloud computing usually leverages virtualization and distributed processing and storage. Future works must deal with intensive processing tasks distributed between multiple virtual machines and multiple enclaves to extend the result's external validity to scenarios more similar to the cloud.

Chapter 4

Software Development Kit and Demonstration

This section presents CACIC-DevKit: a tool for developers to use CACIC architecture, described in Chapter 3, to build IoT systems. The development of this tool is motivated by the fact that CACIC architecture is flexible in terms of data sources, user interfaces, processing tasks, and databases. Therefore, the CACIC-DevKit tool aims to abstract CACIC’s security and access control mechanisms for developers not specialized in enclave development. The CACIC-DevKit code repository is available at <https://github.com/GTA-UFRJ/CACIC-DevKit>, and has more than 8,8k lines of code. The documentation can be found at <https://cacic-devkit.readthedocs.io/en/latest/index.html>. The chosen software license is MIT, allowing for modification and redistribution of copies as long as they include the copyright information provided in the repository. Finally, a video demonstrating the tool based on a use case is available at <https://youtu.be/CFEsD-25Mp0>.

4.1 CACIC-DevKit Functionalities

CACIC-DevKit offers functions, listed in blue in Table 4.1, for developers to build secure IoT systems. While functions in blue compose CACIC’s core, functions in black must be programmed by the developer, depending on the use case. Refer to Figure 3.1 for identifying the parts of the architecture that make up the core, repre-

sented in blue. The steps for developing a system with the proposed functionalities are summarized as follows:

1. program the processing tasks (Function 4) and associate them to a publication message `type`. The `M[publication]` fields are described in Section 3.1. The developer can call auxiliary functions (Functions 5, 6, and 7) from the CACIC's core for querying previous data, for example. These auxiliary functions automatically implement security verifications and access control;
2. program database interfaces (Functions 1, 2, and 3);
3. program the client interface, which can call Functions 8, 9, and 10 for interacting with CACIC;
4. program the data sources, which can call Functions 8, 9, and 10. A sensor can directly send `M[publication]` to the server or an access point can intercept sensor data and build the messages.

An advantage of decoupling core and use case application is the ability to abstract the security mechanisms, making it easier for non-specialists in the area to develop secure systems.

Figure 4.1 presents a possible execution flow of an application built using CACIC-DevTool functions. The developer programs the client interface, which sends a `M[publication]` using `client_publish()`. The developer defines how the message parameters are chosen and how the communication key (`CK`) is stored, according to the use case. Upon receiving the message, the server parses its fields, reads the `CK` and `SK` from the disk, and enters the enclave. Then, the enclave unseals the keys, decrypts the `M[publication]` encrypted fields, performs the security checks, identifies the processing task function using the `type` field, and calls the function. All these operations are performed inside the CACIC server's core, such that the processing tasks function receives the raw data and parameters. That way, the developer can program a function to process its data without worrying about enclaves, security checks, and message formats. After the enclave encrypts the result returned from the processing task with the `SK`, it calls `publish_db()`. Since it is the developer that programs this function, it will perform the publication according to the chosen database that best suits the use case.

Table 4.1: Functions in black must be programmed according to the use case, while functions in blue are part of the core of CACIC-DevKit. The documentation details each function’s arguments, as well as in which file it must be implemented.

Number	Function	Architecture Part that calls the function	Functionality
1	<code>publish_db()</code>	DB Interface	Inserts data into the database using a command
2	<code>query_db()</code>	DB Interface	Queries data from the database using a command
3	<code>multi_query_db()</code>	DB Interface	Queries multiple data from the database using a command
4	<code>task_name()</code> (user-defined)	Message processing module	Processes data according to the request received in M[pub] and generates a result
5	<code>enclave_query_db()</code>	Processing Task	Queries previously published data for use in the processing task
6	<code>enclave_multi_query_db()</code>	Processing Task	Analogous to the above function, but for multiple data
7	<code>enclave_get_payload()</code>	Processing Task Message processing module	Data is accompanied by metadata. The function returns the raw data.
8	<code>client_publish()</code>	User Equipment Sensor/Access Point	Builds a M[pub] and sends it to the server
9	<code>client_query()</code>	User Equipment Sensor/Access Point	Builds a M[con] and sends it to the server
10	<code>client_register()</code>	User Equipment	Builds a M[reg] and sends it to the server

4.2 Use Case and Demonstration

This section describes a use case for the tool based on smart grids. The goal is to demonstrate the tool’s compatibility with user interfaces, sensors with limited processing capability, and databases widely adopted in commercial systems. The use case also helps users interested in learning the tool, offering codes that exemplify the function’s usage. Eibl *et al.* demonstrate that data regarding energy consumption processed in the cloud reveal private information, such as the number of people using a facility at a given time [6]. In this scenario, CACIC allows energy companies or research centers to access energy consumption patterns without accessing individual data generated by smart meters. For this purpose, the client can send a publication message requesting the server to aggregate samples in the enclave and make the processing results available to the selected institutions.

The system demonstration uses a sensor designed with an ESP32 development board featuring a 32-bit Dual-Core microcontroller, running at 240 MHz, with 420 kB of RAM and WiFi 802.11b/g/n connectivity. The sensor also uses a SCT-013 current sensor and a ZMPT101B voltage sensor. The sensor’s firmware, programmed in C++, processes voltage and current signals with the help of the EmonLib library.

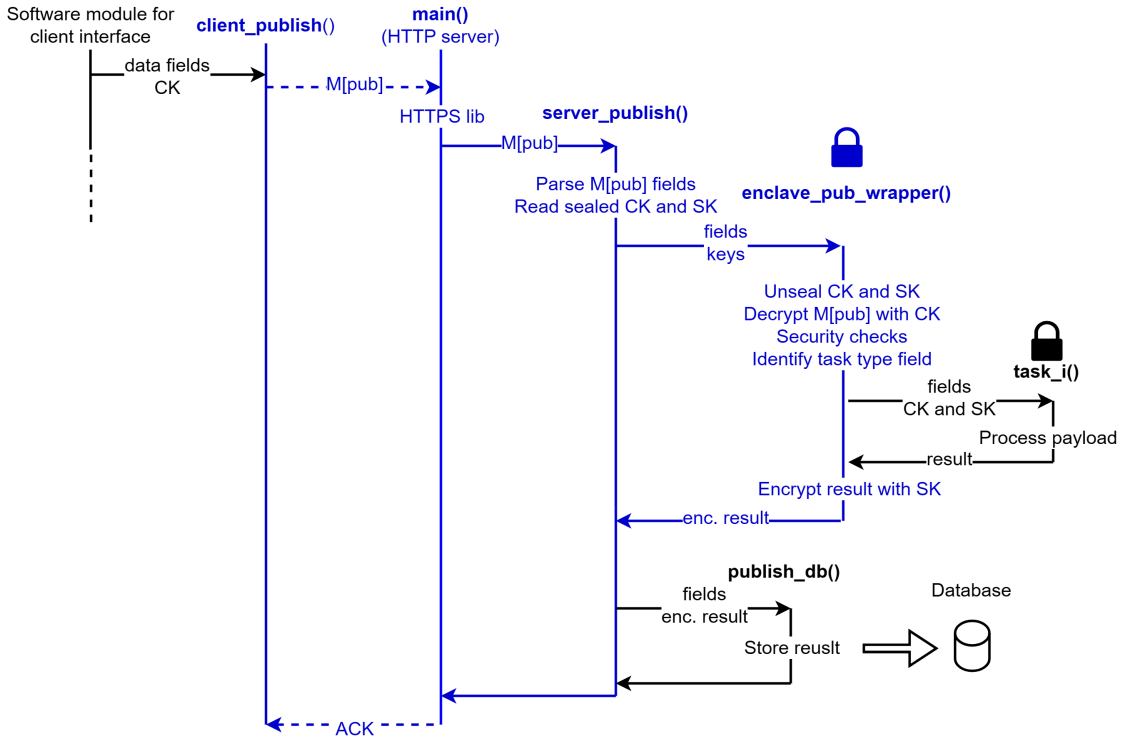


Figure 4.1: Execution flow for a $M[\text{publication}]$. The functions in blue are implemented in the core, while functions in black must be programmed by the developer. The solid arrows represent a function call or return, while the dashed arrows indicate a network message. The lock represents enclave functions.

It periodically sends consumption data using the HTTP protocol to an access point. The architecture is agnostic to the communication protocol within the client’s local network, as the threat model does not account for client-side attacks. A Raspberry Pi 3 Model-B, equipped with a 64-bit Quad-Core CPU running at 3.2 GHz and 1 GB of RAM, runs a WLAN access point software in C++. This software intercepts data from the sensor and constructs the publication messages for CACIC. The server is the same used for evaluating the architecture performance in Section 3.3, and also hosts an SQLite 3 database management system. The client uses a Linux computer running a management application with both a command-line interface and a graphical user interface, implemented using Qt Framework 6.4.1, as shown in Figure 4.2. This application is used to remotely configure the access point and to interact with the server using CACIC’s messages. Figure 4.3 illustrates the topology used for this demonstration.

The demonstration starts with the client registering itself using the manage-

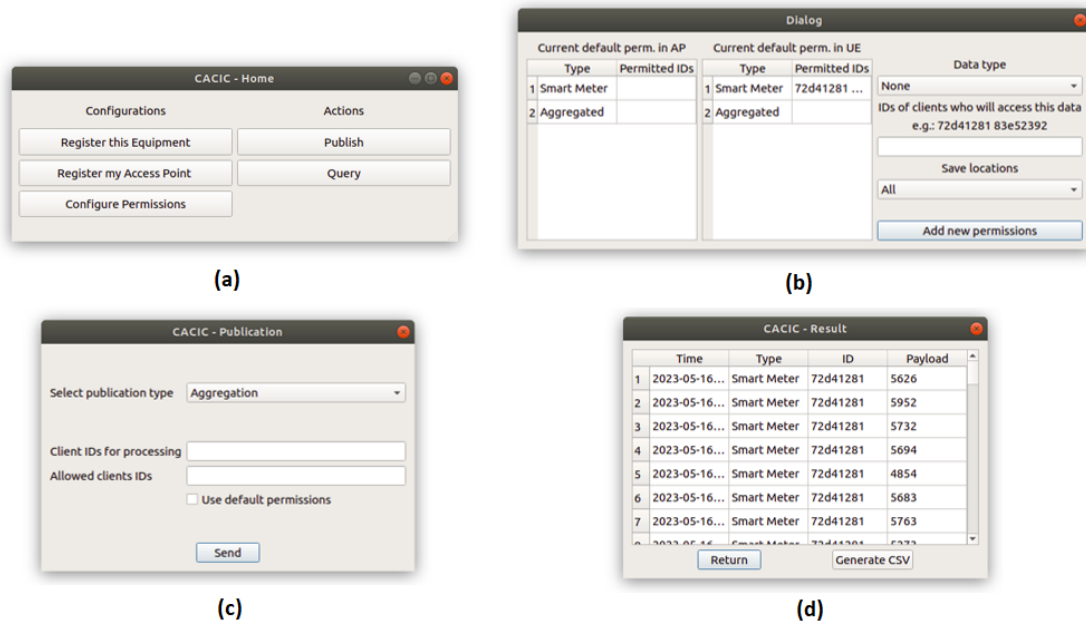


Figure 4.2: (a) Start menu. (b) Menu for configuring the IDs permitted to access each type. (c) Publication request menu. (d) Window with the result of a query.

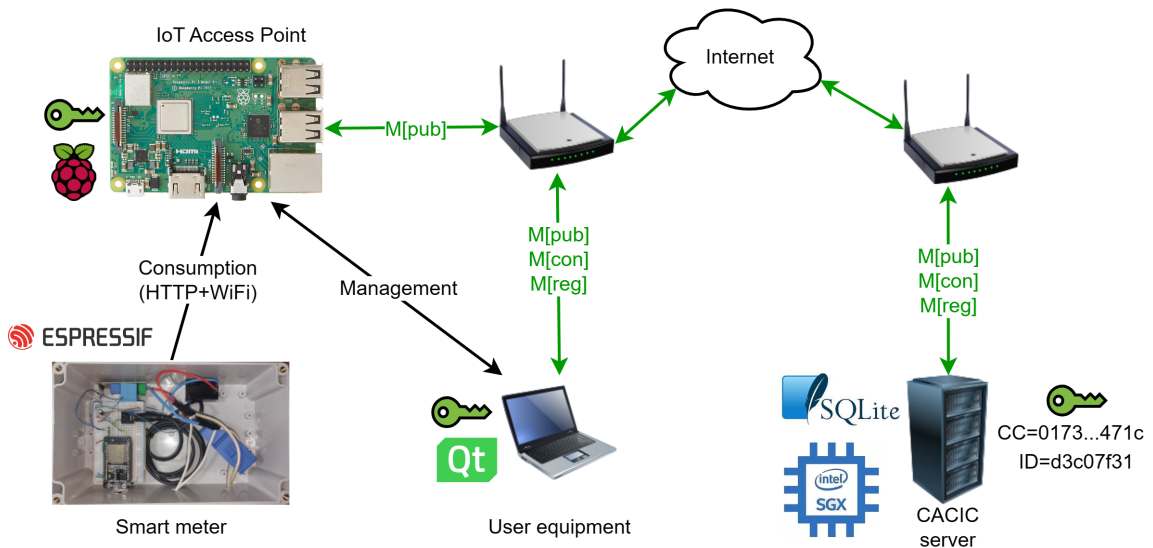


Figure 4.3: The client LAN (left side) communicates with the CACIC server (right side) through the Internet, using CACIC messages, in green. The green key represents the client CK, shared with the server. Both the access point and the user equipment use it to encrypt the messages.

ment application. Through the interface, the client selects a public ID and a secret CK, and shares this information with the server’s enclave and the access point. The ID and CK can be generated by hashing a username and a password, respectively.

The security of the key generation process is not within the scope of this work. Then, the client uses the application to select access permissions for each type of publication message to be sent (Figure 4.2b). The use case implementation defines two types of data publication requests: `publication_consumption_measured` and `publication_consumption_aggregated`. Both the client application and the access point store a dictionary, associating each type of publication message with a list of access permissions. The storage format for access permissions, ID, and CK can vary for different use case implementations.

From now on, the access point can receive samples from the sensor and use the `client_publish()` function (Section 4.1) to forward a message of type `publication_consumption_measured` to the server. There is no processing task associated with this message, so the server simply inserts the encrypted data into the database. Next, the client application can send a `publication_consumption_aggregated` (Figure 4.2c) to the server to calculate the total energy consumption and provide the result to an institution, for example. In this case, the developer's processing task function running inside the server's enclave uses the `enclave_multi_query_db()` function (Section 4.1) to retrieve previous data of type `publication_consumption_measured` and add them up. If another client sends a `publication_consumption_aggregated`, the server refuses to perform aggregations with data for which access has not been granted by the data owner. The application also provides an interface to send a data query message, allowing the filtering of parameters such as types and IDs. Figure 4.2d depicts a user application screen displaying the result of a valid query, where only the data accessible to the user is returned by the server's enclave. The application allows exporting the data in Comma Separated Values (CSV) format for processing in other software applications. The demonstration video showcases installation instructions, other screens of the application, and the testing environment featuring the described equipment. Figure 4.4 shows a print screen of the server logs generated while processing a M[publication] of type `publication_consumption_aggregated`.

Chapter 5

Conclusions

This undergraduate project proposes, implements, and evaluates an architecture to process IoT data in clouds using enclaves. This work differs from others by considering a scenario where an attacker has almost complete control over the server. Therefore, it combines traditional cryptography solutions with trusted execution environments, which are state-of-the-art solutions to protect computations in untrusted parties. The goal is to allow clients to customize who can access their data, even in a scenario where a malicious superuser inside the cloud infrastructure wants to steal clients' data to obtain financial advantages, for example. The performance analysis revealed that the enclave adds only 0.1 ms to the latency for publishing and querying data and that the system processes thousands of requests per second, without introducing a significant CPU and memory overhead.

This work also implements, demonstrates, and documents CACIC-DevKit, a tool for developing generic IoT systems using the proposed architecture. The goals are to simplify the development of IoT systems using enclaves and to confirm that the CACIC architecture is compatible with commercial sensors, interfaces, and databases. For this, an energy consumption data management use case was implemented. The system was demonstrated using a sensor, an access point, and a data management application in a real testbed.

In future work, the architecture must be extended to protect the data within sensors and access points. The system could be equipped with a feature for clients to publish their own data processing codes to be protected in the cloud enclave. The architecture must be extended to allow more complex access control rules based on

expiration timestamps or restrictions for data usage, for example. The performance evaluation must also be extended to analyze the use of enclaves for machine learning, deep learning, and federated learning within CACIC, given the relevance of these technologies in cloud applications. These enclave performance measurements are relevant since they help the research community optimize trusted computing systems and applications. Future performance analysis can use specialized tools for microbenchmarking, in order to identify the bottlenecks and find improvement opportunities [33]. Therefore, another promising future research direction is the implementation of distributed processing and storage architectures using enclaves.

Some researchers in the GTA team consider using CACIC-DevKit in future projects for protecting and processing sensitive data concerning vehicular networks and private machine learning models. These new potential use cases will help the repository and the documentation to evolve.

Bibliography

- [1] LEE, I., LEE, K., “The Internet of Things (IoT): Applications, investments, and challenges for enterprises”, *Business Horizons*, v. 58, n. 4, pp. 431–440, 2015.
- [2] ORTIZ, F. M., SAMMARCO, M., COSTA, L. H. M. K., *et al.*, “Applications and Services Using Vehicular Exteroceptive Sensors: A Survey”, *IEEE Transactions on Intelligent Vehicles*, v. 8, n. 1, pp. 949–969, 2023.
- [3] GANTERT, L., SAMMARCO, M., DETYNIECKI, M., *et al.*, “A Supervised Approach for Corrective Maintenance Using Spectral Features from Industrial Sounds”. In: *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, pp. 723–728, 2021.
- [4] OTHMAN, M. M., EL-MOUSA, A., “Internet of Things & Cloud Computing Internet of Things as a Service Approach”. In: *2020 11th International Conference on Information and Communication Systems (ICICS)*, pp. 318–323, 2020.
- [5] FERNANDES, E., JUNG, J., PRAKASH, A., “Security Analysis of Emerging Smart Home Applications”. In: *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 636–654, 2016.
- [6] EIBL, G., ENGEL, D., “Influence of Data Granularity on Nonintrusive Appliance Load Monitoring”. *IH&MMSec '14*, p. 147–151, New York, NY, USA, 2014.
- [7] PEARSON, S., “Trusted computing platforms, the next security solution”, *HP Labs*, v. 177, 2002.

- [8] SHULTZ, D., “When your voice betrays you”, *Science*, v. 347, n. 6221, pp. 494–494, 2015.
- [9] SHOKRI, R., SHMATIKOV, V., “Privacy-Preserving Deep Learning”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS ’15, p. 1310–1321, New York, NY, USA, 2015.
- [10] COSTAN, V., DEVADAS, S., “Intel SGX explained”, *Cryptology ePrint Archive*, , 2016.
- [11] THOMAZ, G. A., GUERRA, M. B., SAMMARCO, M., *et al.*, “CACIC: Controle de Acesso Confiável Usando Enclaves a Dados em Nuvem da Internet das Coisas”. In: *Anais do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pp. 573–586, SBC, 2022.
- [12] THOMAZ, G. A., GUERRA, M. B., SAMMARCO, M., *et al.*, “Tamper-proof access control for IoT clouds using enclaves”, *Ad Hoc Networks*, v. 147, pp. 103191, 2023.
- [13] THOMAZ, G., GUERRA, M., SAMMARCO, M., *et al.*, “CACIC-DevKit: Construção de Sistemas IoT com Políticas de Acesso Customizáveis e Segurança por Hardware”. In: *Anais Estendidos do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pp. 1–8, Porto Alegre, RS, Brasil, 2023.
- [14] YANG, Q., WANG, H., WU, X., *et al.*, “Secure Blockchain Platform for Industrial IoT with Trusted Computing Hardware”, *IEEE Internet of Things Magazine*, v. 4, n. 4, pp. 86–92, 2021.
- [15] VALADARES, D. C. G., DA SILVA, M. S. L., BRITO, A. E. M., *et al.*, “Achieving Data Dissemination with Security using FIWARE and Intel Software Guard Extensions (SGX)”. In: *2018 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1–7, 2018.
- [16] LI, S., XUE, K., WEI, D. S. L., *et al.*, “SecGrid: A Secure and Efficient SGX-Enabled Smart Grid System With Rich Functionalities”, *IEEE Transactions on Information Forensics and Security*, v. 15, pp. 1318–1330, 2020.

- [17] PRIEBE, C., VASWANI, K., COSTA, M., “EnclaveDB: A Secure Database Using SGX”. In: *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 264–278, 2018.
- [18] ANCIAUX, N., BONNET, P., BOUGANIM, L., *et al.*, “Personal Data Management Systems: The security and functionality standpoint”, *Information Systems*, v. 80, pp. 13–35, 2019.
- [19] XIAO, Y., ZHANG, N., LI, J., *et al.*, “Privacyguard: Enforcing private data usage control with blockchain and attested off-chain contract execution”. In: *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part II 25*, pp. 610–629, Springer, 2020.
- [20] LI, S., XUE, K., WEI, D. S., *et al.*, “SecGrid: A secure and efficient SGX-enabled smart grid system with rich functionalities”. In: *IEEE Transactions on Information Forensics and Security*, v. 15, pp. 1318–1330, 2019.
- [21] SILVA, L. V., BARBOSA, P., MARINHO, R., *et al.*, “Security and privacy aware data aggregation on cloud computing”, 2018.
- [22] AYOADE, G., EL-GHAMRY, A., KARANDE, V., *et al.*, “Secure data processing for IoT middleware systems”, 2019.
- [23] CARPENTIER, R., THIANT, F., POPA, I. S., *et al.*, “An Extensive and Secure Personal Data Management System using SGX”. In: *25th International Conference on Extending Database Technology*, 2022.
- [24] PACHECO, R. G., COUTO, R. S., SIMEONE, O., “On the impact of deep neural network calibration on adaptive edge offloading for image classification”, *Journal of Network and Computer Applications*, p. 103679, 2023.
- [25] DUNCAN, A. J., CREESE, S., GOLDSMITH, M., “Insider attacks in cloud computing”. In: *2012 IEEE 11th international conference on trust, security and privacy in computing and communications*, pp. 857–862, IEEE, 2012.
- [26] RONG, C., NGUYEN, S. T., JAATUN, M. G., “Beyond lightning: A survey on security challenges in cloud computing”, 2013, Special issue on Recent

Advanced Technologies and Theories for Grid and Cloud Computing and Bio-engineering.

- [27] ZEGZHDA, D. P., USOV, E., NIKOL'SKII, A., *et al.*, “Use of Intel SGX to ensure the confidentiality of data of cloud users”, 2017.
- [28] GUIMARÃES, L. C., REBELLO, G. A. F., CAMILO, G. F., *et al.*, “A threat monitoring system for intelligent data analytics of network traffic”. In: *Annals of Telecommunications*, pp. 1–16, 2021.
- [29] NGABONZIZA, B., MARTIN, D., BAILEY, A., *et al.*, “Trustzone explained: Architectural features and use cases”. In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pp. 445–451, IEEE, 2016.
- [30] JOHNSON, S., SCARLATA, V., ROZAS, C., *et al.*, “Intel software guard extensions: EPID provisioning and attestation services”, *White Paper*, v. 1, n. 1-10, pp. 119, 2016.
- [31] SCARLATA, V., JOHNSON, S., BEANEY, J., *et al.*, “Supporting third party attestation for Intel SGX with Intel data center attestation primitives”, *White paper*, , 2018.
- [32] HAAKEGAARD, R., LANG, J., “The elliptic curve diffie-hellman (ecdh)”, *Online at <https://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Haakegaard+Lang.pdf>*, , 2015.
- [33] WEICHBRODT, N., AUBLIN, P.-L., KAPITZA, R., “sgx-perf: A performance analysis tool for intel sgx enclaves”. In: *Proceedings of the 19th International Middleware Conference*, pp. 201–213, 2018.
- [34] SILVA, R., BARBOSA, P., BRITO, A., “Dynsgx: A privacy preserving toolset for dynamically loading functions into intel (r) sgx enclaves”. In: *2017 IEEE International Conference on Cloud Computing Technology and Science (Cloud-Com)*, pp. 314–321, IEEE, 2017.

- [35] ZHANG, Y., ZHAO, M., LI, T., *et al.*, “Survey of Attacks and Defenses against SGX”. In: *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*, pp. 1492–1496, IEEE, 2020.
- [36] BRASSER, F., MÜLLER, U., DMITRIENKO, A., *et al.*, “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *Proceedings of the 11th USENIX Conference on Offensive Technologies, WOOT’17*, p. 11, USA, 2017.
- [37] NILSSON, A., BIDEH, P. N., BRORSSON, J., “A Survey of Published Attacks on Intel SGX”, 2020.
- [38] HADDADPAJOUH, H., DEGHANTANHA, A., PARIZI, R. M., *et al.*, “A survey on internet of things security: Requirements, challenges, and solutions”. In: *Internet of Things*, v. 14, p. 100129, 2021.
- [39] KARJOTH, G., SCHUNTER, M., WAIDNER, M., “Privacy-enabled services for enterprises”. In: *Proceedings. 13th International Workshop on Database and Expert Systems Applications*, pp. 483–487, IEEE, 2002.
- [40] SICARI, S., RIZZARDI, A., DINI, G., *et al.*, “Attribute-based encryption and sticky policies for data access control in a smart home scenario: a comparison on networked smart object middleware”, *International Journal of Information Security*, v. 20, n. 5, pp. 695–713, 2021.
- [41] SUBRAMANYAN, P., SINHA, R., LEBEDEV, I., *et al.*, “A formal foundation for secure remote execution of enclaves”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2435–2450, 2017.
- [42] HOU, L., ZHAO, S., XIONG, X., *et al.*, “Internet of things cloud: Architecture and implementation”, 2016.
- [43] Telefónica I+D official Open Source repositories, “FIWARE-IOTAGENT-UL”, <https://github.com/telefonicaid/iotagent-ul>, 2016–2019.
- [44] ALONSO, R. S., SITTON-CANDANEDO, I., GARCIA, O., *et al.*, “An intelligent Edge-IoT platform for monitoring livestock and crops in a dairy farming scenario”, 2020.

- [45] ARAUJO, V., MITRA, K., SAGUNA, S., *et al.*, “Performance evaluation of FIWARE: A cloud-based IoT platform for smart cities”. In: *Journal of Parallel and Distributed Computing*, v. 132, pp. 250–261, 2019.
- [46] Gil Tene, “wrk2: a HTTP benchmarking tool based mostly on wrk”, <https://github.com/giltene/wrk2>, 2015–2019.
- [47] WEISSE, O., BERTACCO, V., AUSTIN, T., “Regaining lost cycles with Hot-Calls: A fast interface for SGX secure enclaves”, 2017.
- [48] ZHOU, X., JIN, Y., ZHANG, H., *et al.*, “A map of threats to validity of systematic literature reviews in software engineering”. In: *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pp. 153–160, IEEE, 2016.