Desempenho de Algoritmos de *Hashing* e de Cifragem em Dispositivos IoT

Mestrado em Cibersegurança e Informática Forense

Pedro André Simões dos Santos

Leiria, setembro de 2023

Desempenho de Algoritmos de *Hashing* e de Cifragem em Dispositivos IoT

Mestrado em Cibersegurança e Informática Forense

Pedro André Simões dos Santos

Trabalho de Projeto realizado sob a orientação do Professor Doutor Luís Alexandre Lopes Frazão e do Professor Doutor Daniel Alexander Lopes Fuentes

Leiria, setembro de 2023

Originalidade e Direitos de Autor

O presente trabalho de projeto é original, elaborado unicamente para este fim, tendo sido devidamente citados todos os autores cujos estudos e publicações contribuíram para o elaborar.

Reproduções parciais deste documento serão autorizadas na condição de que seja mencionado o Autor e feita referência ao ciclo de estudos no âmbito do qual o mesmo foi realizado, a saber, Curso de Mestrado em Cibersegurança e Informática Forense, no ano letivo 2022/2023, da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria, Portugal, e, bem assim, à data das provas públicas que visaram a avaliação destes trabalhos.

Agradecimentos

Em primeiro lugar quero agradecer ao Professor Doutor Luís Alexandre Lopes Frazão e ao Professor Doutor Daniel Alexander Lopes Fuentes por nunca terem desistido de me ajudar na realização deste trabalho de projeto, apresentando a disponibilidade necessária e preciosa para ultrapassar qualquer adversidade que surgiu durante toda esta jornada.

Agradecer também ao Professor Doutor Mário João Gonçalves Antunes por me ter mostrado o mundo da cibersegurança, incentivado a inscrever-me neste mestrado e por também mostrar sempre disponibilidade e ajuda quando necessário, assim como motivação para a finalização do presente documento.

Por fim, agradecer à minha família e amigos mais próximos que todos os dias se encarregaram de me dar uma palavra de motivação e força, quando estas eram necessárias.

Resumo

O presente trabalho de projeto tem como objetivo realizar um estudo sobre o desempenho de algoritmos de *hashing* e de cifragem num ambiente IoT. Para tal definiram-se alguns parâmetros de medição considerados cruciais, como a utilização do processador, utilização de memória, latência e taxa de transferência.

Foi realizado um estudo teórico que explica conceitos de segurança em IoT, assim como a sua arquitetura, os protocolos de comunicação IoT mais utilizados, o funcionamento e arquitetura dos protocolos CoAP e o MQTT, e é feita uma explicação dos dois tipos de algoritmos em análise. São também detalhados os vários algoritmos de *hashing* e de cifragem utilizados para a realização dos testes, nomeadamente MD5, SHA-256, Blake2 e RIPEMD-160 para os algoritmos de *hashing*, e AES, RC4, DES e Blowfish para os algoritmos de cifragem.

É apresentada a metodologia seguida para a realização dos testes e apresentado o ambiente IoT simulado com recurso a um Raspberry Pi 3. Foram desenvolvidos *scripts* que permitiram a medição e envio de mensagens com vários tamanhos e em várias quantidades, por teste, para um broker MQTT. É também detalhado o *script* padrão e todas as alterações necessárias para a implementação de cada algoritmo.

Este trabalho originou um conjunto de resultados que são apresentados no documento e que permitem perceber quais os algoritmos que são vantajosos em diversos cenários IoT, principalmente para dispositivos de fracos recursos.

Palavras-chave: *Hashing*, Cifragem, IoT, MQTT, Desempenho

Abstract

The aim of this project is to study the performance of hashing and encryption algorithms in an IoT environment. To do this, it was decided to measure parameters such as CPU utilization, memory utilization, latency and transfer rate.

A theoretical study was carried out explaining IoT security concepts, as well as its architecture, the most used IoT communication protocols, such as the operation and architecture of the CoAP and MQTT protocols, and an explanation of the two types of algorithms under analysis. The various hashing and encryption algorithms used to carry out the tests are also detailed, namely MD5, SHA-256, Blake2 and RIPEMD-160 for the hashing algorithms, and AES, RC4, DES and Blowfish for the encryption algorithms.

The methodology used to carry out the tests is presented, as well as the simulated IoT environment using a Raspberry Pi 3. Scripts were developed to measure and send messages of various sizes and in various quantities, per test, to an MQTT broker. The standard script and all the changes needed to implement each algorithm are explained.

This work has produced a set of results, which allow us to see which algorithms are advantageous in various IoT scenarios, especially for low-resource devices.

Keywords: Hashing, Encryption, IoT, MQTT, Performance

Índice

Originalidade e Direitos de Autor ... **iii**

Agradecimentos ... **iv**

Resumo ... **v**

Abstract ... **vi**

Lista de Figuras ... **ix**

Lista de Tabelas ... **xii**

Lista de Siglas e Acrónimos ... **xiii**

1. Introdução ... **1**

1.1. Objetivos da pesquisa ... **1**

1.2. Organização do Projeto ... **2**

2. Trabalho relacionado ... **4**

2.1. Conceitos de segurança em IoT ... **4**

2.2. Protocolos de comunicação IoT ... **7**

2.2.1. Protocolo MQTT ... 9
2.2.2. Protocolo CoAP ... 12
2.2.3. Outros Protocolos ... 15

2.3. Algoritmos de *hashing* ... **16**

2.3.1. Função de hashing ... 17
2.3.2. Algoritmos de *hashing* em análise ... 18

2.4. Algoritmos de cifragem ... **24**

2.4.1. Evolução de algoritmos de cifragem ... 25
2.4.2. Algoritmos de cifragem em análise ... 26

3. Metodologia e algoritmos em análise ... **30**

3.1. Descrição da abordagem experimental ... **30**

3.2. Ambiente de testes ... **31**

3.2.1. Arquitetura do ambiente de testes ... 32
3.2.2. Configurações Raspberry Pi ... 32
3.2.3. Configurações Máquina Virtual 1 (*broker* MQTT) ... 33
3.2.4. Configurações Máquina Virtual 2 (*subscriber* MQTT) ... 34

3.3. Definição dos Parâmetros de Medição ... **34**

3.3.1. Utilização de CPU ... 35
3.3.2. Utilização de Memória ... 36
3.3.3. Latência ... 36
3.3.4. Taxa de Transferência ... 37

3.4. Implementação dos *scripts* de envio de mensagens ... 38
3.4.1. Função gravar_resultados ... 39
3.4.2. Função *main* ... 40
3.4.3. Funções de geração de mensagens aleatórias ... 41

3.5. Implementação dos algoritmos no *script* ... 41
3.5.1. MD5 ... 41
3.5.2. SHA-256 ... 43
3.5.3. Blake2 ... 44
3.5.4. RIPEMD-160 ... 44
3.5.5. AES ... 45
3.5.6. RC4 ... 47
3.5.7. DES ... 48
3.5.8. Blowfish ... 50

4. Resultados obtidos ... 52

4.1. Resultados de utilização de CPU e utilização de memória ... 52
4.1.1. Comparação entre algoritmos de *hashing* e de cifragem ... 56
4.1.2. Recomendações para cenários específicos ... 58

4.2. Resultados de Latência ... 59
4.2.1. Discussão dos resultados ... 62
4.2.2. Recomendações para cenários específicos ... 63

4.3. Resultados de Taxa de Transferência ... 63
4.3.1. Discussão dos resultados ... 65
4.3.2. Recomendações para cenários específicos ... 66

4.4. Identificação de Tendências e Padrões ... 66

4.5. Análise crítica dos Cenários de Teste e Resultados ... 67

5. Conclusões e trabalho futuro ... 68

Referências Bibliográficas ... 70

Lista de Figuras

Figura 1 - Arquitetura de ambiente IoT 5

Figura 2 - Arquitetura *Publish / Subscriber* 7

Figura 3 - Arquitetura Request / Response 8

Figura 4 – Arquitetura *Publish / Subscriber* para MQTT 9

Figura 5 - Formato de mensagem (MQTT) 10

Figura 6 - Modelo de reconhecimento de ações (MQTT) 11

Figura 7 - Modelo 3-way handshake (TCP) 11

Figura 8 - Arquitetura *Response/Request* com *token* e resposta imediata 12

Figura 9 - Arquitetura *Response/Request* com *token* e resposta não imediata 13

Figura 10 - Diferença entre Arquitetura dos protocolos MQTT e CoAP 14

Figura 11 - Formato de mensagem (CoAP) 14

Figura 12 - Geração de um *hash* 17

Figura 13 - MD5 Collision 20

Figura 14 - Diferentes velocidades de diferentes algoritmos de *hashing* 22

Figura 15 - Performances de vários algoritmos de *hashing* (CPU) 23

Figura 16 - Esquema de como funciona uma cifragem 24

Figura 17 - Versão modificada de algoritmo de cifragem 26

Figura 18 - Raspberry Pi 3 utilizado 31

Figura 19 - Arquitetura do ambiente de testes 32

Figura 20 - Pasta com todos os *users* permitidos para comunicar com o *broker* MQTT 34

Figura 21 - Bibliotecas do *script* de recolha de dados 35

Figura 22 - Função medicao_CPU_antes 36

Figura 23 - Função medicao_CPU_depois 36

Figura 24 - Função medicao_memoria 36

Figura 25 - Função medicao_latencia 37

Figura 26 - Função medicao_taxa_transferencia 38

Figura 27 - Subscrição de tópico do *broker* MQTT 39

Figura 28 - Receção das mensagens por parte do *subscriber*......39

Figura 29 - Ficheiro medicoesdefault.txt......39

Figura 30 - Função gravar_resultados......40

Figura 31 - Função *main*......40

Figura 32 - Funções de geração de mensagens aleatórias......41

Figura 33 - Bibliotecas adicionais MD5 e função md5_hash......42

Figura 34 - Alteração função medicao_latencia (MD5)......42

Figura 35 - Alteração função gerar_mensagens (MD5)......43

Figura 36 - Função sha256_hash......43

Figura 37 - Alterações função medicao_latencia (SHA-256)......43

Figura 38 - Alterações função gerar_mensagens (SHA-256)......43

Figura 39 - Função blake2_hash......44

Figura 40 - Alterações função medicao_latencia (Blake2)......44

Figura 41 - Alterações gerar_mensagens (Blake2)......44

Figura 42 - Função ripemd_hash......45

Figura 43 - Alterações medicao_latencia (RIPEMD-160)......45

Figura 44 - Alterações função gerar_mensagens (RIPEMD-160)......45

Figura 45 - Bibliotecas e função pad (AES)......46

Figura 46 - Função aes_encrypt (AES)......46

Figura 47 - Alterações função medicao_taxa_transferencia (AES)......47

Figura 48 - Alterações função gerar_mensagens (AES)......47

Figura 49 – Bibliotecas e função pad (RC4)......48

Figura 50 - Função rc4_encrypt (RC4)......48

Figura 51 - Alterações função medicao_taxa_transferencia (RC4)......48

Figura 52 - Alteração função gerar_mensagens /RC4)......48

Figura 53 - Função pad (DES)......49

Figura 54 - Função des_encrypt (DES)......49

Figura 55 - Alterações função medicao_taxa_transferencia (DES)......50

Figura 56 - Função gerar_mensagens (DES)......50

Figura 57 - Função pad (Blowfish)......50

Figura 58 - Função blowfish_encrypt (Blowfish) 51

Figura 59 - Alterações função medicao_taxa_transferencia (Blowfish)........ 51

Figura 60 - Função gerar_mensagens (Blowfish)........ 51

Figura 61 - Utilização de CPU e utilização de memória no envio de 10 mensagens de 100 kb........ 53

Figura 62 - Utilização de CPU e utilização de Memória no envio de 10 mensagens de 1 Mb........ 54

Figura 63 - Utilização de CPU e utilização de Memória no envio de 100 mensagens de 100 kb 55

Figura 64 – Utilização de CPU e utilização de Memória no envio de 100 mensagens com 1 Mb........ 56

Figura 65 - Latência por mensagem (algoritmos de *hashing*) 60

Figura 66 - Latência no envio de 10 mensagens com 100 kb (algoritmos de cifragem) 60

Figura 67 - Latência no envio de 10 mensagens com 1 Mb (algoritmos de cifragem)........ 61

Figura 68 - Latência no envio de 100 mensagens com 100 kb (algoritmos de cifragem) 61

Figura 69 - Latência no envio de 100 mensagens com 1 Mb (algoritmos de cifragem)........ 62

Figura 70 - Taxa de Transferência no envio de 10 mensagens com 100 kb (algoritmos de cifragem)........ 64

Figura 71 - Taxa de Transferência no envio de 10 mensagens com 1 Mb (algoritmos de cifragem)........ 64

Figura 72 - Taxa de Transferência no envio de 100 mensagens com 100 kb (algoritmos de cifragem)........ 65

Figura 73 - Taxa de Transferência no envio de 100 mensagens com 1 Mb (algoritmos de cifragem)........ 65

Lista de Tabelas

Tabela 1 - Cenários de teste 52

Tabela 2 - Melhor desempenho por cenário (parâmetro Utilização de CPU) 58

Tabela 3 - Melhor desempenho por cenário (parâmetro Utilização de Memória) 58

Tabela 4 - Melhor desempenho por cenário (parâmetro Latência) 63

Tabela 5 - Melhor desempenho por cenário (parâmetro Taxa de Transferência) 66

Lista de Siglas e Acrónimos

AES	Advanced Encryption Standard
AMQP	Advanced Message Queuing Protocol
ASCII	American Standard Code for Information Interchange
CIA	Confidentiality, Integrity e Availability
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
DES	Data Encryption Standard
DoS	Denial of Service
DTLS	Datagram Transport Layer Security
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IoT	Internet of Things
M2M	Machine-to-Machine
MD5	Message-Digest algorithm 5
MQTT	Message Queuing Telemetry Transport
NIST	National Institute of Standards and Technology
RAM	Random Access Memory
RC4	Rivest Cipher 4
REST	Represental State Transfer
RIPEMD	RACE Integrity Primitives Evaluation Message Digest
SHA	Secure Hash Algorithm
SQL	Structured Query Language
TCP	Transport Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
XMPP	Extensible Messaging and Presence Protocol

1. Introdução

A segurança e integridade dos dados são preocupações cruciais em ambientes de comunicação e troca de informações sensíveis. Com o rápido crescimento das aplicações IoT (*Internet of Things*, Internet das Coisas em português) e a necessidade de proteger os dados que fazem parte do ambiente IoT, a escolha de algoritmos de *hashing* e de cifragem adequados torna-se essencial. Nesse contexto, a evolução contínua desses algoritmos tem um impacto direto em parâmetros de medição, como a utilização do processador, a utilização da memória, as taxas de transferência e a latência das comunicações.

1.1. Objetivos da pesquisa

Este documento foi escrito no âmbito do Mestrado em Cibersegurança e Informática Forense do Instituto Politécnico de Leiria.

Os seus objetivos são: apresentar ao leitor alguns protocolos para IoT que existem, dedicando os testes ao protocolo MQTT; dar a conhecer alguns algoritmos de *hashing* e de cifragem que podem ser aplicados aquando do envio de mensagens; desenvolver *software* genérico de testes que implemente estes mesmos algoritmos; recolher parâmetros de desempenho de dispositivo como a utilização de CPU (*Central Processing Unit*), a utilização de memória, a taxa de transferência e a latência; e realizar uma análise baseada numa comparação dos resultados obtidos, cuja finalidade é descobrir qual é o algoritmo que melhor se adequa aos vários tipos de cenários reais.

Este estudo será conduzido num ambiente experimental composto por um Raspberry Pi 3 e duas máquinas virtuais, uma como *broker* MQTT e outra como subscritor, de modo a simular um sistema de envio de mensagens típico de um ambiente IoT.

Será realizado um levantamento dos parâmetros de medição necessários, visando avaliar o desempenho e os efeitos desses algoritmos relativamente à utilização do CPU, utilização da memória, da taxa de transferência e da latência nas comunicações, através de testes em dispositivos IoT. Para a realização desses testes e medições será desenvolvido *software* capaz de medir os parâmetros supracitados em vários cenários diferentes, que diferem em

dois parâmetros principais: o número de mensagens enviadas e o tamanho das mensagens enviadas.

A compreensão aprofundada dessas relações permitirá identificar possíveis explicações para o impacto dos algoritmos de *hashing* e de cifragem através dos parâmetros de medição definidos, contribuindo para o desenvolvimento de estratégias mais eficientes e seguras no âmbito da cibersegurança em ambientes IoT.

1.2.Organização do Projeto

Este trabalho de projeto de mestrado apresentado está organizado em 5 capítulos, os quais descrevem todo o trabalho de investigação, desenvolvimento e testes efetuados, assim como as respetivas análises e conclusões dos resultados obtidos nos testes, sendo o atual o capítulo introdutório.

O segundo capítulo apresenta uma fundamentação teórica, onde são apresentados os conceitos fundamentais de cibersegurança relacionados com o projeto em questão, bem como uma explicação detalhada sobre o protocolo MQTT, utilizado no ambiente experimental. Além disso, são descritos os algoritmos de *hashing* e de cifragem utilizados, fornecendo a base teórica necessária para a compreensão da pesquisa e são abordadas as características e princípios de funcionamento de cada um.

No terceiro capítulo é apresentada a metodologia adotada na elaboração deste estudo. São descritas as etapas da implementação dos *scripts* de envio de mensagens para o *broker* MQTT. É também apresentado o ambiente de testes utilizado (Raspberry Pi 3 e máquinas virtuais). São apresentados os critérios de medição e a abordagem experimental. Além disso, também é apresentado todo o desenvolvimento realizado a nível de *software* para a realização dos testes idealizados.

No quarto capítulo, seguem-se os resultados obtidos que são apresentados e analisados. Os dados relacionados aos parâmetros de medição são discutidos relativamente aos algoritmos utilizados, permitindo identificar o impacto específico de cada algoritmo nos parâmetros avaliados. São realizadas comparações entre os algoritmos de *hashing* e de cifragem, procurando identificar padrões e tendências relevantes para o estudo, e, com base nos resultados apresentados, expõe-se uma discussão e interpretação destes.

As conclusões finais da pesquisa, apresentadas no quinto e último capítulo, sintetizam os principais resultados e destacam as contribuições deste projeto. Também são discutidas as limitações do estudo e fornecidas sugestões para trabalhos futuros na área.

2. Trabalho relacionado

No presente capítulo serão apresentados conceitos básicos de IoT e abordados aspetos como a arquitetura, formatos de mensagem e respetivo *overhead*, transporte, segurança, escalabilidade e interoperabilidade de alguns protocolos utilizados em ambiente IoT. Será analisado o que são e como funcionam os algoritmos de *hashing* e de cifragem, e apresentada uma fundamentação teórica para cada um dos algoritmos que vão ser utilizados para a realização dos testes, o que posteriormente levará às conclusões e contribuições deste trabalho de projeto.

2.1. Conceitos de segurança em IoT

Neste tópico são abordados alguns temas introdutórios sobre os dispositivos IoT, como o que são, que ameaças existem em IoT, que soluções para segurança IoT se podem implementar, e o que podem ser tendências e desafios futuros para a segurança em IoT.

O termo dispositivo IoT é atribuído a todos os dispositivos ligados em rede ou à Internet, que não sejam os tradicionais dispositivos com essa ligação, como computadores, telemóveis, servidores, entre outros [1].

Segundo a lei de Moore [2], a cada dois anos, os números de transístores dos *chips* de processamento iriam duplicar ao mesmo custo, fazendo assim com que os *chips* de processamento possam ser mais pequenos, mais rápidos e mais baratos. A verdade é que a evolução tecnológica nas últimas décadas tem sido exponencial, evolução essa que permite cada vez mais o desenvolvimento de novos dispositivos com novas funcionalidades e características, nomeadamente como a capacidade de recolher, processar, guardar e enviar dados para um destino.

Grande parte destes novos dispositivos são designados de dispositivos IoT, ou fazem parte do universo da Internet das Coisas, e alguns exemplos desses dispositivos que estão cada vez mais a emergir são: carros inteligentes, capazes de analisar quais as melhores rotas para reduzir o gasto de combustível ou até mesmo prever a necessidade de manutenção de alguma parte do veículo; casas inteligentes, onde via um simples *tablet* por exemplo, conseguimos acender as luzes da nossa casa, controlar o ar condicionado de uma divisão, ou até mesmo abrir uma garagem.

Tudo isto, assenta sobre uma pequena arquitetura de funcionamento, composta por três partes:

- **Dispositivo Inteligente**: por exemplo, uma televisão com recursos de computação.
- **Aplicação IoT**: conjunto de serviços e software que integra os dados recebidos pelos dispositivos inteligentes.
- **Interface Gráfica de Utilizador**: um dispositivo IoT que serve para todos os dispositivos inteligentes ligados à aplicação IoT, poderem ser controlados e geridos.

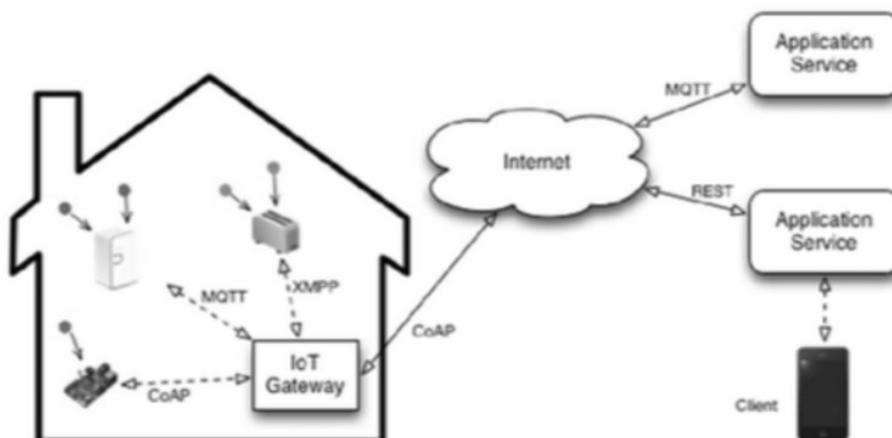


Figura 1 - Arquitetura de ambiente IoT

A Figura 1 demonstra um ambiente IoT doméstico, onde existem os dispositivos inteligentes que comunicam por protocolos de comunicação para IoT (CoAP, MQTT e XMPP) com um IoT Gateway, que juntamente com o serviço aplicacional (*Application Service*) vão formar a Aplicação IoT em si, e que depois são disponíveis para o cliente através de uma Interface Gráfica de Utilizador [3].

Embora o contexto IoT tenha trazido inúmeras inovações e conveniências para o nosso dia a dia, apresenta também desafios de segurança informática. As ameaças associadas à IoT são variadas e podem comprometer a confidencialidade, integridade e disponibilidade dos dispositivos que atuam neste ambiente.

A Karspersky, realizou um estudo [4] e afirmou que entre janeiro e junho de 2021 ocorreram cerca de 1.51 mil milhões de incidentes relacionados com dispositivos IoT, e a tendência para este número aumentar é acompanhada, relacionalmente, do aumento do número de dispositivos IoT ligados à Internet ao longo dos anos, pois, mais dispositivos, mais

possibilidades de ataques. Algumas ameaças em IoT provém de vulnerabilidades aproveitadas por ataques informáticos.

Alguns exemplos de vulnerabilidades são o *SQL* (*Structured Query Language*) *Injection*, que permite a injeção de comandos SQL que podem comprometer a segurança do sistema, ou o DoS (*Denial of Service*), onde é causada uma interrupção de serviço que pode afetar a disponibilidade da rede. Ora, os ataques informáticos podem visar a exploração de vulnerabilidades em dispositivos para assumir o controlo desses mesmos dispositivos, roubar informações ou realizar outras atividades maliciosas [5].

Para garantir que os dispositivos e os dados que neles interagem estejam protegidos contra as ameaças supracitadas, e muitas outras existentes, é necessário arranjar soluções para mitigar o impacto que possa resultar de eventuais ataques. A seguir, são apresentadas algumas das principais estratégias para reforçar a segurança em IoT.

Algumas soluções para a segurança em IoT (e algumas da quais vão ser exploradas mais adiante neste documento) são:

- **Cifragem:** a cifragem de dados pode prevenir o acesso de dados a entidades não autorizadas durante uma transmissão.
- **Autenticação e controlo de acessos:** os dispositivos devem possuir *passwords* fortes, como a utilização de números, caracteres especiais, ou até mesmo a utilização de um gestor de *passwords*, assim como possuir políticas de controlo de acesso que permitam que apenas os utilizadores autorizados possam aceder aos dados transmitidos.
- **Atualizações constantes do *Firmware* e do *Software*:** o *firmware* e o *software* devem ser atualizados sempre que possível para proteger os dispositivos contra novas ameaças
- **Monitorização e criação de alertas:** a monitorização e a criação de alertas é essencial para detetar e prevenir a ocorrência de incidentes.
- **Segurança física:** o acesso físico deve ser restrito apenas para pessoal autorizado, e todos os dispositivos IoT devem estar guardados em locais seguros.

2.2.Protocolos de comunicação IoT

Neste tópico, são introduzidos alguns protocolos utilizados em IoT, tentando realizar para cada protocolo uma pequena apresentação, nomeadamente qual a sua arquitetura, formato de mensagem e *overhead*, transporte e segurança, escalabilidade e interoperabilidade, e algumas implementações possíveis.

Existem dois grandes tipos de arquitetura quando se fala em protocolos de IoT, são eles:

- *Publish / Subscribe*
- *Request / Response*

Como é possível observar na Figura 2, a arquitetura *Publish / Subscribe* é uma arquitetura onde há um dispositivo centralizado que vai copiar todas as mensagens que entrem pelo seu canal de entrada (*input channel*), proveniente de dispositivos que publicam essas mesmas mensagens (*publisher*) e enviar todas essas mensagens por um canal de saída (*output channel*) a todos os dispositivos que subscreveram esse canal de saída (*subscribers*) [6].

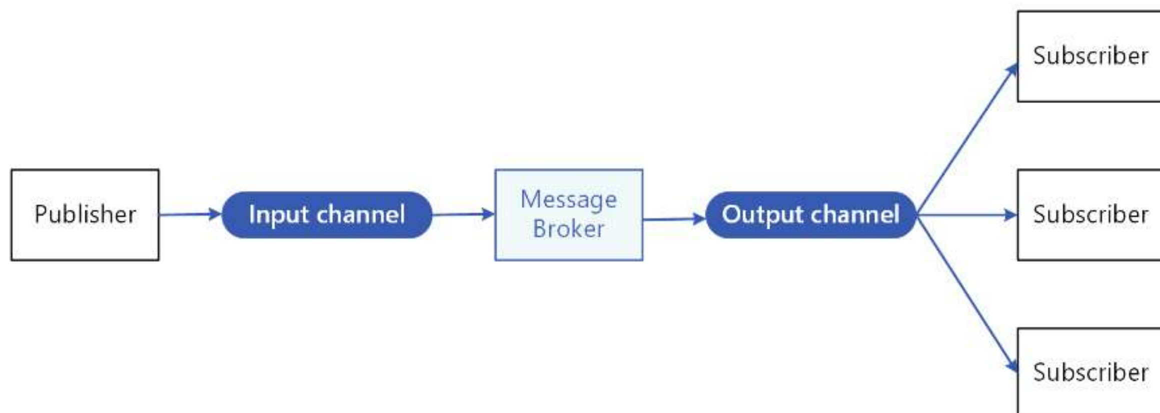


Figura 2 - Arquitetura *Publish / Subscriber*

Existem vários cenários onde esta arquitetura é a ideal para se aplicar, são eles quando [6]:

- Uma aplicação tem de difundir informações para um número significativo de *subscribers*.
- Uma aplicação tem de comunicar com uma ou mais aplicações ou serviços desenvolvidos de forma independente, que podem utilizar diferentes plataformas, linguagens de programação e protocolos de comunicação.
- Uma aplicação pode enviar informações aos *subscribers* sem precisar de respostas em tempo real por parte dos consumidores.

- Os sistemas que estão a ser integrados foram concebidos para suportar um modelo de consistência eventual para os respetivos dados.
- Uma aplicação tem de comunicar informações a vários *subscribers*, que podem ter requisitos de disponibilidade ou agendamentos de tempo de atividade diferentes dos do remetente.

Como é possível observar na Figura 3, a arquitetura *Request / Response* é baseada em pedidos (*requests*) e respostas (*responses*), onde um fluxo típico desta arquitetura passa por um cliente enviar um pedido a um servidor e receber uma resposta [7].

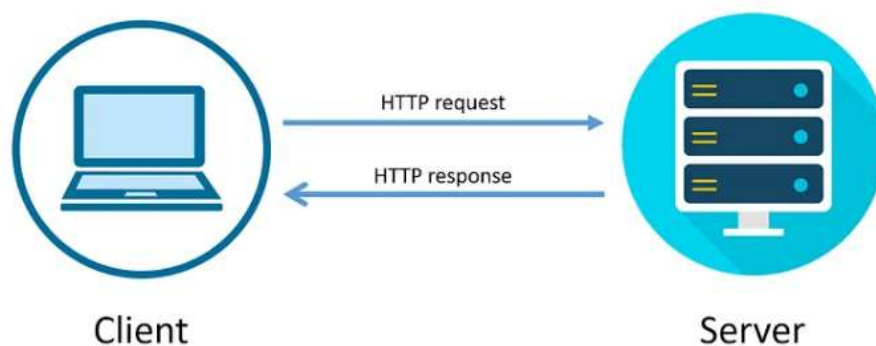


Figura 3 - Arquitetura Request / Response

Existem vários cenários onde esta arquitetura é a ideal para se aplicar, são eles quando [8]:

- As aplicações em utilização estão num browser de Internet (ou seja, as aplicações estão do lado do cliente). Em muitos casos, essas aplicações precisam de comunicar com um servidor para obter ou enviar dados. A arquitetura *request/response* é uma abordagem simples e direta para essa comunicação.
- A única opção de comunicação disponível é o protocolo HTTP. Em alguns ambientes, devido a restrições de segurança como *firewalls*, o servidor não pode iniciar uma comunicação com o cliente (*callback*). Nesses casos, a arquitetura *request/response* é ideal, pois o cliente é sempre quem inicia a comunicação, enviando um pedido ao servidor, esperando por uma resposta.

Nos subcapítulos seguintes, será possível compreender como é que os protocolos de comunicação IoT, o MQTT e o CoAP, se adaptam a estes tipos de arquiteturas.

2.2.1. Protocolo MQTT

O MQTT é um protocolo de troca de mensagens entre dispositivos e redes IoT. É um protocolo extremamente leve, o que é ideal para ligar dispositivos remotos numa rede com alta latência ou baixa largura de banda. O MQTT é atualmente utilizado numa grande variedade de indústrias, tais como a automóvel, telecomunicações, petróleo e gás, entre outras [9]. O protocolo MQTT é amplamente utilizado em cenários de IoT devido à sua eficiência e baixo consumo de energia e adota uma arquitetura baseada em *Publish / Subscribe*.

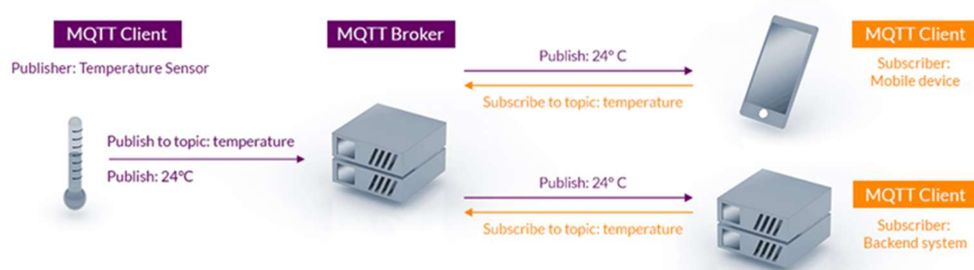


Figura 4 – Arquitetura *Publish / Subscriber* para MQTT

Com base na Figura 4, podemos perceber como funciona este tipo de arquitetura. Existem dois tipos de dispositivos, cliente e *broker*, e dois tipos de intervenientes, *publishers* e *subscribers* [9].

Os clientes MQTT são dispositivos que vão interagir com os dados, que constituem os dois tipos de intervenientes (o *publisher – temperature sensor*; e o *subscriber – mobile device* e *backend system*); o *broker* MQTT é um servidor capaz de receber e enviar todas as mensagens enviadas ou requeridas pelos dispositivos intervenientes.

Na Figura 4 é possível ver um exemplo de um sensor de temperatura (dispositivo *client* e *publisher*), que vai publicar no tópico "Temperature" uma mensagem a indicar a temperatura atual (24°C). O *broker* MQTT (dispositivo *broker*) vai então armazenar nesse tópico a mensagem, a qual será enviada para qualquer dispositivo que tenha subscrito o tópico em questão.

Por último, o telemóvel e um sistema de *backend* (dispositivos *client* e *subscriber*), que subscreveram o tópico "Temperature", vão receber a mensagem que foi enviada/publicada pelo sensor e que foi armazenada pelo *broker* MQTT [10].

Como é possível observar na Figura 5, o formato de uma mensagem que utilize o protocolo MQTT possui 4 partes principais. São elas o *Control Header* que possui 1 *byte* de comprimento e é composto por várias *flags* que vão determinar a qualidade de serviço da mensagem; o *Packet Length*, ou seja, o comprimento do pacote da mensagem que varia entre 1 e 4 *bytes*; o *Variable Length Header*, que é uma parte do cabeçalho variável, e que por isso pode possuir um mínimo de 0 *bytes*; e por fim o *Payload* que pode também possuir um mínimo de 0 *bytes*.

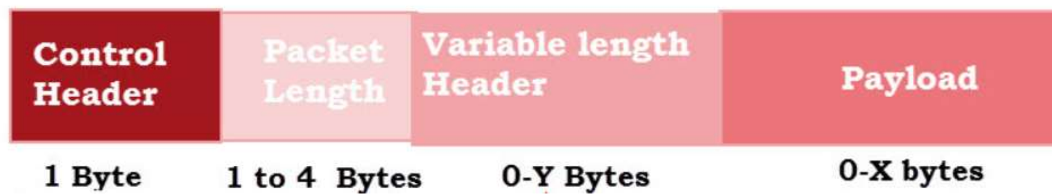


Figura 5 - Formato de mensagem (MQTT)

Sendo assim, o mínimo de *bytes* por mensagem associado ao protocolo MQTT é de 2 *bytes*, como quando, por exemplo, um cliente termina a ligação com o *broker*, onde o *Control Header* possui o 1 *byte* associado, e o *Packet Length* é apenas de 1 *byte* [10].

Devido à utilização de um cabeçalho mais compacto e pacotes de mensagem mais reduzidos, este protocolo é adequado para ligações de baixa largura de banda, uma vez que o seu *overhead* é reduzido [11], ou seja, evita a sobrecarga por parte da mensagem, de modo a provocar um impacto considerável na largura de banda, latência ou consumo de energia.

Por padrão, o protocolo MQTT utiliza como protocolo de comunicação da camada de transporte o TCP (*Transport Control Protocol*), que por norma é mais confiável do que o outro protocolo existente da camada de transporte, o UDP (*User Datagram Protocol*), uma vez que o TCP vai realizar uma confirmação de receção por cada pacote de mensagem que é transmitido.

Como é possível observar na Figura 6, para cada interação entre um cliente, seja ele *publisher* ou *subscriber*, há uma confirmação por parte do *broker* MQTT sob a forma de *acknowledges* (ACKs). Estas confirmações são realizadas quando o cliente MQTT inicia ou termina a ligação com *broker* (*connect*), quando há uma publicação de mensagens no *broker* (*publish*), e até mesmo quando um cliente MQTT subscreve um tópico do *broker* (*subscribe*)

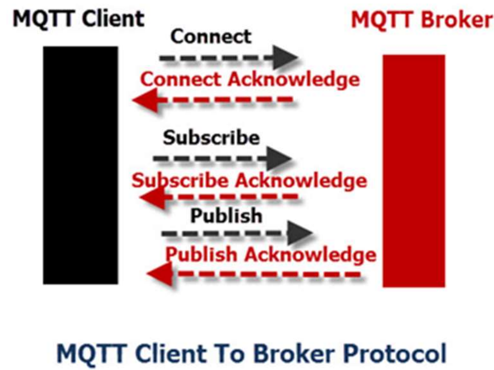


Figura 6 - Modelo de reconhecimento de ações (MQTT)

A Figura 7 mostra que no modelo "*3-way handshake*" existe sempre o reconhecimento (*acknowledge* - ACK) de um pedido de sincronização (*synchronize* – SYN) entre o cliente e o servidor para cada pedido e resposta; reconhecimento esse que também ocorre no modelo do protocolo MQTT na comunicação com o *broker*, anteriormente explicado [10] [12].

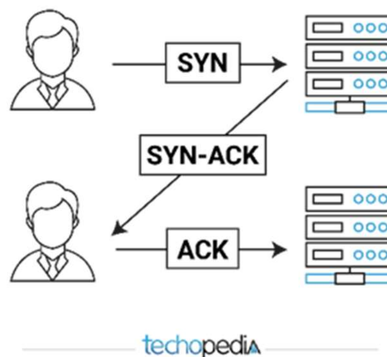


Figura 7 - Modelo 3-way handshake (TCP)

As duas principais implementações de segurança neste protocolo são a utilização do protocolo de segurança TLS (*Transport Layer Security*), que irá cifrar a mensagem enviada, e os mecanismos de autenticação aplicacionais dedicados ao *broker*, como o exemplo do MQTT Lens [13], uma vez que este protocolo, como já referido, é muito baseado em dois tipos de utilizadores (cliente e *broker*).

O protocolo MQTT é considerado um protocolo bastante escalável [9], uma vez que foi concebido para suportar uma abundância de dispositivos e mensagens. Além disso, muitos

dispositivos IoT mantem sessões persistentes, o que requer uma diminuição do tempo de uma nova ligação e o aumento do tempo de ligação dos dispositivos, sendo estes dois grandes fatores para um protocolo IoT escalável (número de dispositivos e tempo de ligação por dispositivo) e o facto de ser um protocolo *open-source*, faz com que a sua interoperabilidade com vários outros tipos de sistemas e dispositivos seja possível.

2.2.2. Protocolo CoAP

O CoAP (*Constrained Application Protocol*), é um protocolo concebido para IoT, e é orientado para o funcionamento com dispositivos *constrained*, que são dispositivos com recursos limitados.

Diferente do protocolo MQTT, o protocolo CoAP rege-se por uma arquitetura *request / response*, onde existem dois tipos de dispositivos: os *CoAP Servers*, normalmente dispositivos que providenciam informação, como sensores; e os *CoAP Clients*, que são dispositivos que querem aceder à informação dada pelos *CoAP Servers*.

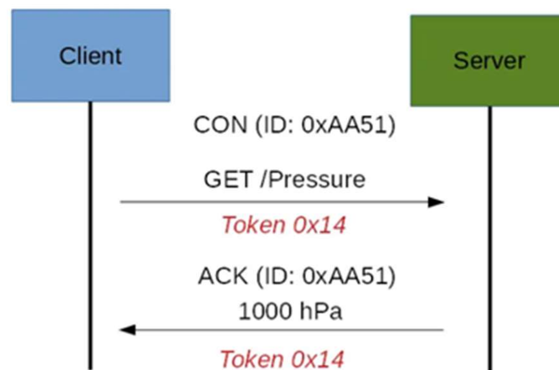


Figura 8 - Arquitetura *Response/Request* com *token* e resposta imediata

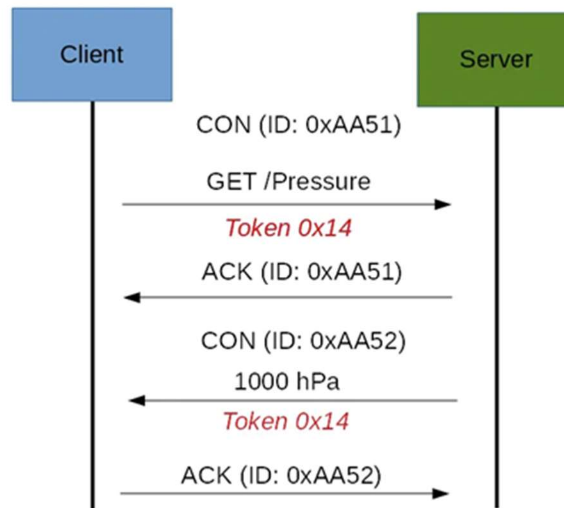


Figura 9 - Arquitetura *Response/Request* com *token* e resposta não imediata

Neste tipo de arquitetura, como podemos ver na Figura 8 e na Figura 9, como os nomes indicam, funcionam sob a forma de pedido e resposta. Um dispositivo cliente vai efetuar um pedido (GET) a um dispositivo servidor, e este irá responder com um reconhecimento (ACK), contendo este a resposta ao pedido do servidor ou um erro [14].

O CoAP utiliza um *token* de pedido, que vai adicionar alguma confiabilidade ao envio dos pedidos, de modo que haja então alguma "confirmação" de como cada pedido é de um certo cliente, e que cada resposta vai ser atribuída a esse mesmo cliente. Em cada pedido pode ser requerido um *token* (opcional) para que o servidor, quando for responder a esse pedido, saiba qual foi o cliente.

No caso de o servidor não conseguir responder imediatamente, envia um *acknowledgement* (ACK) sem o *token* do pedido, mas logo que possa, envia novamente uma resposta sob a forma de uma mensagem de confirmação do pedido feito, que não obteve resposta imediata (CON).

Como explicado anteriormente, é possível que este *token* seja inexistente, se o tipo da mensagem for não-confirmável (NON). No caso de o tipo da mensagem ser confirmável (CON – onde o servidor exige um ACK por parte do cliente), é enviado o *token*, caso contrário, a mensagem não é confirmável (NON – onde o servidor não exige um ACK por parte do cliente), e não é assegurada a confirmação supracitada.

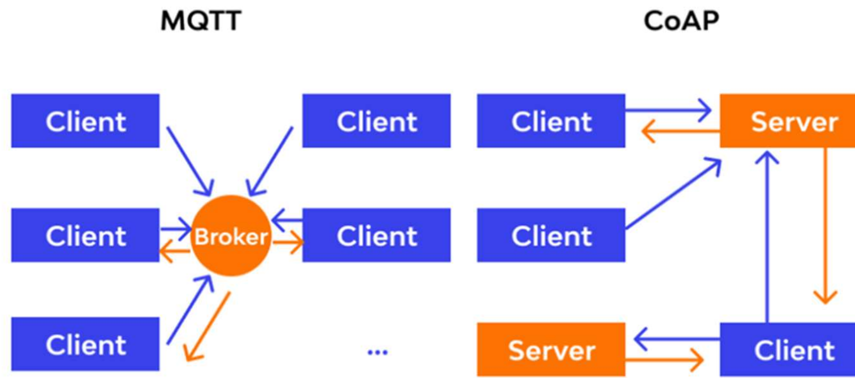


Figura 10 - Diferença entre Arquitetura dos protocolos MQTT e CoAP

Como se observa na Figura 10, a grande diferença entre a arquitetura *publish / subscriber* e a arquitetura *request / response*, é o facto de a primeira utilizar *brokers* que armazenam a informação, e que posteriormente, outros dispositivos acedem ou recebem essa informação; na segunda, cada dispositivo que providencia algum tipo de informação, funciona como um servidor, ao qual os dispositivos que querem aceder a essa informação, tem de fazer um pedido (*request*), e receber uma resposta (*response*) [15].

A Figura 11, mostra o formato de uma mensagem que utilize o protocolo CoAP, que possui 5 partes principais. São elas a "Ver", que é um indicador da versão que utiliza 2 *bits*, o "T", que indica o tipo da mensagem (confirmável - CON ou não confirmável - NON) e utiliza 2 *bits*, o "*Token Length*", que representa o tamanho que o *token* pode ocupar e utiliza 4 *bits*, o "Code", que representa o código da resposta e utiliza 8 *bits*, e o "*Message ID*" que utiliza 16 bits [16].

<table>
<tr><td>Ver</td><td>T</td><td>TKL</td><td>Code</td><td>Message ID</td></tr>
<tr><td colspan="5">Token</td></tr>
<tr><td colspan="5">Options (se houver)</td></tr>
<tr><td colspan="3">1 1 1 1 1 1 1 1</td><td colspan="2">Payload (se houver)</td></tr>
</table>

Figura 11 - Formato de mensagem (CoAP)

Como são utilizados muitos *bits* para uma mensagem, em comparação com o protocolo MQTT, o protocolo CoAP tende a ter mais problemas de *overhead*, só que este protocolo possui alguns mecanismos para reduzir esse *overhead* mais elevado, como a utilização do protocolo de comunicação da camada de transporte UDP.

Sob o protocolo UDP, que envia as mensagens sob a forma de datagramas, como medida de segurança, pode ser utilizado o protocolo de segurança DTLS (*Datagram Transport Layer Security*) que irá criar um canal seguro de transporte dos datagramas.

Outros mecanismos de segurança para o protocolo CoAP são a utilização de chaves que ficam associadas a qualquer dispositivo cliente que troque mensagens com um dispositivo servidor para que a confiabilidade no envio de mensagens seja aumentada, ou até mesmo a utilização de certificados que comprovem a identidade do emissor nas mensagens, de modo a que estas não sejam alteradas ou intercetadas.

Relativamente à escalabilidade do protocolo CoAP, pode-se pensar que este não é um protocolo escalável por este ser destinado a dispositivos com recursos limitados, mas existe uma implementação em JavaScript (*californium* [17]) que permite a utilização do protocolo CoAP em dispositivos não restritos, focando-se na usabilidade e escalabilidade do protocolo, e reduzindo a eficiência e recursos do mesmo.

O facto de vários dispositivos, quer sejam IoT ou não, possam utilizar protocolos como o protocolo de segurança DTLS [11], reforça a existência de interoperabilidade para o protocolo CoAP, assim como o facto de vários outros protocolos de comunicação em IoT, como o protocolo HTTP (*Hypertext Transfer Protocol*), utilizarem o mesmo tipo de arquitetura (*Request / Response*) que o protocolo CoAP, que assegura semelhanças no funcionamento, e, por conseguinte, interoperabilidade.

2.2.3. Outros Protocolos

Existem muitos outros protocolos para IoT que possuem características semelhantes às características apresentadas nos protocolos MQTT e CoAP.

Um deles é o protocolo AMQP (*Advanced Message Queuing Protocol*), que é bastante utilizado no processo de desenvolvimento de aplicações *open-source* relacionadas a clientes e *brokers*, possuindo assim uma arquitetura do tipo *Publish / Subscriber*. Relativamente ao seu formato de mensagem, é composto por duas partes variáveis (*Payload* e *Performative)*, que correspondem ao conteúdo da mensagem e ao tipo de mensagem, que dependendo do conteúdo, podem variar no número de *bits*; e a três partes constantes (*Protocol Header* que possui 8 *bits*, o *Frame Header* que possui 8 bits e contém informações sobre a mensagem e o *End of Frame* que possui 1 *bit* e simboliza o término da mensagem), totalizando um mínimo de 17 *bits* por mensagem, o que leva a uma possibilidade maior de *overhead* em

comparação com os protocolos anteriormente explicados. O protocolo de comunicação da camada de transporte utilizado é o TCP, logo, assim como o protocolo MQTT, uma alternativa de segurança poderá ser a utilização do protocolo de segurança TLS que irá cifrar as mensagens.

Relativamente a implementações, este protocolo possui algumas implementações: como o RabbitMQ, destinado à escalabilidade em *cloud*, concebido para fornecer mensagens via adaptadores SMTP (*Simple Mail Transfer Protocol*), STOMP (*Streaming Text Oriented Messaging Protocol*), e HTTP (*Hypertext Transfer Protocol)*; e o OpenAMQ que é um *broker* com bibliotecas em C e C++, que contém uma API (*Application Programming Interface*) com ferramentas de administração remotas e muitas outras funções [18].

Um protocolo mais antigo é o protocolo XMPP (*Extensible Messaging and Presence Protocol*), utilizado maioritariamente para aplicações de mensagens, como chats e emails, possibilitando também possível gerir dispositivos, redes e serviços de forma segura, sólida e interoperável [19]. Possui uma arquitetura *Request / Response*, e TCP é o seu protocolo de comunicação da camada de transporte, sendo assim bastante semelhante em termos de arquitetura ao protocolo MQTT, e em termos de soluções de segurança como a utilização do protocolo de segurança TLS que irá cifrar as mensagens. Relativamente a escalabilidade e interoperabilidade, é um protocolo que tem mais de 10 anos de testes de escalabilidade e segurança, possuindo uma grande quantidade de servidores e milhões de utilizadores espalhados pelo mundo. Está disponível em várias linguagens de programação, e para qualquer tipo de servidor e cliente, reforçando assim a interoperabilidade deste protocolo, uma vez que poderá ser utilizado em vários projetos, sem que compatibilidade seja um problema [20].

2.3.Algoritmos de *hashing*

Após anteriormente se terem analisado alguns protocolos de comunicação existentes, é essencial abordar de que forma garantimos a integridade dos dados que viajam através desses protocolos. Para isso são apresentados dois grupos principais de algoritmos: os de *hashing* e os de cifragem. No primeiro subcapítulo, é explicado o conceito de *hashing*, elucidando o seu papel crucial para garantir a integridade das mensagens relativamente à segurança. Serão apresentados exemplos de como uma função de *hashing* funciona e serão detalhados os

algoritmos específicos adotados neste projeto. No capítulo 3, serão apresentadas as implementações dos algoritmos para a realização dos testes.

2.3.1. Função de hashing

Um algoritmo de *hashing* é uma função matemática que a partir de uma mensagem vai gerar um "código de confirmação" que servirá para validar a integridade da mensagem.

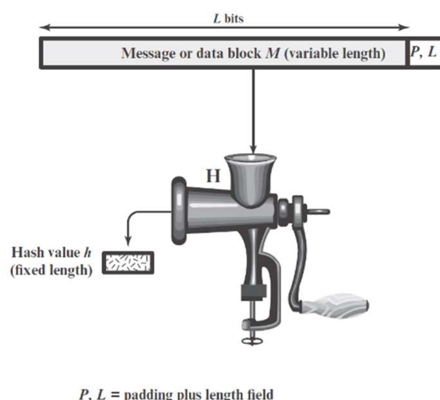


Figura 12 - Geração de um *hash*

Na Figura 12, é possível compreender o processo de geração de um *hash*, onde à mensagem "M" é aplicado o algoritmo de *hashing* "H", que resulta num valor fixo mais reduzido, o *hash* "k" [21].

De modo a dar um exemplo mais prático do que é que uma função de *hashing* produz, é apresentado o resultado do *hash* da mensagem "Chicago" realizado por dois algoritmos de *hashing* diferentes.

MD5: 72b003ba1a806c3f94026568ad5c5933

SHA-256: f6bf870a2a5bb6d26ddbeda8e903f3867f729785a36f89bfae896776777d50af

Como vemos, para a mesma mensagem, são gerados resultados diferentes. Isto ocorre uma vez que cada algoritmo de *hashing* possui diferentes características, como o *digest* da mensagem, que no MD5 (*Message-Digest algorithm 5*) produz um *hash* de 128 *bits* resultando em 32 caracteres hexadecimais e o SHA-256 produz um *hash* de 256 *bits* resultando em 64 caracteres hexadecimais, o número de *bits* representáveis, as funções de compressão e transformação diferentes, entre outros fatores.

As funções de *hashing* pretendem assim assegurar a integridade de uma mensagem, conforme a Tríade CIA (*Confidentiality, Integrity e Availability*).

As funções de *hashing* podem ser aplicadas em diversos cenários práticos [22]:

- **Message Digest:** Consiste na aplicação de funções criptográficas de *hash* para garantir a integridade dos ficheiros. Ao armazenar arquivos na *cloud*, o *hash* do ficheiro é calculado e armazenado localmente. Ao ir buscar os ficheiros armazenados, o *hash* é recalculado e comparado para verificar se o arquivo foi alterado.
- **Verificação de senha:** As funções de *hash* podem ser utilizadas para verificar senhas. Ao fazer login, o *hash* da senha é calculado e comparado com o *hash* armazenado no servidor, garantindo que a senha original não seja exposta durante a transmissão.
- **Para operações do compilador:** Para distinguir palavras-chave de uma linguagem de programação (como "if", "else", entre outros) de outros identificadores, o compilador pode armazenar essas palavras-chave num conjunto implementado utilizando uma tabela *hash*.
- **Fazer a ligação entre o nome e o caminho de um ficheiro:** Num sistema local, para armazenar a ligação entre o nome e o caminho de um ficheiro, é utilizado um mapa (nome_do_arquivo, caminho_do_arquivo) implementado com uma tabela *hash*.

2.3.2. Algoritmos de *hashing* em análise

Neste subcapítulo, procede-se à análise detalhada dos algoritmos MD5, SHA-256, Blake2 e RIPEMD-160. A sua análise focará em componentes específicos destes algoritmos, como o impacto nos parâmetros em análise na presente tese, o tamanho do *hash* gerado e reflexões pertinentes sobre a sua robustez e segurança no geral.

MD5: Este é um dos primeiros algoritmos a obter uma aprovação generalizada. Foi concebido em 1991 e, na altura, foi considerado extremamente seguro. Desde então, inúmeros *hackers* descobriram como atacar facilmente mensagens que utilizaram este algoritmo para realizar o *hash* de uma mensagem, e atualmente já o conseguem fazer em questões de segundos através da utilização de ataques de *brute force* e utilizando *hardware* convencional, nomeadamente placas gráficas. Para a maioria dos cenários possíveis de

aplicação de algoritmos de *hashing*, considera-se que este algoritmo já não é seguro, uma vez que é computacionalmente fácil de atacar.

Algo que é intrínseco aos algoritmos de *hashing*, é não ser possível reverter o processo de *hashing* e gerar o ficheiro/mensagem original a partir do *hash* gerado. No algoritmo MD5, cada input vai gerar sempre o mesmo *hash* de 32 caracteres hexadecimais e isso torna este algoritmo útil para validação de ficheiros/mensagens.

Cada *hash* gerado com o algoritmo MD5 é de 16 *bytes* e é representado por 32 caracteres hexadecimais. Isto pode ser explicado pelo facto de cada caracter representar 4 *bits* (logo o total de *bits* = 4 x 32 = 128 *bits*), e como 8 *bits* equivale a 1 *byte*, 2 caracteres equivalem a 8 *bits*, e por conseguinte 1 *byte*.

O tamanho do *hash* gerado pode influenciar os parâmetros em análise, como a utilização do CPU ser menor uma vez que processar uma *hash* de uma mensagem pode requerer mais poder computacional, mas enviar uma *hash* ou a mensagem em texto simples é completamente diferente; a utilização de memória ser menor uma vez que armazenar uma *hash* requer menos memória do que armazenar uma mensagem em texto simples; e a latência ser menor uma vez que quanto maior a mensagem, maior a latência esperada, e como as *hashes* geradas neste projeto serão superiores às mensagens em texto simples, é de se esperar que a latência seja menor.

Relativamente à complexidade do algoritmo MD5, a taxa de transferência pode ser influenciada por este algoritmo, uma vez que, como sugere um estudo feito sobre o MD5 [23], a complexidade do mesmo pode influenciar a taxa de transferência através de diferentes implementações do algoritmo, uma vez que como processa blocos de 512 *bits* em 64 passos que manipulam 128 *bits* de uma vez, esperavam uma taxa de transferência de 165 Mbps e obtiveram entre 87/100 Mbps.

Talvez a maior vulnerabilidade deste algoritmo seja a "MD5 Collision" [24], onde um atacante, como podemos ver através do esquema da Figura 13, pode criar um ficheiro que gere o mesmo *hash* que uma mensagem intercetada por ele, uma vez que sendo um algoritmo com pouca possibilidade de *hash* gerado (pelo seu tamanho de *hash* produzido), há ficheiros/mensagens que podem gerar o mesmo *hash*, e isso pode levar à receção de ficheiros/mensagens maliciosas em vez do ficheiro/mensagem esperada [24].

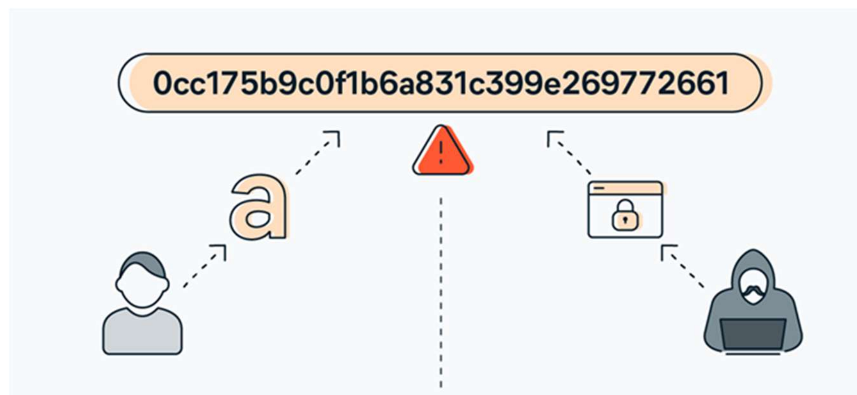


Figura 13 - MD5 Collision

Sobre o contexto geral de segurança deste dispositivo, podemos concluir que não possui a melhor segurança, mas que em termos de integridade, pode ser uma ótima solução devido à sua simplicidade que a torna mais rápida de confirmar.

SHA(-256): Os algoritmos da família SHA (*Secure Hash Algorithm*) são considerados ligeiramente mais seguros. As primeiras versões foram desenvolvidas pelo governo dos Estados Unidos, mas outros programadores desenvolveram as estruturas originais e tornaram as variações posteriores mais rigorosas e difíceis de quebrar. Em geral, quanto maior o número após as letras "SHA", mais recente é a versão e mais complexo é o programa. Por exemplo, o SHA-3 inclui fontes de aleatoriedade no código, o que o torna muito mais difícil de decifrar do que os anteriores. Por esse motivo, tornou-se um algoritmo de *hashing* padrão em 2015.

O algoritmo SHA-256 é uma variante do algoritmo SHA-2, o sucessor do primeiro algoritmo SHA, o SHA-1 [25]. Este produz um valor de 256 *bits*, sendo a *hash* gerada composta por 64 caracteres perfazendo 32 *bytes*, ou seja, explicado estes valores atendemos ao facto de cada caracter representar 4 *bits* (logo o total de *bits* = 4 x 64 = 256 *bits*), e como 8 *bits* equivale a 1 *byte*, 2 caracteres equivalem a 8 *bits*, e por conseguinte 1 *byte*.

Novamente, o tamanho da chave pode influenciar os parâmetros em análise, como: a utilização do CPU ser menor uma vez que processar uma *hash* de uma mensagem pode requerer mais poder computacional, mas enviar uma *hash* ou a mensagem em texto simples é completamente diferente; a utilização de memória ser menor uma vez que armazenar uma *hash* requer menos memória do que armazenar uma mensagem em texto simples; e a latência ser menor uma vez que quanto maior a mensagem, maior a latência esperada, e como as

hashes geradas neste projeto serão superiores às mensagens em texto simples, é de se esperar que a latência seja menor.

Relativamente à complexidade do algoritmo e o impacto que este pode ter sobre os parâmetros em análise, um estudo feito sobre a taxa de transferência com a implementação do algoritmo SHA-256 [26] referiu que o método de transformação deste algoritmo é benéfico se parâmetros como a taxa de transferência no envio de mensagens for crucial no cenário real em questão, tudo isto por causa da redução de 64 para 32 ciclos no que toca à transformação do conteúdo, ou formação da *hash*.

Devido ao facto do seu baixo consumo de CPU, o SHA-256 é o algoritmo utilizado em redes de Bitcoin [27], que utilizam a tecnologia Blockchain, considerada bastante segura, e uma vez que este algoritmo produz *hashes* únicas (porque produz um *hash* diferente para cada mensagem, que se difere de todos os outros à mínima alteração no conteúdo original) e irreversíveis [28] (não sendo possível obter o conteúdo original através de engenharia reversa), torna-o ideal para ser utilizado numa tecnologia de segurança de dados tão conhecida como a blockchain.

Falando sobre vulnerabilidades e segurança que este algoritmo oferece, este algoritmo é bastante mais seguro que o algoritmo MD5, uma vez que devido à sua *hash* gerada com o dobro do tamanho, a probabilidade de ocorrer um ataque de colisão é mais reduzida. No entanto, no contexto de ser utilizado em sistemas de autenticação, por exemplo, continua vulnerável a ataques de força bruta no caso de serem utilizadas *passwords* fracas ou com baixa entropia, que é a complexidade de uma mensagem ou *password* baseada na sua aleatoriedade, quantidade de caracteres e disposição dos mesmos [29].

Blake-2: O Blake-2 é um algoritmo de *hashing* moderno que oferece alto desempenho e segurança. Ele é uma evolução do antigo algoritmo Blake e está disponível em duas versões: Blake-2b, otimizado para arquiteturas de 64 *bits*, e Blake-2s, otimizado para sistemas de 8 a 32 *bits*. Blake-2 é amplamente utilizado em várias aplicações de segurança devido à sua velocidade e resistência a ataques criptográficos conhecidos.

O Algoritmo Blake2 é um algoritmo mais rápido que os algoritmos de *hashing* anteriormente apresentados, como podemos ver na Figura 14 [30], porém é relativamente menos seguro do que a sua versão original, o SHA-3.

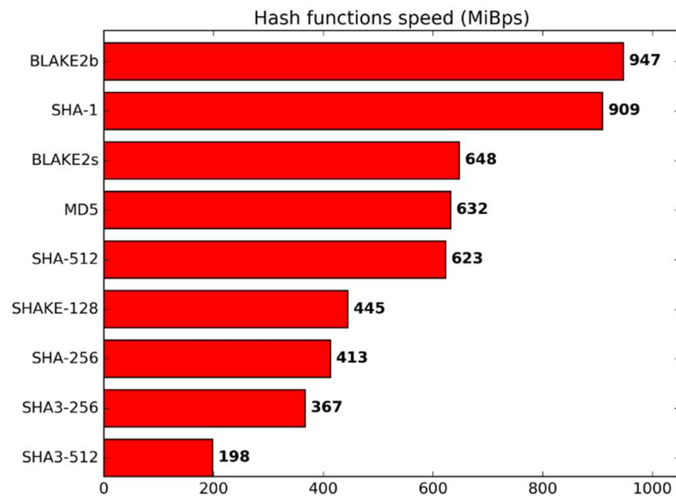


Figura 14 - Diferentes velocidades de diferentes algoritmos de *hashing*

O algoritmo Blake-2 possui duas versões que diferem no tamanho da *hash* gerada, onde a versão Blake2b gera uma *hash* com 64 *bytes*, e a versão Blake2s gera uma *hash* com 32 *bytes* [31].

Relativamente ao impacto que este algoritmo pode ter nos parâmetros em análise, um estudo feito sobre várias características do algoritmo Blake2 [32] afirma que a "arquitetura compacta" do algoritmo Blake2 pode reduzir em cerca de 34% a utilização de memória, e em comparação com o algoritmo SHA-2, apresenta uma melhoria de 36% na taxa de transferência. Outro estudo que foca na implementação do algoritmo Blake2 em sistemas de armazenamento [33] referiu que este algoritmo consome recursos significativos de CPU, que comprova o facto de este ser mais rápido que os outros algoritmos de *hash*.

Falando sobre vulnerabilidades que este algoritmo possa conter, é relevante lembrar que este algoritmo é baseado no algoritmo Blake que foi finalista na competição de *hash* do NIST (*National Institute of Standards and Technology*) que decorreu entre 2007 e 2012 para apurar o algoritmo escolhido para ser o padrão do SHA-3, logo este algoritmo possui uma boa avaliação de segurança. Em contrapartida, o facto de ser um algoritmo relativamente recente em comparação com os restantes algoritmos anteriormente referidos, não possui tanta análise de falhas e vulnerabilidades. Escolher a versão errada (Blake2s ou Blake2b) para cada cenário pode resultar em vulnerabilidades que não existiriam se a versão correta para cada cenário fosse bem escolhida, uma vez que cada uma destas versões do algoritmo Blake2 é adequada a certos tipos de cenários reais específicos.

RIPEMD-160: RIPEMD-160 (*RACE Integrity Primitives Evaluation Message Digest*) é um algoritmo de *hashing* desenvolvido em 1996 baseado no algoritmo MD4, e é uma versão melhorada do algoritmo RIPEMD cuja diferença está no tamanho das *hashes* produzidas, onde o RIPEMD produz *hashes* de 128 *bits*, e o RIPEMD-160 produz *hashes* de 160 *bits* (este é um dos fatores que levou este algoritmo a ser escolhido, uma vez que possui um tamanho de *hash* diferente dos restantes algoritmos de *hashing*).

Possui várias versões desde o RIPEMD(-128), a sua primeira versão que produz *hashes* de 128 *bits*, o RIPEMD-160, que produz *hashes* de 160 *bits*, aumentando assim a segurança do algoritmo em comparação com a sua versão anterior [34], e as versões RIPEMD-256 e RIPEMD-320 que foram uma fusão das duas primeiras versões, garantindo a segurança do RIPEMD-160 e a rapidez de computação do RIPEMD(-128), uma vez que quanto maior o tamanho da *hash* gerada, mais recursos computacionais serão utilizados, e mais lento o algoritmo será.

Relativamente ao impacto que este algoritmo pode ter nos parâmetros em análise, como explicado anteriormente, por ser um algoritmo mais lento em comparação com os anteriores, é de se esperar que possua uma utilização de CPU maior, e um estudo que comparou o desempenho em termos de utilização de CPU entre os algoritmos MD4, MD5, SHA-1, RIPEMD-128 e RIPEMD-160 [35], menciona que o RIPEMD-160 é cerca de 15% mais lento que o algoritmo SHA-1 e 4 vezes mais lento que o algoritmo MD4 como podemos ver na Figura 15 [35].

algorithm	performance (Mbit/s)	
	Assembly	C
MD4	190.6	81.4
MD5	136.2	59.7
SHA-1	54.9	21.2
RIPEMD-128	77.6	35.6
RIPEMD-160	45.3	19.3

Figura 15 - Performances de vários algoritmos de *hashing* (CPU)

Falando sobre a segurança no geral deste algoritmo, é considerado mais seguro do que alguns algoritmos anteriormente referidos como o MD5, e foi desenhado para resistir a ataques de colisão de *hash*. Em contrapartida é um algoritmo mais lento e não possui tanta utilização como os algoritmos MD5, SHA-256 e Blake2, logo não foi tão analisado e testado como os algoritmos anteriores.

2.4. Algoritmos de cifragem

Neste tópico será explicado como funciona um algoritmo de cifragem, como é que os algoritmos de cifragem foram evoluindo ao longo do tempo, e serão descritos os algoritmos de cifragem específicos utilizados neste projeto.

Um dos principais fundamentos da criptografia, que consiste na conversão de dados num formato legível para um formato codificado, é a cifragem/decifragem de dados [36].

É possível perceber como funciona um algoritmo de cifragem, observando a Figura 16, que nos mostra que existe uma chave para decifrar uma mensagem de *Plain Text* (texto simples, legível) para *Cipher Text* (texto codificado), e é essa mesma chave que vai ser utilizada para converter novamente o *Cipher Text* em *Plain Text* por outro utilizador/dispositivo [37].

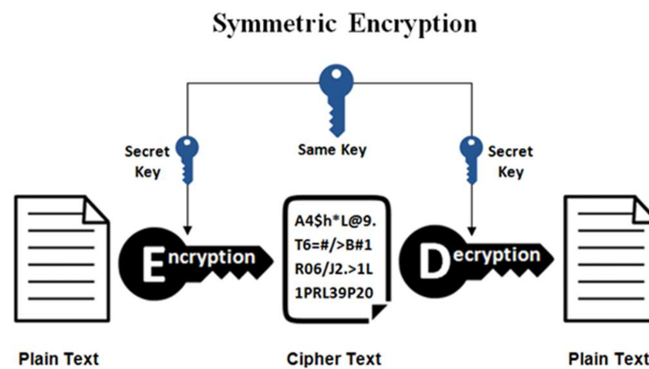


Figura 16 - Esquema de como funciona uma cifragem

Os algoritmos de cifragem são utilizados em vários cenários, quer seja para cifrar dados que estão em trânsito, ou seja, dados que quando acedemos a um site na Internet estão a ser pedidos ao servidor ou enviados por um servidor, cifrar dados armazenados nos próprios servidores, ou até mesmo em máquinas locais (computadores próprios) e dispositivos locais (discos rígidos), ou até mesmo para criar comunicações seguras entre dispositivos, como quando se utiliza um *chat* de uma rede social. Por exemplo, a rede social Telegram, utiliza o protocolo MTPro que utiliza um dos algoritmos de cifragem da família de algoritmos AES para garantir a confidencialidade das mensagens, uma vez que este é um algoritmo de cifragem forte e muito utilizado, onde cada mensagem é cifrada com uma chave diferente, gerada aleatoriamente [38].

Estes são desenvolvidos para responder a um dos propósitos da Tríade CIA, neste caso a Confidencialidade, uma vez que cifrar dados/mensagens garante a sua confidencialidade.

Para a Integridade, por exemplo, existem algoritmos de troca de chaves (pública – chave pública de cada dispositivo, chave privada – chave privada de cada dispositivo) como o algoritmo HMAC para realizar trocas de dados/mensagens com a garantia de que o recetor é o recetor verdadeiro dos dados/mensagens e que o emissor é o emissor verdadeira das mensagens.

2.4.1. Evolução de algoritmos de cifragem

Existem várias formas de um algoritmo de cifragem ficar mais seguro em relação a outros algoritmos já existentes. Dois exemplos são o aperfeiçoamento da função matemática do algoritmo, e a adição de outros mecanismos, como adicionar um algoritmo de *hashing*.

Relativamente ao aperfeiçoamento da função matemática de um algoritmo, um estudo feito [39] que visava a construção de uma versão modificada do algoritmo "Cifra de César". Este algoritmo de cifragem é dos mais conhecidos, uma vez que é dos mais antigos, utilizado quando Júlio César governava o império romano. Este simples algoritmo, que lhe valeu nenhuma mensagem ter sido intercetada e compreendida na altura, consistia em cada caracter da mensagem ser trocado pela terceira letra seguinte do alfabeto relativamente à cifragem, e cada caracter dessa mensagem cifrada ser trocado pelo terceiro caracter anterior do alfabeto, relativamente à decifragem (como vemos, a chave para cifrar é a mesma, ou inversa, para decifrar). A palavra "Chicago", passaria a ser "Fklfdjr", e esta nova palavra, se decifrada com a Cifra de César, voltaria a "Chicago".

Este estudo supracitado consistiu em alterar a rudimentar função matemática da Cifra de César (que consistia em +3 e -3) para algo mais complexo, como podemos observar na Figura 17 [39].

The Proposed Encryption Algorithm:

1- *Read the Plaintext.*
2- *Calculate N number that is (the number of characters in the plain text Mod 26) .*
3- *Do step 4 for N times :*
4- *For each Plaintext letter P :*
   A. *Calculate K from:*

$$K = \begin{cases} (i * P_{i-1}) & if\ (i > 1) \\ (i * C_{N}) & if\ (i = 1) \end{cases}$$

i : is the position of the letter to be encrypted.

   B. *substitute the Ciphertext letter C as* [C= (P + K) Mod 255].

Figura 17 - Versão modificada de algoritmo de cifragem

Outra forma de melhorar um algoritmo de cifragem é adicionar uma função de *hashing* sobre a mensagem original, e adicionar esse *hash* gerado à mensagem original antes da chave de decifragem atuar sobre a mensagem original. Com a mensagem original transformada em codificada, basta decifrar a mensagem e ainda verificar se o *hash* é o verdadeiro.

Não é propriamente uma melhoria na componente da Confidencialidade (que faz parte da Tríade CIA que os algoritmos de cifragem tendem a ter impacto), mas é um acrescento da componente de Integridade, que por si só já pode ser uma melhoria ao nível da segurança que este algoritmo vai proporcionar.

2.4.2. Algoritmos de cifragem em análise

Neste subcapítulo, procede-se à análise detalhada dos algoritmos AES, RC4, DES e Blowfish. A sua análise focará em componentes específicos destes algoritmos, como o impacto nos parâmetros em análise na presente tese, a natureza das suas chaves (simétrica ou assimétrica), o tamanho da chave e reflexões pertinentes sobre a sua robustez e segurança global.

AES: O algoritmo AES (*Advanced Encryption Standard*) é um algoritmo de cifragem simétrica, amplamente adotado. Ele suporta tamanhos de chave de 128, 192 ou 256 *bits*, conhecido pela sua eficiência e segurança. AES é utilizado em criptografia de dados, VPNs e protocolos de segurança da Internet.

O AES é um algoritmo de chave simétrica, que proporciona confidencialidade, e, ao mesmo tempo alguma autenticação, uma vez que a informação é codificada com uma chave simétrica e não pode ser descodificada com outra chave, desde que a chave seja mantida em segredo entre as partes comunicantes [40].

O AES tem a capacidade de lidar com três tamanhos de chave diferentes: 128, 192 e 256 *bits*, em que cada um dos quais possui blocos de 128 *bits*. O tamanho da chave determina o número de passos utlizados pelo AES: 10 passos para chaves de 128 *bits*, 12 passos para chaves de 192 *bits* e 14 passos para chaves de 256 *bits* [41]. Em termos de complexidade, o AES é um algoritmo iterativo. Ele baseia-se em duas técnicas comuns para cifrar e decifrar dados, conhecidas como substituição e permutação. As técnicas de substituição e permutação traduzem-se numa série de operações matemáticas realizadas em algoritmos de cifragem em blocos de *bits*.

O impacto nos parâmetros em análise pode novamente variar uma vez que segundo um estudo [42] que comparou a utilização de CPU e a taxa de transferência entre os algoritmos AES e DES (*Data Encryption Standard*), revelou que há diferenças nos resultados obtidos através da medição de valores uma vez que possuem tamanhos de chave diferentes (64 *bits* para DES e 128, 192 ou 256 *bits* para AES) e a complexidade do algoritmo é diferente (número de passos por bloco em cada algoritmo).

Algumas inconveniências da utilização deste algoritmo surgem pelo facto de ser um algoritmo de chaves simétricas. Se a chave for intercetada ou comprometida de alguma forma, quem tiver a chave poderá decifrar os dados. Portanto, a segurança dos dados cifrados com AES depende na maioria da segurança da chave. Esta segurança da chave pode ser concebida através de uma troca segura.

RC4: RC4 (*Rivest Cipher 4*) é um algoritmo de cifragem desenvolvido por Ron Rivest em 1987. É conhecido pela sua simplicidade e velocidade, e foi amplamente utilizado em protocolos como SSL/TLS e WEP (*Wired Equivalent Privacy*). No entanto, vulnerabilidades foram descobertas ao longo dos anos, tornando-o menos seguro para aplicações modernas. Devido a essas fraquezas, a utilização do RC4 é atualmente desencorajada em favor de algoritmos mais seguros [43].

O algoritmo RC4 é um algoritmo de cifragem diferente dos restantes algoritmos de cifragem em análise, uma vez que não opera sobre blocos de *bits*, mas sim sobre fluxos de *bits*.

Relativamente ao impacto que a utilização deste algoritmo pode ter nos parâmetros em análise, um estudo feito sobre o consumo de energia, utilização de CPU e taxa de transferência entre os algoritmos AES e RC4 [44], indica que o algoritmo RC4 é muito eficiente em termos de utilização de CPU em blocos maiores, mas em blocos mais pequenos,

este apresenta usar mais tempo do CPU. Apurou também que o RC4 é tão eficiente que apenas necessita de 256 *bytes* de RAM para funcionar, e em relação à taxa de transferência, este é mais adequado para blocos grandes, enquanto o AES é mais eficiente para blocos menores.

Sobre a sua segurança, este algoritmo é vulnerável a um ataque de recuperação de chave, uma vez que um atacante consegue “recuperar” a chave utilizada pelo algoritmo apenas se tiver acesso a alguma parte da mensagem cifrada ou da mensagem original [45].

DES: O algoritmo DES (*Data Encryption Standard*) utiliza uma chave de 56 *bits*. Utilizando esta chave, o DES recebe um bloco de texto simples de 64 *bits* como entrada e gera um bloco de texto cifrado de 64 *bits*. O processo DES tem várias etapas envolvidas, sendo cada etapa designada por "passo". Dependendo do tamanho da chave que está a ser utilizada, o número de "passos" varia. Por exemplo, uma chave de 128 *bits* requer 10 "passos", uma chave de 192 *bits* requer 12 "passos", e assim por diante.

Desenvolvido em 1970, e assim como o AES, o DES é um algoritmo de troca de chaves simétrica, ou seja, utiliza a mesma chave quer para cifrar, quer para decifrar. Ele opera sobre blocos de 64 *bits*, utiliza uma chave de 64 *bits* e realiza 16 permutações e substituições como processo de cifragem.

Relativamente ao impacto que este algoritmo possa ter nos parâmetros em análise, um estudo feito com o objetivo de comparar o tempo de decifração de ficheiros de áudio com os algoritmos DES, AES e Blowfish [46] concluiu que os algoritmos AES e Blowfish possuem taxas de transferência relativamente semelhantes, enquanto o algoritmo DES tinha uma taxa de transferência com o dobro do tempo.

Outro estudo que realizou uma comparação entre os algoritmos AES e DES em termos relacionados à utilização do CPU [47], concluiu que o tempo de cifragem do algoritmo DES é relativamente superior ao tempo de cifragem do algoritmo AES, o que levou à conclusão (com confirmação) que o algoritmo DES possuía uma percentagem de utilização do CPU menor que o algoritmo AES (2% para 4.2% no Windows e 1.9% para 2% no MAC).

Falando agora sobre a segurança deste algoritmo, como possui apenas 56 *bits* de chave utilizados, sendo que os restantes 8 que perfazem os 64 servem para verificação de paridade, é considerado pequeno pelos padrões atuais de um algoritmo eficiente, tornando-o assim vulnerável a ataques de força bruta. De relembrar que com “novos” algoritmos

(desenvolvidos depois) mais seguros que foram surgindo, como o AES logo de seguida, este algoritmo caiu em desuso.

Blowfish: Criado em 1993 por Bruce Schneier como alternativa ao algoritmo DES, o Blowfish é um algoritmo de cifragem, e que utiliza troca de chaves simétrica em blocos de 64 *bits*, e cuja chave possui um comprimento de 448 *bits* [48].

Relativamente ao impacto que a utilização deste algoritmo possa ter sobre os parâmetros em análise, um estudo feito para apurar algumas diferenças de desempenho entre os algoritmos Blowfish e o seu sucessor [49], o algoritmo AES, apurou que o tempo de utilização do CPU é significativamente menor do que a utilização do CPU por parte do algoritmo AES, apresentando um tempo de processamento de 0.078 segundos para o Blowfish e um tempo de processamento de 1.544 segundos para o AES.

Outro estudo que também procurou realizar uma comparação de desempenho, mas desta vez entre os algoritmos AES, DES e Blowfish [50], chegou à conclusão que o Blowfish precisa de 1.01% menos memória do que o DES e 2.48% menos memória do que o AES para conjuntos de dados mais pequenos, e no que toca a tempo de execução, associado à utilização de CPU, o Blowfish é aproximadamente 7% mais rápido do que o AES em sistemas com restrições de tempo.

Relativamente à segurança, e tentando apurar algumas vantagens e desvantagens do algoritmo Blowfish, este aparenta ser mais rápido do que os algoritmos de cifragem anteriores, junto de uma boa eficiência. Em contrapartida é menos seguro, e o facto de lidar com blocos de tamanho de 64 *bits* pode ser um obstáculo ao lidar com grandes quantidades de dados em comparação com outros algoritmos que lidem com blocos de tamanho de 128, 192 e 256 *bits*.

3. Metodologia e algoritmos em análise

Neste capítulo será apresentada a metodologia utilizada para a realização da componente prática, incluindo o ambiente de testes onde são mostradas as configurações do ambiente e a arquitetura do mesmo. Serão definidos os parâmetros de medição em análise e apresentadas as implementações realizadas no desenvolvimento de *software*, nomeadamente os *scripts* utilizados para a medição dos parâmetros em análise com a implementação dos algoritmos de *hashing* e de cifragem em utilização.

3.1. Descrição da abordagem experimental

Nos últimos anos, devido ao facto do conceito de IoT estar a emergir cada vez mais, os dispositivos que permitem a interligação de objetos agora considerados IoT, como luzes automáticas, portas de garagem, e qualquer tipo de objeto que pode ser controlado à distância, estão também a ganhar a sua devida importância, muito por causa do seu poder para simular cenários onde os objetos normais supracitados se tornam efetivamente objetos/dispositivos IoT.

Estes dispositivos são maioritariamente SBCs (*Single Board Computers*), e entre os mais conhecidos existe o Raspberry Pi, que possui uma capacidade de processamento equivalente a um computador e algumas *boards* Arduino que vence na sua simplicidade, versatilidade e preço [51]. Para a realização deste trabalho de projeto, havia disponíveis ambos os dispositivos, mas devido à experiência do autor, foi escolhido utilizar o dispositivo Raspberry Pi.

Sendo o objetivo deste estudo analisar o impacto que a utilização de algoritmos de *hashing* e de cifragem, tendo por base resultados retirados através da medição de parâmetros (utilização de CPU, utilização de Memória, Latência e Taxa de Transferência) num ambiente IoT utilizando o protocolo MQTT e, por conseguinte, compreender como a escolha dos algoritmos afeta o desempenho em cenário reais, serão realizados testes controlados num ambiente simulado com a utilização de um Raspberry Pi 3 como cliente *publisher* MQTT e duas máquinas virtuais, uma como *broker* MQTT e outra como *subscriber* MQTT.

Esta escolha visa simular um cenário realista de comunicação IoT (pela utilização de um dispositivo IoT – Raspberry Pi) através do protocolo de comunicação MQTT.

Serão desenvolvidos *scripts* em linguagem Python para automatizar o processo de envio de mensagens para o *broker* MQTT a partir do Raspberry Pi e cada *script* incorporará um algoritmo de *hashing* ou de cifragem diferente, permitindo testar o impacto de cada algoritmo nos parâmetros de medição.

A parte prática deste projeto seguirá o seguinte procedimento: configuração do ambiente experimental, implementação dos *scripts* com os algoritmos de *hashing* e de cifragem, execução dos testes para envio das mensagens, recolha dos dados relacionados aos parâmetros de medição e análise dos resultados obtidos através dos dados recolhidos. A análise dos dados proporcionará gráficos para comparar o desempenho dos algoritmos de *hashing* e de cifragem. As conclusões serão retiradas com base nas diferenças observadas nos parâmetros de medição, procurando relacioná-las à utilização de cada algoritmo.

3.2.Ambiente de testes

Relativamente ao ambiente de testes, este será composto por três dispositivos: um Raspberry Pi 3 Model B V1.2 (ver Figura 18) para correr os *scripts* de todos os 8 algoritmos, enviar as mensagens referentes a cada tipo de teste e recolher os dados resultantes desses testes (atuando como *publisher*) e duas máquinas virtuais, onde uma atua como *broker* MQTT para receber os dados e a outra serve apenas para visualizar os dados (atuando como *subscriber*). Visto que os desempenhos dos algoritmos estão a ser medidos em condições iguais, não afetará o objetivo do projeto, ou seja, a comparação entre os vários algoritmos.

Figura 18 - Raspberry Pi 3 utilizado

De notar que os *scripts* enviam as mensagens sob a forma de *hash,* no caso dos algoritmos de *hashing,* e sob a forma cifrada no caso dos algoritmos de cifragem, não havendo necessidade de implementar nenhum mecanismo de comparação de *hash* ou de decifragem

nas máquinas virtuais, uma vez que o objetivo deste estudo é avaliar o impacto da utilização de vários algoritmos no Raspberry Pi. Como é possível ver na Figura 19, as máquinas virtuais são configuradas para ser possível criar numa delas um *publisher* que envie mensagens com as implementações de segurança (os algoritmos) e que realize as medições, e, na outra um sistema para observar as mensagens recebidas pelos *subscribers*.

3.2.1. Arquitetura do ambiente de testes

De modo a dar a compreender ao leitor como se efetuarão todos os testes e medições, tendo em conta o processo explicado anteriormente, segue-se a arquitetura do ambiente de testes.

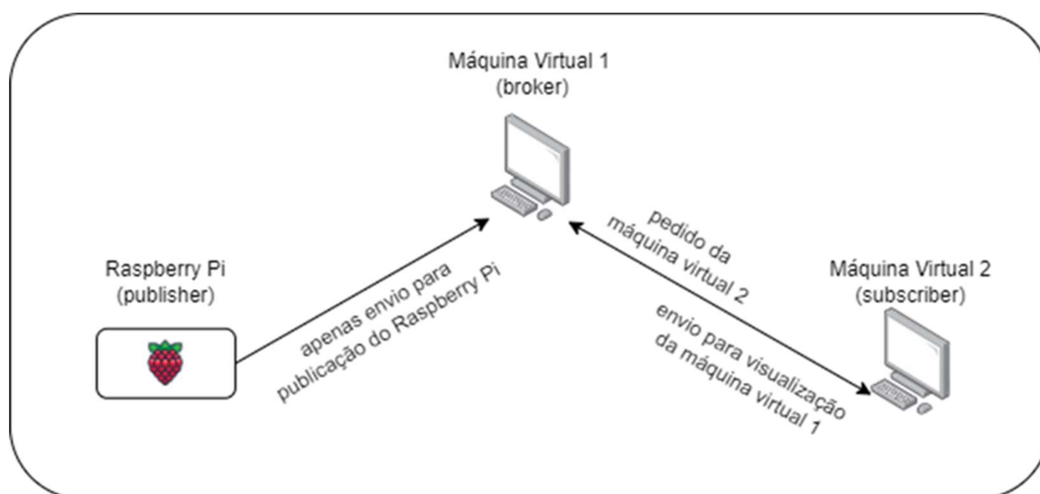


Figura 19 - Arquitetura do ambiente de testes

Como podemos ver na Figura 19, é representado o esquema de arquitetura do MQTT que apresenta as ligações entre vários clientes e o *broker* de uma arquitetura *publish / subscriber*, mas agora com os dispositivos utilizados neste projeto.

3.2.2. Configurações Raspberry Pi

Relativamente às configurações e mecanismos que permitem a execução dos testes, podemos separar por dispositivo essas mesmas configurações e mecanismos.

Para o Raspberry Pi, foi utilizado o *software* Raspberry Pi Imager [52] que vai instalar num cartão SD de 32GB (100MB/s Classe 10) da Kingston Technology, o sistema operativo Raspbian com um utilizador para aceder ao dispositivo com *username* e *password*, e vai habilitar a possibilidade de estabelecer uma ligação SSH ao Raspberry Pi, assim como aceder à rede local utilizada.

Utilizado o *software*, foi configurado o IP estático 192.168.1.10 para o Raspberry Pi poder comunicar com os restantes dispositivos, e criada a diretoria onde vão ficar registados os *scripts* utilizados neste projeto, assim como os respetivos ficheiros onde vão ser guardados os resultados de cada teste.

3.2.3. Configurações Máquina Virtual 1 (*broker* MQTT)

Para a primeira máquina virtual, que vai servir de *broker* MQTT, é necessário instalar o Mosquitto [53], que é um *broker open-source* que implementa MQTT, sendo a instalação feita através do comando "apt install mosquitto".

Posto isto, para a máquina virtual 1 se poder comportar como *broker* MQTT, é necessário aceder ao ficheiro */etc/mosquitto/mosquitto.conf* e adicionar algumas configurações:

- *bind_address 192.168.1.11* – o endereço IP colocado é o IP estático atribuído à máquina virtual que está compreendido na rede local, e este comando serve para o Mosquitto, como *broker*, permitir ligações externas para o seu próprio IP.
- *port 1883* – esta linha de código vai definir o porto padrão para a troca de mensagens MQTT.
- *allow_anonymous false* – desabilita a autenticação anónima por parte dos clientes, ou seja, é necessário a criação de *usernames* e *passwords* de acesso para a troca de mensagens MQTT.
- *passwrd_file /etc/mosquitto/passwd* – esta linha de código especifica em que ficheiro os *usernames* e *passwords* de acesso ficam registados.

Por último, é necessário criar os *usernames* e *passwords* que vão poder aceder ao *broker* MQTT. Para isso é necessário executar o comando "sudo mosquitto_passwd -c /etc/mosquitto/passwd <username>". Após substituir o parâmetro <*username*> pelo *username* de acesso, é necessário escrever a password e ficam criados um *username* e *password* prontos a utilizar.

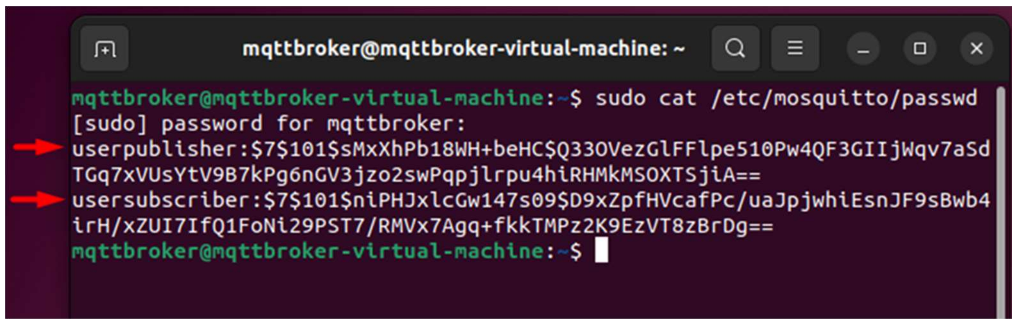


Figura 20 - Pasta com todos os *users* permitidos para comunicar com o *broker* MQTT

Como apresenta a Figura 20, no ficheiro */etc/mosquitto/passwd* é possível observar que foram criados dois *usernames*, e as respetivas *passwords* de acesso ao *broker* MQTT – uma para o Raspberry Pi (*username* = userpublisher) e outra para a máquina virtual 2 (*username* = usersubscriber).

3.2.4. Configurações Máquina Virtual 2 (*subscriber* MQTT)

Relativamente à segunda máquina virtual, que vai adotar o papel de *subscriber*, foi configurado o IP estático 192.168.1.12, compreendido na rede local utilizada para o projeto, e foi necessário instalar um cliente MQTT através do comando "apt install mosquitto-clients", para que este dispositivo seja capaz de interagir com o *broker* MQTT.

Basta executar na máquina virtual 2 o comando "mosquitto_sub -h 192.168.1.11 -t teste -u usersubscriber -P <password>" para que este dispositivo possa subscrever o tópico "teste", e ficar à escuta do *broker* MQTT (endereço IP 192.168.1.11), com o *username* anteriormente criado para este dispositivo e a respetiva *password*.

3.3. Definição dos Parâmetros de Medição

De seguida será explicado o código do programa principal, concebido e desenvolvido pelo autor deste projeto, que faz a recolha dos parâmetros da utilização da memória, da latência, da utilização do CPU e a taxa de transferência do envio das mensagens para o *broker* MQTT. Este código foi desenvolvido com recurso à linguagem Python, mais precisamente, a versão 3 do Python.

Neste subcapítulo serão explicadas as bibliotecas e as funções que tratam da recolha/medição dos parâmetros e as ferramentas e metodologias utilizadas para a sua recolha.

```
import time
import paho.mqtt.client as mqtt
import psutil
from pathlib import Path
import random
import string
```

Figura 21 - Bibliotecas do *script* de recolha de dados

Na Figura 21 é possível observar a importação de 6 bibliotecas que atuam no funcionamento do *script*. São elas:

- ***time***: este módulo fornece várias funções relacionadas ao tempo, como medir intervalos de tempo e pausar a execução do programa.
- ***paho.mqtt.client***: este módulo é utilizado para implementar clientes MQTT e possibilitar a ligação a um broker MQTT. Ele permite a publicação e subscrição de mensagens em tópicos MQTT.
- ***psutil***: este módulo é utilizado para obter informações sobre o sistema, como as utilizações de CPU e memória.
- ***Path***: O módulo pathlib é usado para trabalhar com caminhos de ficheiros de forma mais eficiente e mais legível do que os métodos tradicionais baseados em *strings*. O Path é uma classe dentro do módulo pathlib que representa um caminho de ficheiro.
- ***random***: Esta biblioteca fornece funções para gerar números aleatórios e fazer escolhas aleatórias de sequências.
- ***string***: Esta biblioteca contém várias constantes e classes úteis para manipular strings, como alfabetos, dígitos e outros caracteres especiais.

Nos próximos subcapítulos serão abordadas as funções desenvolvidas para permitir a recolha dos parâmetros de medição para cada algoritmo.

3.3.1. Utilização de CPU

Esta função, presente na Figura 22, utiliza a biblioteca "psutil" para medir a utilização de CPU do sistema, onde "psutil.cpu_percent(interval=None" retorna a utilização de CPU em percentagem durante o intervalo especificado ("None" = instantaneamente). Isto serve para medir a utilização de CPU logo no início de cada teste, e com a função presente na Figura 23, vai realizar uma medição da utilização de CPU quando o teste tiver ocorrido, efetuando uma pausa de 1 segundo em todo o código para medir novamente a utilização de CPU, e então registar uma utilização de CPU antes e depois de cada teste para realizar a diferença.

```
def medicao_CPU_antes():
        return psutil.cpu_percent(interval=None)
```

Figura 22 - Função medicao_CPU_antes

```
def medicao_CPU_depois(initial_cpu):
        time.sleep(1)
        final_cpu = psutil.cpu_percent(interval=0)
        avg_cpu = (initial_cpu + final_cpu) / 2
        return avg_cpu
```

Figura 23 - Função medicao_CPU_depois

A função retorna a utilização de CPU em percentagem.

3.3.2. Utilização de Memória

Como observado na Figura 24, a função "medicao_memoria", utiliza a biblioteca "psutil" para obter informações sobre a utilização da memória, onde "psutil.virtual_memory()" retorna um objeto que representa a memória virtual (RAM) do sistema e "percent" é um atributo desse objeto que retorna o valor da utilização de memória em percentagem. O resultado é a utilização de memória necessário para cada algoritmo realizar o teste em questão.

```
def medicao_memoria():
        uso_memoria = psutil.virtual_memory().percent
        return uso_memoria
```

Figura 24 - Função medicao_memoria

3.3.3. Latência

Como observado na Figura 25, esta função mede a latência de uma mensagem que seja publicada no tópico MQTT, uma vez que os *hashes* das mensagens enviadas em cada cenário têm o mesmo tamanho, assume-se que a latência para todas seja igual.

```
def medicao_latencia():
	resultado_latencia = 0

	def on_connect(client, userdata, flags, rc):
		global mensagem
		nonlocal resultado_latencia

		print("Connected")
		timestamp_antes = time.perf_counter()
		client.publish("teste", f"timestamp_antes: {timestamp_antes}")
		client.publish("teste", mensagem)
		timestamp_depois = time.perf_counter()
		client.publish("teste", f"timestamp_depois: {timestamp_depois}")

		timestamp_antes_ns = timestamp_antes * 1000000000
		timestamp_depois_ns = timestamp_depois * 1000000000

		latency_result = timestamp_depois_ns - timestamp_antes_ns
		client.disconnect()

	broker_ip = "192.168.1.11"

	client = mqtt.Client()
	client.username_pw_set("userpublisher", "teseipl2223")
	client.on_connect = on_connect

	client.connect(broker_ip, 1883, 60)
	client.loop_start()
	time.sleep(1)
	client.loop_stop()

	return resultado_latencia
```

Figura 25 - Função medicao_latencia

A função utiliza a biblioteca "paho.mqtt.client" para publicar e receber mensagens no *broker* MQTT e durante a publicação, são incluídos *timestamps* nas mensagens para marcar o momento exato de envio e receção.

Quando uma mensagem é recebida de volta, o tempo entre o *timestamp* de envio ("timestamp_antes") e o *timestamp* de receção ("timestamp_depois") é usado para calcular a latência. Este valor aparece em formato Unix (número de segundos desde o dia 1 de janeiro de 1970). Como a diferença apresentada entre o "timestamp_antes" e o "timestamp_depois" em segundos é muito baixa (uma vez que os resultados obtidos possuíam muitas casas decimais, por exemplo, valores como 0.000431354 segundos), é necessário multiplicar por 1,000,000,000 para ser obtido o resultado em nanossegundos, isto para visualmente se realçar a diferença entre os diferentes tempos nas comparações. Esta função retorna então a latência média em nanossegundos.

3.3.4. Taxa de Transferência

A função da Figura 26, mede a taxa de transferência de mensagens enviadas para o *broker* MQTT em mensagens por segundo.

```
def medicao_taxa_transferecncia():
        broker_ip = "192.168.1.11"
        topic = "teste"
        num_mensagens = 10
        tamanho_mensagem = 102400
        publish_intervalo = 0.01

        client = mqtt.Client()
        client.username_pw_set("userpublisher", "teseipl2223")
        client.connect(broker_ip, 1883, 60)
        client.loop_start()

        start_time = time.time()
        for i in range(num_mensagens):
                mensagem = generate_random_string(tamanho_mensagem)
                client.publish(topic, mensagem)
                time.sleep(publish_intervalo)
        end_time = time.time()

        client.loop_stop()

        tempo_decorrido = end_time - start_time
        taxa_transferencia = num_mensagens / tempo_decorrido
        return taxa_transferencia
```

Figura 26 - Função medicao_taxa_transferencia

A função utiliza a biblioteca "paho.mqtt.client" para publicar mensagens no *broker* MQTT e conta o tempo decorrido entre o início e o fim do envio das 10 mensagens aleatórias de 100 kb (102400 *bytes*) e calcula a taxa de transferência dividindo o número total de mensagens pelo tempo decorrido.

3.4.Implementação dos *scripts* de envio de mensagens

De seguida serão apresentadas as restantes funções do *script* principal e também dar a conhecer o ambiente em funcionamento aquando da execução do *script*. De notar que o procedimento para os restantes algoritmos é o mesmo, só muda o código de cada *script*.

Relativamente ao funcionamento em si da recolha dos dados através da execução de *scripts*, é possível observar na Figura 27 e Figura 28, como é que se comporta a máquina virtual que simula um utilizador que subscreveu (*subscriber*) ao tópico que possui as mensagens recebidas do *broker* pelo utilizador que publicou (*publisher* – Raspberry Pi).

Na Figura 27, vemos que o output da execução do *script* começa com os *timestamps* de envio e de receção da mensagem, assim como o número da mensagem (maioritariamente para efeitos de cálculo da latência).

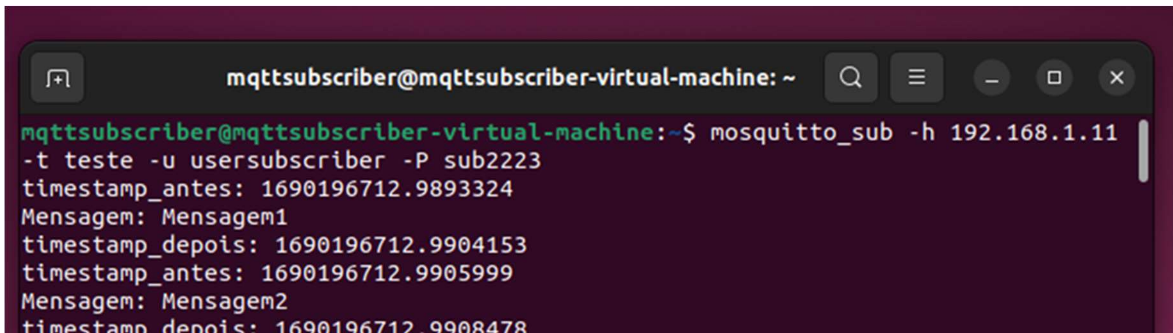


Figura 27 - Subscrição de tópico do *broker* MQTT

Na Figura 28, vemos que as mensagens são recebidas e possíveis de visualizar.

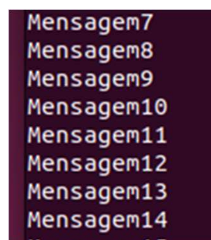


Figura 28 - Receção das mensagens por parte do *subscriber*

Relativamente ao armazenamento dos dados da recolha, estes são registados por uma função do *script* (explicada no final deste capítulo), e escritos num ficheiro com extensão ".txt", como se pode observar ver na Figura 29.

Figura 29 - Ficheiro medicoesdefault.txt

No restante conteúdo do *script*, existem outras as bibliotecas e duas funções: "grava_resultados()" e "main()".

3.4.1. Função gravar_resultados

A função "gravar_resultados" que está representada na Figura 30, é responsável por guardar os resultados das medições num arquivo de texto.

```
def gravar_resultados(resultados):
        directory = "/home/publisher/TeseIPL2223/default"
        filename = "medicoesdefault.txt"
        filepath = Path(directory) / filename

        with open(filepath, 'w') as file:
                for measurement, value in results.items():
                        file.write(f"{measurement}: {value}\n")

        print(f"Medicoes salvas em {filepath}")
```

Figura 30 - Função gravar_resultados

Esta função recebe o parâmetro *resultados* que contém as medições dos parâmetros e utiliza a biblioteca “pathlib” para criar o caminho do ficheiro “.txt” onde os resultados serão guardados.

No final, dá uma pequena indicação que as medições foram salvas no ficheiro de texto.

3.4.2. Função *main*

A função principal do *script*, “main()”, como se pode observar na Figura 31, executa cada uma das funções de medição para obter os valores correspondentes de CPU, memória, latência e a taxa de transferência, e armazena esses valores na lista “results {}”.

```
def main():
        global mensagens

        mensagens = generate_messages(10, 102400)

        resultados = {}

        uso_de_memoria = medicao_memoria()
        resultados['Uso de Memoria'] = f"{uso_de_memoria}%"

        initial_cpu = medicao_CPU_antes()

        latencia = medicao_latencia()
        resultados['Latencia'] = f"{latencia} ns"

        avg_cpu = medicao_CPU_depois(initial_cpu)
        resultados['Uso de CPU'] = f"{avg_cpu}%"

        taxa_de_transferencia = medicao_taxa_transferencia()
        resultados['Taxa de Transferencia'] = f"{taxa_de_transferencia} menssagens/segundo"

        grava_resultados(resultados)
```

Figura 31 - Função *main*

No início é apresentada a linha de código “mensagens = generate_messages(10, 102400)” que é a linha de código que vai ser alterada durante os testes, uma vez que é responsável pelo número de mensagens e pelo tamanho das mensagens enviadas em cada teste. Neste caso, o primeiro número é referente ao número de mensagens (10), e o segundo é referente ao tamanho das mensagens que vão ser cifradas ou às quais vão ser calculados os *hashes* (102400 *bytes*, que equivalem a 100 kb).

Esta função principal, executa no final a função "grava_resultados(resultados)" ao recorrer à lista "resultados{}" que vai encaminhar os resultados obtidos para o ficheiro de texto.

3.4.3. Funções de geração de mensagens aleatórias

O código principal contém duas funções presentes na Figura 32, cujo objetivo é gerar uma *string*, que será cada uma das mensagens, aleatória e de tamanho específico (função "gerar_string_random()"), e gerar uma lista de mensagens aleatórias (função "gerar_mensagens()") para que estas possam ser cifradas ou produzida a *hash* através delas e possam ser enviadas. Estas mensagens são geradas antes do cálculo de quaisquer resultados dos parâmetros em análise, logo o tempo necessário para a geração não conta para os resultados obtidos.

```
def gerar_string_random(length):
    caracteres = string.ascii_letters + string.digits
    return ''.join(random.choice(caracteres) for i in range(length))

def gerar_mensagens(num_mensagens, tamanho_mensagem)
    return [generate_random_string(tamanho_mensagem) for _ in range(num_mensagens)]
```

Figura 32 - Funções de geração de mensagens aleatórias

3.5. Implementação dos algoritmos no *script*

Neste tópico, serão descritos mais detalhadamente os algoritmos utilizados neste projeto.

Para o grupo de algoritmos de *hashing* (MD5, SHA-256, Blake2 e Blowfish) a descrição focar-se-á no tamanho do *hash* produzido, na complexidade do algoritmo e como esta pode influenciar os parâmetros medidos, no panorama de segurança que o algoritmo se insere e a implementação a fazer no *script* principal que realizará a medição dos dados para análise de resultados.

Para o grupo de algoritmos de cifragem (AES, RC4, DES e Blowfish) a descrição focar-se-á no tipo de algoritmo, no tamanho da chave, na complexidade do algoritmo, no panorama de segurança que o algoritmo se insere e a implementação a fazer no *script* principal que realizará a medição dos dados para análise de resultados.

3.5.1. MD5

Neste subcapítulo são apresentadas as alterações necessárias para implementar o algoritmo MD5 no *script* que realiza as medições dos parâmetros em análise.

Como se pode observar na Figura 33, é utilizada a biblioteca “hashlib” que implementa uma interface para algoritmos de *hash* e *digest* de mensagens [54]. É criada uma função “md5_hash()” que vai realizar o *hashing* da mensagem a enviar pelo *script*, e na Figura 34 é implementada essa mesma função na função que serve para medir a latência.

```
import string
import hashlib

def md5_hash(mensagem):
        return hashlib.md5(mensagem.encode()).hexdigest()
```

Figura 33 - Bibliotecas adicionais MD5 e função md5_hash

```
def medicao_latencia():
        resultado_latencia = 0

        def on_connect(client, userdata, flags, rc):
                global mensagem
                nonlocal resultado_latencia

                print("Connected")
                mensagem = md5_hash(mensagem)
                timestamp_antes = time.perf_counter()
                client.publish("teste", f"timestamp_antes: {timestamp_antes}")
                client.publish("teste", mensagem)
                timestamp_depois = time.perf_counter()
                client.publish("teste", f"timestamp_depois: {timestamp_depois}")
```

Figura 34 - Alteração função medicao_latencia (MD5)

Estas implementações são apenas necessárias nestas duas funções que medem os parâmetros da latência e da taxa de transferência, uma vez que as outras duas funções que medem a utilização de CPU e de memória estão a ser executadas durante cada teste, logo, não necessitam de novas implementações.

Uma última alteração é na função “gerar_mensagens()” para que às mensagens que são geradas, seja logo produzido o *hash* e então, sejam utilizadas nos testes. Podemos verificar essa alteração na Figura 35.

```
def gerar_mensagens(num_mensagens, tamanho_mensagem):
    return [md5_hash(generate_random_string(tamanho_mensagem)) for _ in range(num_mensagens)]
```

Figura 35 - Alteração função gerar_mensagens (MD5)

3.5.2. SHA-256

Neste subcapítulo são apresentadas as alterações necessárias para implementar o algoritmo SHA-256 no *script* que realiza as medições dos parâmetros em análise.

As alterações que são feitas no *script* deste algoritmo são semelhantes às alterações feitas no *script* do algoritmo MD5, uma vez que se recorre principalmente à biblioteca “hashlib” para se poder recorrer aos algoritmos de *hash*.

```
def sha256_hash(mensagem):
        return hashlib.sha256(mensagem.encode()).hexdigest()
```

Figura 36 - Função sha256_hash

```
def medicao_latencia():
        resultado_latencia = 0

        def on_connect(client, userdata, flags, rc):
                global mensagem
                nonlocal resultado_latencia

                print("Connected")
                mensagem = sha256_hash(mensagem)
                timestamp_antes = time.perf_counter()
                client.publish("teste", f"timestamp_antes: {timestamp_antes}")
                client.publish("teste", mensagem)
                timestamp_depois = time.perf_counter()
                client.publish("teste", f"timestamp_depois: {timestamp_depois}")
```

Figura 37 - Alterações função medicao_latencia (SHA-256)

```
def gerar_mensagens(num_mensagens, tamanho_mensagem):
    return [sha256_hash(generate_random_string(tamanho_mensagem)) for _ in range(num_mensagens)]
```

Figura 38 - Alterações função gerar_mensagens (SHA-256)

Como é possível observar, foram feitas as mesmas alterações realizadas no algoritmo MD5, sendo que na Figura 36 vai ser possível produzir o *hash* de cada mensagem com o algoritmo SHA-256, na Figura 37 é possível verificar as alterações na função que mede a latência, e na Figura 38 é possível verificar que a cada mensagem aleatória gerada é automaticamente calculado a *hash*.

3.5.3. Blake2

As alterações que são feitas no *script* deste algoritmo são mais uma vez semelhantes às alterações feitas nos *scripts* dos algoritmos MD5 e SHA-256, uma vez que se recorre outra vez à biblioteca "hashlib" para se poder recorrer aos algoritmos de *hash*.

```
def blake2_hash(mensagem):
        return hashlib.blake2s(mensagem.encode()).hexdigest()
```

Figura 39 - Função blake2_hash

```
def medicao_latencia():
        resultado_latencia = 0

        def on_connect(client, userdata, flags, rc):
                global mensagem
                nonlocal resultado_latencia

                print("Connected")
                mensagem = blake2_hash(mensagem)
                timestamp_antes = time.perf_counter()
                client.publish("teste", f"timestamp_antes: {timestamp_antes}")
                client.publish("teste", mensagem)
                timestamp_depois = time.perf_counter()
                client.publish("teste", f"timestamp_depois: {timestamp_depois}")
```

Figura 40 - Alterações função medicao_latencia (Blake2)

```
def gerar_mensagens(num_mensagens, tamanho_mensagem):
    return [blake2_hash(generate_random_string(tamanho_mensagem)) for _ in range(num_mensagens)]
```

Figura 41 - Alterações gerar_mensagens (Blake2)

Como é possível observar, foram feitas alterações semelhantes às efetuadas nos algoritmos MD5 e SHA-256, sendo que na Figura 39 vai ser possível produzir o *hash* de cada mensagem com o algoritmo Blake2s, na Figura 40 é possível verificar as alterações na função que mede a latência, e na Figura 41 é possível verificar que a cada mensagem aleatória gerada é automaticamente calculado a *hash*.

3.5.4. RIPEMD-160

Relativamente às alterações que são feitas no *script* deste algoritmo, a maioria são semelhantes aos anteriores, já que mais uma vez que se recorre à biblioteca "hashlib" para se poderem implementar os algoritmos de *hash*.

```
def ripemd_hash(mensagem):
        hash = hashlib.new('ripemd160')
        hash.update(mensagem.encode('utf-8'))
        return hash.hexdigest()
```

Figura 42 - Função ripemd_hash

```
def medicao_latencia():
        resultado_latencia = 0

        def on_connect(client, userdata, flags, rc):
                global mensagem
                nonlocal resultado_latencia

                print("Connected")
                mensagem = ripemd_hash(mensagem)
                timestamp_antes = time.perf_counter()
                client.publish("teste", f"timestamp_antes: {timestamp_antes}")
                client.publish("teste", mensagem)
                timestamp_depois = time.perf_counter()
                client.publish("teste", f"timestamp_depois: {timestamp_depois}")
```

Figura 43 - Alterações medicao_latencia (RIPEMD-160)

```
def gerar_mensagens(num_mensagens, tamanho_mensagem):
    return [ripemd_hash(generate_random_string(tamanho_mensagem)) for _ in range(num_mensagens)]
```

Figura 44 - Alterações função gerar_mensagens (RIPEMD-160)

Como é possível observar, foram feitas as mesmas alterações que foram feitas nos algoritmos anteriores, nas mesmas funções com exceção da função presente na Figura 42, a qual vai permitir produzir o *hash* de cada mensagem com o algoritmo Blake2s. Esta alteração que especifica a utilização do algoritmo RIPEMD-160 deve-se ao facto de este não ser um algoritmo nativo da biblioteca "hashlib". Como a biblioteca "hashlib" permite a utilização de outros algoritmos presentes em outras bibliotecas, é possível através da biblioteca "pycryptodome" utilizar este algoritmo.

Na Figura 43 é possível verificar as alterações na função que mede a latência, e na Figura 44 é possível verificar que a cada mensagem aleatória gerada é automaticamente calculado a *hash*.

3.5.5. AES

No *script* que serve para medir os parâmetros em análise neste projeto, são importadas bibliotecas que utilizam o algoritmo AES (*from Crypto.Cipher import AES*), o módulo "Random" ("from Crypto import Random") para gerar números aleatórios seguros e a função PBKDF2 (*Password-Based Key Derivation Function 2*) para derivar uma chave

segura a partir da senha. Por último é importado o módulo “base64”, utilizado para codificar dados em base64.

Na Figura 45, é possível observar que o tamanho dos blocos foi definido como 128 *bits* uma vez que o algoritmo AES opera sobre blocos de 128 *bits*. A função *pad* garante que os dados a serem cifrados tenham um tamanho que seja um múltiplo do tamanho do bloco, que é uma exigência do algoritmo AES.

```
from Crypto.Cipher import AES
from Crypto import Random
from Crypto.Protocol.KDF import PBKDF2
import base64

BLOCK_SIZE = 128

def pad(s):
        padding_length = BLOCK_SIZE - len(s) % BLOCK_SIZE
        padding = bytes([padding_length]) * padding_length
        return s + padding
```

Figura 45 - Bibliotecas e função pad (AES)

Na Figura 46 é possível verificar a função que é responsável por cifrar as mensagens com o algoritmo AES, desde a criação de um *salt* para a chave ser mais segura, até geração da chave e preparação para cifrar.

```
def aes_cifrar(mensagem):
        salt = b"this is a salt"
        key = Random.new().read(32)
        raw = pad(mensagem.encode())
        iv = Random.new().read(AES.block_size)
        cipher = AES.new(key, AES.MODE_CBC, iv)
        return base64.b64encode(iv + cipher.encrypt(raw)).decode()
```

Figura 46 - Função aes_encrypt (AES)

Na Figura 47 e na Figura 48 são apresentadas as alterações que fazem com que as mensagens sejam efetivamente cifradas com o algoritmo AES (muito semelhantes aos algoritmos de *hashing*).

```
        start_time = time.time()
        for i in range(num_mensagens):
                mensagem = aes_encrypt(generate_random_string(tamanho_mensagem))
                client.publish(topic, mensagem)
                time.sleep(publish_interval)
        end_time = time.time()
```

Figura 47 - Alterações função medicao_taxa_transferencia (AES)

```
def gerar_mensagens(num_mensagens, tamanho_mensagem):
    return [aes_encrypt(generate_random_string(tamanho_mensagem)) for i in range(num_mensagens)]
```

Figura 48 - Alterações função gerar_mensagens (AES)

3.5.6. RC4

Relativamente ao *script* do algoritmo RC4, as alterações são bastante semelhantes ao algoritmo AES. Na Figura 49, é possível observar que o tamanho dos blocos foi definido como 256 *bits* uma vez que o algoritmo RC4 não opera sobre blocos de tamanho fixo, mas sim sobre fluxos de *bits*, ou seja, sobre blocos de tamanho não fixo [55]. A função *pad* serve principalmente para garantir que os dados a serem cifrados tenham um tamanho que seja um múltiplo do tamanho do fluxo de *bits* proposto (256). Por ser um algoritmo que opera em fluxos de *bits* e não em blocos de *bits*, foi proposto o tamanho 256 para diferenciar um pouco dos restantes algoritmos e é necessário utilizar o *encoder* "latin1" em vez do *encoder* padrão "utf-8". À função *pad* é adicionada uma pequena alteração, em comparação com os outros algoritmos de cifragem, uma vez que o valor da variável "padding_length" excedia o limite de 256, e então foi necessário modificar a função *pad* para lidar com mensagens grandes sem exceder esse limite.

Para além disso também é importado o módulo ARC4 que vai possibilitar a cifragem de um fluxo de dados com o algoritmo RC4.

```
from Crypto.Cipher import ARC4
from Crypto import Random
import base64

BLOCK_SIZE = 256

def pad(s):
    full_blocks = len(s) // BLOCK_SIZE
    remaining_bytes = len(s) % BLOCK_SIZE
    padding_length = BLOCK_SIZE - remaining_bytes

    if padding_length == BLOCK_SIZE:
        return s

    padding = bytes([padding_length]) * padding_length
    return s + padding.decode('latin1')
```

Figura 49 – Bibliotecas e função pad (RC4)

Na Figura 50 é apresentada a função que vai permitir a cifragem das mensagens com o algoritmo RC4.

```
def rc4_cifrar(mensagem):
    key = Random.new().read(128)
    cipher = ARC4.new(key)
    mensagem = pad(mensagem)
    encrypted_message = cipher.encrypt(mensagem.encode())
    return base64.b64encode(encrypted_message).decode()
```

Figura 50 - Função rc4_encrypt (RC4)

Na Figura 51 e na Figura 52 são apresentadas as alterações que fazem com que as mensagens sejam efetivamente cifradas com o algoritmo RC4 (muito semelhantes aos algoritmos de *hashing*).

```
    start_time = time.time()
    for i in range(num_mensagens):
            mensagem = rc4_encrypt(generate_random_string(tamanho_mensagem))
            client.publish(topic, mensagem)
            time.sleep(publish_interval)
    end_time = time.time()
```

Figura 51 - Alterações função medicao_taxa_transferencia (RC4)

```
def gerar_mensagens(num_mensagens, tamanho_mensagem):
    return [rc4_encrypt(generate_random_string(tamanho_mensagem)) for i in range(num_mensagens)]
```

Figura 52 - Alteração função gerar_mensagens /RC4)

3.5.7. DES

O *script* relativo ao algoritmo DES é semelhante ao *script* do AES, mudando apenas o tamanho dos blocos para 64 *bits*, uma vez que é sobre este valor que o algoritmo opera.

Na Figura 53, é possível observar que o tamanho dos blocos foi definido como 64 *bits* uma vez que o algoritmo DES opera sobre blocos de 64 *bits*. A função *pad* garante que os dados a serem cifrados tenham um tamanho que seja um múltiplo do tamanho do bloco.

```
from Crypto.Cipher import DES
from Crypto import Random
import base64

BLOCK_SIZE = 64

def pad(s):
    padding_length = BLOCK_SIZE - len(s) % BLOCK_SIZE
    padding = bytes([padding_length]) * padding_length
    return s + padding.decode('utf-8')
```

Figura 53 - Função pad (DES)

Na Figura 54 é possível verificar a função que é responsável por cifrar as mensagens com o algoritmo DES, diferindo do algoritmo AES através da não utilização de um *salt* para tornar a cifragem mais segura, e apenas gerando uma chave e um objeto para cifrar e os respetivos métodos para realizar a cifragem.

```
def des_cifrar(mensagem):
        key = Random.new().read(8)
        cipher = DES.new(key, DES.MODE_ECB)
        mensagem = pad(mensagem)
        encrypted_message = cipher.encrypt(mensagem.encode())
        return base64.b64encode(encrypted_message).decode()
```

Figura 54 - Função des_encrypt (DES)

Na Figura 55 e na Figura 56 são apresentadas as alterações que fazem com que as mensagens sejam efetivamente cifradas com o algoritmo DES (muito semelhantes aos algoritmos de *hashing*).

```
    start_time = time.time()
    for i in range(num_mensagens):
        mensagem = des_encrypt(generate_random_string(tamanho_mensagem))
        client.publish(topic, mensagem)
        time.sleep(publish_interval)
    end_time = time.time()
```

Figura 55 - Alterações função medicao_taxa_transferencia (DES)

```
def gerar_mensagens(num_mensagens, tamanho_mensagens):
    return [des_encrypt(generate_random_string(tamanho_mensagens)) for i in range(num_mensagens)]
```

Figura 56 - Função gerar_mensagens (DES)

3.5.8. Blowfish

Para o algoritmo Blowfish, para além das bibliotecas e módulos referidos anteriormente, é necessário importar o módulo "Blowfish" da biblioteca "Crypto.Cipher".

Na Figura 57, é possível observar que o tamanho dos blocos foi definido como 64 *bits* uma vez que o algoritmo Blowfish, assim como o algoritmo DES, opera sobre blocos de 64 *bits*. A função *pad* garante que os dados a serem cifrados tenham um tamanho que seja um múltiplo do tamanho do bloco.

```
from Crypto.Cipher import Blowfish
from Crypto import Random
import base64

BLOCK_SIZE = 64

def pad(s):
    padding_length = BLOCK_SIZE - len(s) % BLOCK_SIZE
    padding = bytes([padding_length]) * padding_length
    return s + padding.decode('utf-8')
```

Figura 57 - Função pad (Blowfish)

Na Figura 58 é possível verificar a função que é responsável por cifrar as mensagens com o algoritmo Blowfish, que é semelhante ao método de cifragem com o algoritmo DES, que apenas gera uma chave e um objeto para cifrar e os respetivos métodos para realizar a cifragem.

```
def blowfish_cifrar(mensagem):
    key = Random.new().read(56)
    cipher = Blowfish.new(key, Blowfish.MODE_ECB)
    mensagem = pad(mensagem)
    encrypted_message = cipher.encrypt(mensagem.encode())
    return base64.b64encode(encrypted_message).decode()
```

Figura 58 - Função blowfish_encrypt (Blowfish)

Na Figura 59 e na Figura 60 são apresentadas as alterações que fazem com que as mensagens sejam efetivamente cifradas com o algoritmo Blowfish (muito semelhantes aos algoritmos de *hashing*).

```
        start_time = time.time()
        for i in range(num_mensagens):
                mensagem = blowfish_encrypt(generate_random_string(tamanho_mensagem))
                client.publish(topic, mensagem)
                time.sleep(publish_interval)
        end_time = time.time()
```

Figura 59 - Alterações função medicao_taxa_transferencia (Blowfish)

```
def gerar_mensagens(num_mensagens, tamanho_mensagem):
    return [blowfish_encrypt(generate_random_string(tamanho_mensagem)) for i in range(num_mensagens)]
```

Figura 60 - Função gerar_mensagens (Blowfish)

4. Resultados obtidos

Neste capítulo são apresentados os resultados dos testes efetuados com todos os *scripts* referentes a cada algoritmo. Os testes foram efetuados tendo em conta quatro cenários diferentes, cujo objetivo é encontrar cenários reais aos quais estes "cenários de teste" se podem assemelhar.

Tabela 1 - Cenários de teste

Cenários de teste	
10 mensagens de 100 kb	10 mensagens de 1 Mb
100 mensagens de 100 kb	100 mensagens de 1 Mb

Com a ajuda da Tabela 1 é possível compreender todos os cenários de testes, tendo-se recorrido a quatro cenários distintos pelo número de mensagens enviadas e pelo tamanho das mesmas. Os quatro cenários são o envio de 10 mensagens com 100 kb cada, o envio de 10 mensagens com 1 Mb cada, o envio de 100 mensagens com 100 kb cada e o envio de 100 mensagens com 1 Mb cada.

4.1.Resultados de utilização de CPU e utilização de memória

Relativamente aos resultados sobre a utilização de CPU e memória, de modo a simplificar a visualização e compreensão dos resultados, estes dois parâmetros foram agrupados nos mesmo gráficos, e todos os algoritmos foram colocados no mesmo gráfico, uma vez que os resultados não foram muito diferentes, ao contrário dos restantes parâmetros.

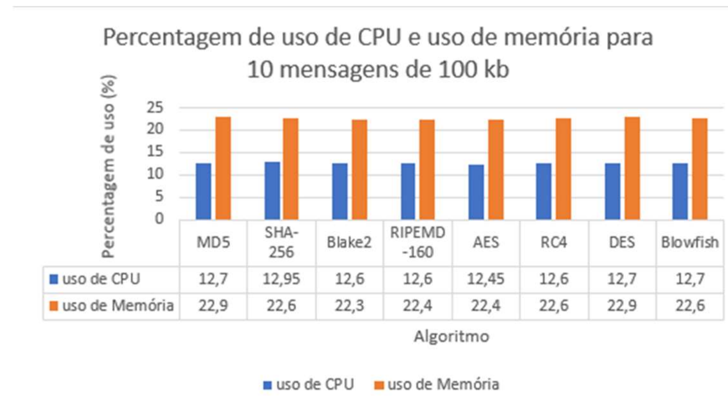


Figura 61 - Utilização de CPU e utilização de memória no envio de 10 mensagens de 100 kb

Na Figura 61, é possível verificar que para o envio de 10 mensagens com 100 kb, o algoritmo de *hashing* que obteve a maior percentagem de utilização de CPU foi o algoritmo SHA-256, com cerca de 2.77% a mais do que os algoritmos com menor percentagem de utilização de CPU, o Blake2 e o RIPEMD-160.

Os algoritmos de cifragem que obtiveram a maior percentagem de utilização de CPU foram os algoritmos DES e Blowfish, cerca de 2.01% a mais de utilização do que o algoritmo AES, que obteve a menor percentagem de utilização de CPU.

Para a utilização de memória, o algoritmo de *hashing* que obteve a maior percentagem de utilização foi o algoritmo MD5, cerca de 2.69% a mais do que o algoritmo Blake2 que obteve a menor percentagem utilização de memória.

O algoritmo de cifragem que obteve a maior percentagem de utilização de memória foi o algoritmo DES, com cerca de 2.23% a mais do que o algoritmo AES, que obteve a menor percentagem de utilização de CPU.

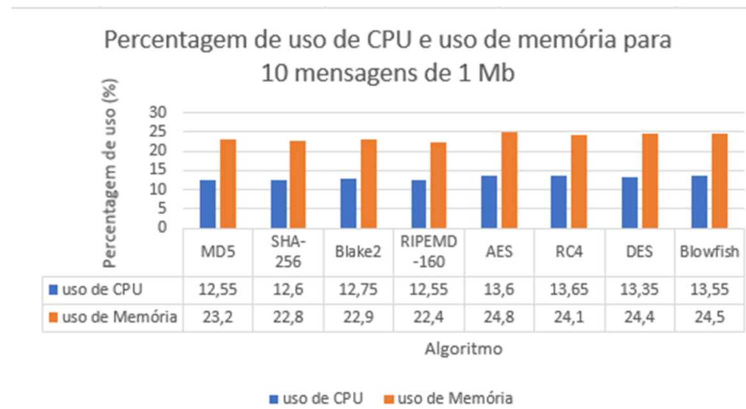


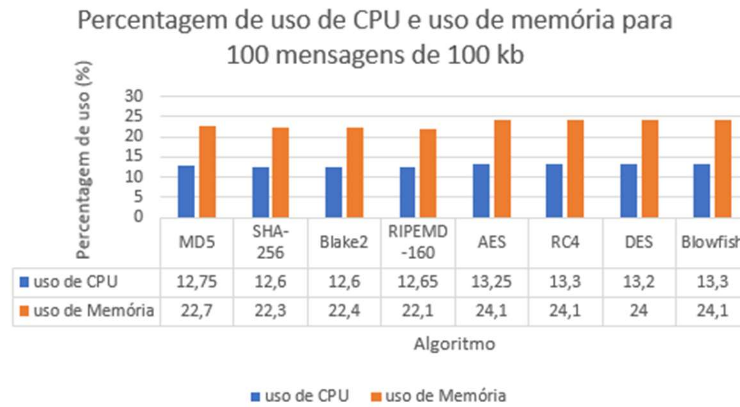
Figura 62 - Utilização de CPU e utilização de Memória no envio de 10 mensagens de 1 Mb

Na Figura 62, é possível verificar que para o envio de 10 mensagens com 1 Mb, o algoritmo de *hashing* que obteve a maior percentagem de utilização de CPU foi o algoritmo Blake2, com 1.59% a mais do que os algoritmos MD5 e RIPEMD-160, que obtiveram a menor percentagem de utilização de CPU.

O algoritmo de cifragem que obteve a maior percentagem de utilização de CPU foi o algoritmo RC4, com 2.24% a mais do que o algoritmo DES que obteve a menor percentagem de utilização de CPU.

Para a utilização de memória, o algoritmo de *hashing* que obteve a maior percentagem de utilização de memória foi o algoritmo MD5, com 3.57% a mais do que o algoritmo RIPEMD-160 que obteve a menor percentagem de utilização de memória.

O algoritmo de cifragem que obteve a maior percentagem de utilização de memória foi o algoritmo AES, com 2.90% a mais do que o algoritmo RCA, que obteve a menor percentagem de utilização de CPU.



	MD5	SHA-256	Blake2	RIPEMD-160	AES	RC4	DES	Blowfish
uso de CPU	12,75	12,6	12,6	12,65	13,25	13,3	13,2	13,3
uso de Memória	22,7	22,3	22,4	22,1	24,1	24,1	24	24,1

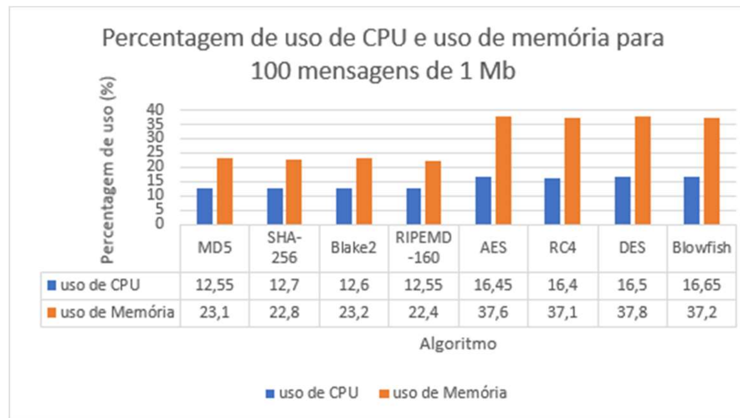
Figura 63 - Utilização de CPU e utilização de Memória no envio de 100 mensagens de 100 kb

Na Figura 63, é possível verificar que para o envio de 100 mensagens com 1 Mb, o algoritmo de *hashing* que obteve a maior percentagem de utilização de CPU foi o algoritmo MD5, com 1.19% a mais do que os algoritmos SHA-256 e o Blake2 que obtiveram a menor percentagem de utilização de CPU.

Os algoritmos de cifragem que obtiveram a maior percentagem de utilização de CPU foram os algoritmos RC4 e Blowfish, com 0.75% a mais do que o algoritmo DES, que obteve a menor percentagem de utilização de CPU.

Para a utilização de memória, o algoritmo de *hashing* que obteve a maior percentagem de utilização de memória foi o algoritmo MD5, com 2.71% a mais do que o algoritmo RIPEMD-160, que obteve a menor percentagem de utilização de memória.

Os algoritmos de cifragem, apresentaram todos o valor de 24.1%, com exceção do algoritmo DES que apresentou um valor de 24%, representando assim uma diferença de 0.41%.



	MD5	SHA-256	Blake2	RIPEMD-160	AES	RC4	DES	Blowfish
uso de CPU	12,55	12,7	12,6	12,55	16,45	16,4	16,5	16,65
uso de Memória	23,1	22,8	23,2	22,4	37,6	37,1	37,8	37,2

Figura 64 – Utilização de CPU e utilização de Memória no envio de 100 mensagens com 1 Mb

Por último, na Figura 64, é possível verificar que para o envio de 100 mensagens com 1 Mb, o algoritmo de *hashing* que obteve a maior percentagem de utilização de CPU foi o algoritmo SHA-256, com 1.19% a mais do que os algoritmos MD5 e RIPEMD-160 que obtiveram a menor percentagem de utilização de CPU.

O algoritmo de cifragem que obteve a maior percentagem de utilização de CPU foi o algoritmo Blowfish, com 1.52% a mais do que o algoritmo RC4 que obteve a menor percentagem de utilização de CPU.

Para a utilização de memória, o algoritmo de *hashing* que obteve a maior percentagem de utilização de memória foi o algoritmo Blake2, com 3.57% a mais do que o algoritmo RIPEMD-160 que obteve a menor percentagem de utilização de memória.

O algoritmo de cifragem que obteve a maior percentagem de utilização de memória foi o algoritmo DES, com 1.88% a mais do que o algoritmo RC4, que obteve a menor percentagem de utilização de CPU.

4.1.1. Comparação entre algoritmos de *hashing* e de cifragem

Antes de referir qualquer comparação, é necessário referir que as percentagens apresentadas ao longo do presente capítulo não são referentes à diferença percentual obtidas nos resultados dos testes, mas sim a diferença percentual que um algoritmo que obteve o pior resultado num parâmetro num dos cenários teve em relação ao algoritmo que obteve melhores resultados nesse mesmo parâmetro e cenário. Utilizando o último exemplo do parágrafo anterior (DES e RC4), o valor 37,8 (cenário de 100 mensagens com 1 Mb de tamanho em cada mensagem) que corresponde ao resultado do algoritmo DES é 1,0188 vezes (aumento de 1.88%) maior do que o valor 37,1 que corresponde ao resultado obtido pelo algoritmo RC4.

Tendo em conta os resultados obtidos para os algoritmos de *hashing*, neste subcapítulo serão retiradas conclusões focadas no cenário onde foram enviadas 10 mensagens com 100 kb, simulando um ambiente mais pequeno e com poucas mensagens, e outro cenário onde foram enviadas 100 mensagens com 1 Mb, que por sua vez simula um ambiente maior, com muitas mensagens, com mais informação do que o cenário anterior.

Sobre a utilização de CPU, o algoritmo RIPEMD-160 destacou-se em três dos quatro cenários em análise como o algoritmo com a menor percentagem de utilização de CPU, tornando-o no algoritmo de *hashing* mais benéfico neste aspeto.

O algoritmo de cifragem que mais se destacou relativamente à utilização de CPU foi o AES para o cenário menor (10 mensagens de 100 kb), e o RC4 para o cenário maior (100 mensagens de 1 Mb), tornando-os os algoritmos de cifragem mais benéficos neste aspeto. As diferenças de CPU nos algoritmos de cifragem também não pareceram significantes (2.01% superior ao AES no cenário menor e 1.52% superior ao RC4 no cenário maior, para o algoritmo de cifragem que obteve maior utilização de CPU, o Blowfish), mas, no que toca à utilização de CPU, um “simples” aumento de 1% pode já ser uma grande diferença.

Tendo em conta os resultados do parâmetro da utilização do CPU, os algoritmos RIPEMD-160 para *hashing* e AES para cifragem, são algoritmos que podem resultar numa maior vida útil da bateria do dispositivo IoT, assim como um desempenho mais consistente.

Relativamente à utilização de memória, os algoritmos de *hashing* que mais se destacaram foram os algoritmos Blake2 para o menor cenário (10 mensagens de 100 kb) e o RIPEMD-160 para o maior cenário (100 mensagens de 1 Mb). Apresentaram uma melhoria de 2.69% e 3.57% respetivamente em comparação com os algoritmos de *hashing* MD5 e Blake2 que obtiveram maior utilização de memória.

Para os algoritmos de cifragem que obtiveram menor utilização de memória, destacaram-se os algoritmos AES para o menor cenário, e o RC4 para o maior cenário, apresentando uma melhoria de 2.23% e de 1.88% em comparação com o algoritmo de cifragem DES que obteve maior utilização de memória.

Tendo em conta os resultados deste parâmetro, os algoritmos Blake2 e RIPEMD-160 para *hashing* e, AES e RC4 para cifragem, serão escolhas a ter no caso de o cenário possuir recursos limitados ou para um cenário onde seja necessária uma rápida inicialização, uma vez que com menos memória para gerir, os dispositivos iniciam mais rapidamente.

Tabela 2 - Melhor desempenho por cenário (parâmetro Utilização de CPU)

Cenários e Resultados – Utilização de CPU					
Tamanho da mensagem		100 kb	1 Mb	100 kb	1 Mb
Número de mensagens		10	10	100	100
Algoritmos	***Hashing***	Blake RIPEMD (12,6)	MD5 RIPEMD (12,55)	SHA Blake (12,6)	MD5 RIPEMD (12,55)
	Cifragem	AES (12,45)	DES (13,35)	DES (13,2)	RC4 (16,4)

A Tabela 2 resume os algoritmos que obtiveram os melhores desempenhos, referentes ao parâmetro de utilização de CPU, em cada um dos cenários de testes.

Tabela 3 - Melhor desempenho por cenário (parâmetro Utilização de Memória)

Cenários e Resultados – Utilização de memória					
Tamanho da mensagem		100 kb	1 Mb	100 kb	1 Mb
Número de mensagens		10	10	100	100
Algoritmos	***Hashing***	Blake (22,3)	RIPEMD (22,4)	RIPEMD (22,1)	RIPEMD (22,4)
	Cifragem	AES (22,4)	RC4 (24,1)	DES (24)	RC4 (37,1)

A Tabela 3 resume os algoritmos que obtiveram os melhores desempenhos, referentes ao parâmetro de utilização de CPU, em cada um dos cenários de testes.

4.1.2. Recomendações para cenários específicos

Tendo em conta que os algoritmos Blake2 e MD5 se destacaram cada um para tamanhos de mensagens diferentes, relativamente à utilização de CPU, é recomendado a utilização do algoritmo Blake2 para um cenário onde se utilizem dispositivos IoT que enviem quantidades

de dados menores (100kb) e possuam recursos limitados. De acordo com os resultados anteriores, o algoritmo MD5 é recomendado em cenários onde o tamanho das mensagens a serem convertidos em *hashes* sejam maiores ou iguais a 1Mb, e também com recursos limitados.

A utilização do algoritmo Blowfish não é recomendada, uma vez que obteve os priores resultados em três dos quatro cenários, e por isso, é de se evitar aplicar este algoritmo a qualquer cenário crítico que necessite de eficiência do CPU para funcionar.

No parâmetro de utilização de memória, os algoritmos RIPEMD-160 e RC4 obtiveram os melhores resultados, por isso, e tendo em conta o parâmetro em análise, podem ser recomendados para cenários de sistemas críticos, onde a baixa utilização de memória poderá proporcionar um melhor desempenho e uma resposta mais rápida, como por exemplo um sistema de refrigeração médico onde é necessário que o ambiente esteja sempre a uma certa temperatura, e caso essa temperatura baixe, seja emitido um alerta o mais rápido possível. Apesar de o algoritmo MD5 ser o que resulta numa *hash* mais pequena, a utilização de memória demonstrou fracos resultados neste parâmetro, e por isso recomenda-se a utilização do algoritmo RIPEMD-160 em quaisquer cenários semelhantes aos supracitados onde a memória seja um recurso escasso.

4.2.Resultados de Latência

Relativamente aos resultados sobre a latência, estes serão apresentados separando os algoritmos de *hashing* num gráfico e os algoritmos de cifragem noutros.

No caso dos algoritmos de *hashing*, o resultado do algoritmo é de tamanho fixo independentemente da mensagem que é convertida em *hash*. Assim, não é necessário comparar detalhadamente os diferentes valores para os diferentes tamanhos de mensagens. No entanto, apresentar-se-á a comparação entre os diferentes tamanhos fixos de cada algoritmo.

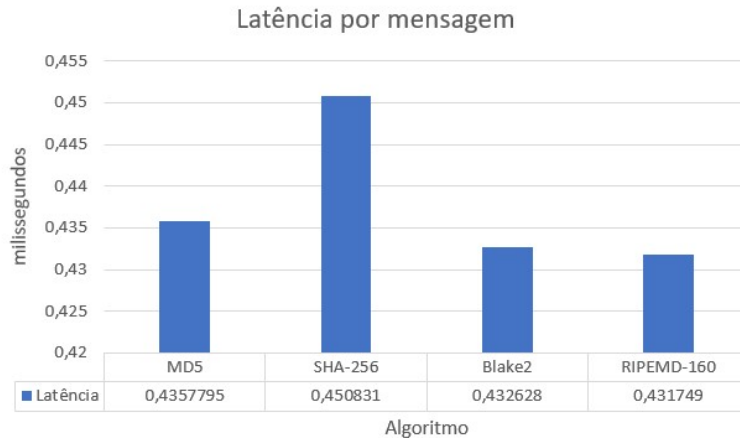


Figura 65 - Latência por mensagem (algoritmos de *hashing*)

Na Figura 65, é possível observar que no que respeita ao parâmetro da latência, o algoritmo de *hashing* que obteve a maior latência por mensagem foi o algoritmo SHA-256, cerca de 4.42% a mais do que o algoritmo RIPEMD-160, que obteve a menor latência.

De seguida são apresentados os resultados relativos ao parâmetro da latência para os algoritmos de cifragem, onde são apresentados resultados referentes aos quatro cenários de análise.

Figura 66 - Latência no envio de 10 mensagens com 100 kb (algoritmos de cifragem)

Na Figura 66, é possível verificar que para o envio de 10 mensagens com 100 kb, o algoritmo de cifragem que obteve a maior latência foi o algoritmo Blowfish, cerca de 15.25% a mais do que o algoritmo DES, que obteve a menor latência.

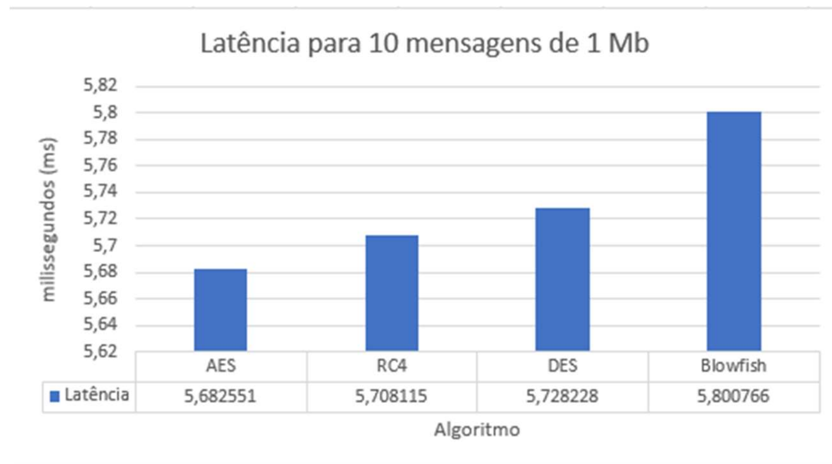


Figura 67 - Latência no envio de 10 mensagens com 1 Mb (algoritmos de cifragem)

Na Figura 67, é possível verificar que para o envio de 10 mensagens com 1 Mb, o algoritmo de cifragem que obteve a maior latência foi o algoritmo Blowfish, cerca de 2.08% a mais do que o algoritmo AES que obteve a menor latência.

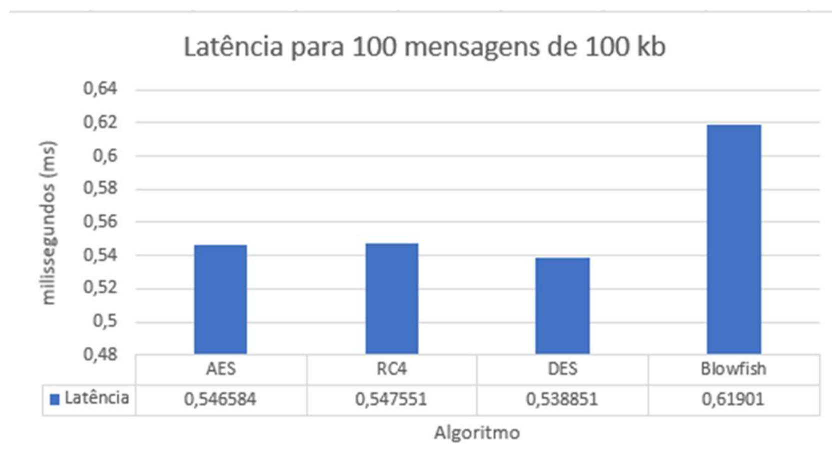


Figura 68 - Latência no envio de 100 mensagens com 100 kb (algoritmos de cifragem)

Na Figura 68, é possível verificar que para o envio de 100 mensagens com 100 kb, o algoritmo de cifragem que obteve a maior latência foi o algoritmo Blowfish, cerca de 14.87% a mais do que o algoritmo DES, que obteve a menor latência.

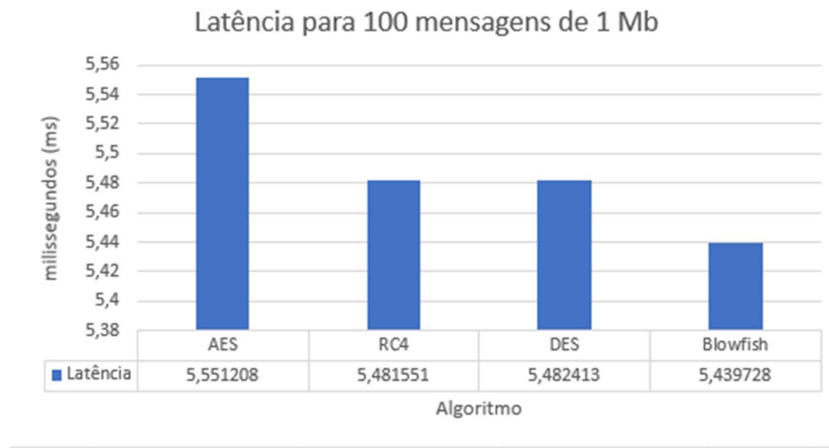


Figura 69 - Latência no envio de 100 mensagens com 1 Mb (algoritmos de cifragem)

Na Figura 69, é possível verificar que para o envio de 100 mensagens com 1 Mb, o algoritmo de cifragem que obteve a maior latência foi o algoritmo AES, cerca de 2.04% a mais do que o algoritmo Blowfish que obteve a menor latência.

4.2.1. Discussão dos resultados

Para a latência, o algoritmo de *hashing* que mais se destacou foi o algoritmo RIPEMD-160 contando com uma melhoria de 4.42% em relação ao algoritmo SHA-256, que obteve o pior desempenho.

Entre o algoritmo que obteve o melhor resultado e o algoritmo que obteve o pior resultado, é esperado que o RIPEMD-160 por produzir um *hash* de 160 *bits* obtenha melhores resultados do que o SHA 256 que produz um *hash* de 256 *bits*.

Nos algoritmos de cifragem, destacaram-se os algoritmos DES e Blowfish, para os cenários menor e maior respetivamente, obtendo melhorias de 15.23% sobre o algoritmo Blowfish no caso do DES, tornando este um excelente algoritmo de cifragem para cenários mais reduzidos, e uma melhoria de 2.04% para o algoritmo AES por parte do algoritmo Blowfish.

Tabela 4 - Melhor desempenho por cenário (parâmetro Latência)

Cenários e Resultados – Latência					
Tamanho da mensagem		100 kb	1 Mb	100 kb	1 Mb
Número de mensagens		10	10	100	100
Algoritmos	***Hashing***	RIPEMD (0,431749)			
	Cifragem	DES (0,936615)	AES (5,682551)	DES (0,538851)	Blowfish (5,439728)

A Tabela 4 resume os algoritmos que obtiveram os melhores desempenhos, referentes ao parâmetro de latência, em cada um dos cenários de testes.

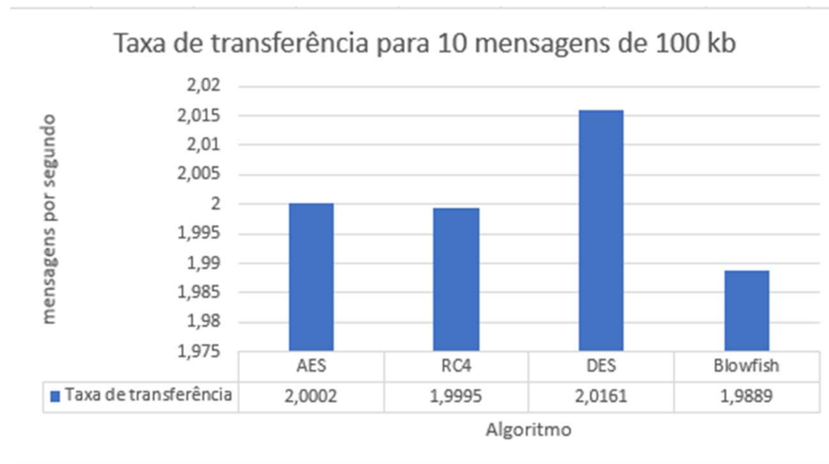
4.2.2. Recomendações para cenários específicos

Tendo em conta os resultados deste parâmetro, os algoritmos RIPEMD-160 para *hashing* e DES para cifragem, são algoritmos que podem ser mais propícios para aplicações que meçam ou apresentem resultados em tempo real, e consequentemente, apresentam também uma melhor experiência aos utilizadores.

Visto que o algoritmo RIPEMD-160 se destacou, é recomendado para um cenário de *streaming*, uma vez que estes serviços/cenários requerem uma necessidade de que o envio de mensagens seja mais rápido, e a latência é crucial para que a qualidade do serviço se mantenha. Já o algoritmo DES, que se destacou nos resultados deste parâmetro, pode ser aplicado num cenário de monitoramento de segurança ou saúde, onde convém que não existam atrasos significativos no envio das mensagens.

4.3.Resultados de Taxa de Transferência

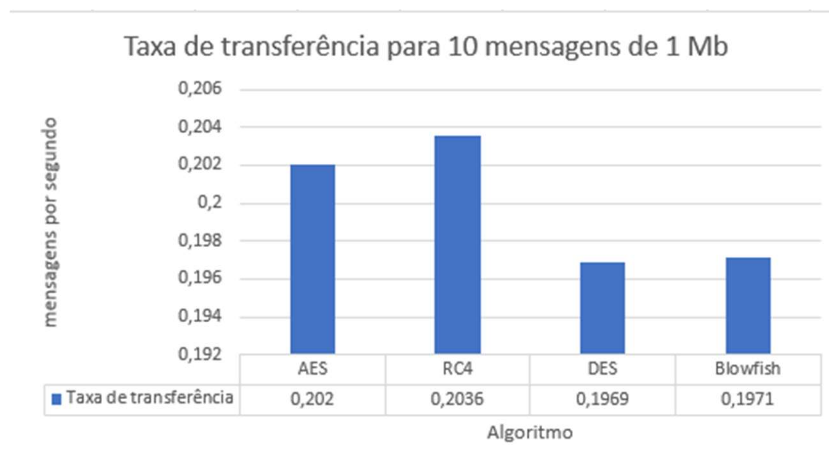
Relativamente aos resultados sobre a taxa de transferência, apenas serão apresentados os resultados referentes aos algoritmos de cifragem, uma vez que no caso dos algoritmos de *hashing*, o resultado do algoritmo é de tamanho fixo independentemente da mensagem que é convertida em *hash*. Assim, neste resultado, também não é necessário comparar detalhadamente os diferentes valores para os diferentes tamanhos de mensagens.



	AES	RC4	DES	Blowfish
Taxa de transferência	2,0002	1,9995	2,0161	1,9889

Figura 70 - Taxa de Transferência no envio de 10 mensagens com 100 kb (algoritmos de cifragem)

Na Figura 70, é possível verificar que para o envio de 10 mensagens com 100 kb, o algoritmo de cifragem que obteve a maior taxa de transferência foi o algoritmo DES, cerca de 1.36% a mais do que o algoritmo Blowfish que obteve a menor taxa de transferência.



	AES	RC4	DES	Blowfish
Taxa de transferência	0,202	0,2036	0,1969	0,1971

Figura 71 - Taxa de Transferência no envio de 10 mensagens com 1 Mb (algoritmos de cifragem)

Na Figura 71, é possível verificar que para o envio de 10 mensagens com 1 Mb, o algoritmo de cifragem que obteve a maior taxa de transferência foi o algoritmo RC4, cerca de 3.40% a mais do que o algoritmo DES que obteve a menor taxa de transferência.

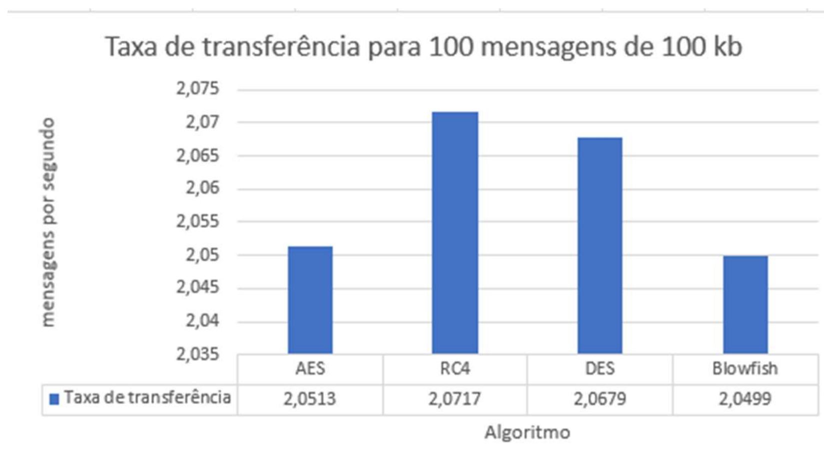


Figura 72 - Taxa de Transferência no envio de 100 mensagens com 100 kb (algoritmos de cifragem)

Na Figura 72, é possível verificar que para o envio de 100 mensagens com 100 kb, o algoritmo de cifragem que obteve a maior taxa de transferência foi o algoritmo RC4, cerca de 0.99% a mais do que o algoritmo AES que obteve a menor taxa de transferência.

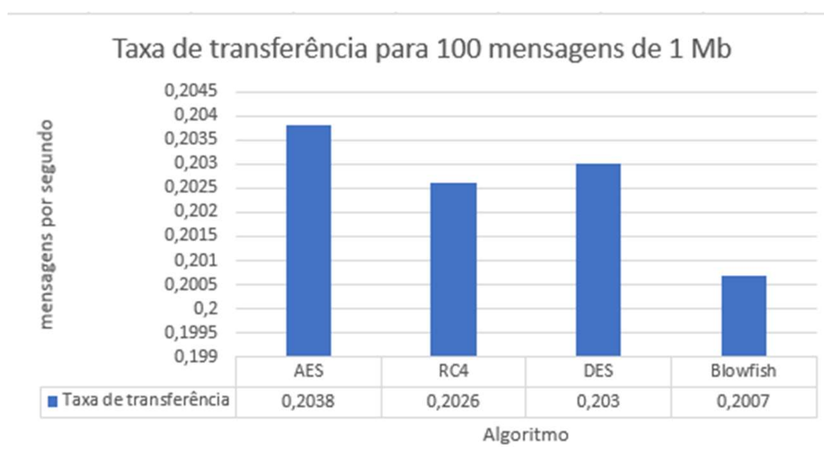


Figura 73 - Taxa de Transferência no envio de 100 mensagens com 1 Mb (algoritmos de cifragem)

Na Figura 73, é possível verificar que para o envio de 100 mensagens com 1 Mb, o algoritmo de cifragem que obteve a maior taxa de transferência foi o algoritmo AES, cerca de 1.54% a mais do que o algoritmo Blowfish que obteve a menor taxa de transferência.

4.3.1. Discussão dos resultados

O algoritmo Blowfish destacou-se em todos os cenários, tendo obtido resultados 1.36% e 1.54% inferiores do que os algoritmos de cifragem que obtiveram os maiores resultados neste parâmetro.

Tabela 5 - Melhor desempenho por cenário (parâmetro Taxa de Transferência)

Cenários e Resultados – Taxa de Transferência				
Tamanho da mensagem	100 kb	1 Mb	100 kb	1 Mb
Número de mensagens	10	10	100	100
Algoritmos Cifragem	DES (2,0161)	RC4 (0,2036)	RC4 (2,0717)	AES (0,2038)

A Tabela 5 resume os algoritmos que obtiveram os melhores desempenhos, referentes ao parâmetro da taxa de transferência, em cada um dos cenários de testes.

4.3.2. Recomendações para cenários específicos

Pela sua baixa taxa de transferência em todos os cenários, o algoritmo Blowfish não recomendado para cenários onde se pretende um processamento do algoritmo e respetivo envio mais rápido, uma vez que envia menos mensagens por segundo.

4.4. Identificação de Tendências e Padrões

Neste subcapítulo são apuradas todas as tendências e padrões encontrados nos resultados, agora atendendo a todos os resultados de todos os cenários de testes.

Para o parâmetro de utilização de CPU, foram encontrados três padrões. Nos algoritmos de *hashing*, o algoritmo Blake2 foi o algoritmo com melhor desempenho nos cenários que envolviam mensagens menores (100 kb), e o algoritmo MD5 para os cenários que envolviam mensagens maiores (1 Mb). Nos algoritmos de cifragem foi possível apurar que o algoritmo de cifragem Blowfish obteve os piores resultados em três dos quatro cenários.

Relativamente ao parâmetro de utilização de memória, foi possível apurar dois padrões para os algoritmos de *hashing* e um para os algoritmos de cifragem. Nos algoritmos de *hashing*, o algoritmo RIPEMD-160 destacou-se em todos os parâmetros com exceção do cenário que envolvia mensagens menores (10 mensagens com 100 kb), onde foi o segundo algoritmo com melhor desempenho. Foi possível também apurar que o algoritmo de *hashing* MD5 obteve os piores resultados em três dos quatro cenários. Nos algoritmos de cifragem, destacou-se o algoritmo RC4, assumindo o melhor desempenho em todos os resultados, com exceção, novamente, do cenário que envolviam mensagens menores (10 mensagens com 100

kb), e do cenário de 100 mensagens com 100 kb, onde, neste último, todos os algoritmos tiveram um desempenho relativamente igual.

Sobre o parâmetro latência, foi possível apurar um padrão relativo aos algoritmos de cifragem. Onde o algoritmo Blowfish apresentou os resultados mais elevados em três dos quatro cenários, com exceção do cenário de 100 mensagens de 1 Mb.

Para o parâmetro da taxa de transferência, o algoritmo de cifragem Blowfish apresentou os piores resultados em três dos quatro cenários, com exceção do cenário de 10 mensagens com 1 Mb.

4.5.Análise crítica dos Cenários de Teste e Resultados

Relativamente aos cenários de teste, foram especificados o tamanho das mensagens e o número de mensagens a enviar, os quais foram definidos para tentar simular um ambiente IoT com dispositivos de fracos recursos onde, por exemplo, numa horta/quinta com sensores que enviam dados como temperatura, humidade, entre outros, as mensagens não são enviadas muitas mensagens, e estas são normalmente também não são muito grandes.

Em muitas situações a diferença entre os algoritmos foi mínima, mas no longo prazo de utilização, o cuidado a ter com estes resultados, pode trazer resultados benéficos em termos de consumo de energia, rapidez no envio de mensagens, ou sobre qualquer aspeto que seja consequência dos parâmetros em análise.

De frisar que os ambientes e dispositivos utilizados nos artigos selecionados na base teórica, e que agora estão a ser comparados com os resultados obtidos neste trabalho, não são os mesmos aqui utilizados, e por isso, há fatores que influenciam os resultados obtidos em ambos os lados, quer sejam a nível do *software* ou do *hardware*, ou até mesmo das configurações realizadas. Estas variações não invalidam qualquer trabalho (artigos da base teórica ou o presente trabalho), mas é necessário dar a entender que existem estas diferenças.

Neste projeto foi utilizada a linguagem de programação Python, que não é tão favorável em termos de eficiência e desempenho como outras linguagens bastantes conhecidas como C, e foi utilizado um Raspberry Pi 3, que pode ser melhor (no caso de outro trabalho utilizar versões anteriores do Raspberry Pi ou *hardware* menos evoluído) ou pior (no caso de outro trabalho utilizar versões posteriores ou *hardware* mais evoluído) do que qualquer outro dos muitos dispositivos de *hardware* para IoT que existem.

5. Conclusões e trabalho futuro

O objetivo principal deste projeto era realizar um estudo para comparar o desempenho de dispositivos IoT na utilização de algoritmos de segurança, isto através da medição de parâmetros de desempenho como a utilização de processador, memória, latência e taxa de transferência, tendo sido neste caso em específico utilizado um Raspberry Pi.

Este projeto começou com uma explicação sobre a Internet das Coisas e com alguns conceitos de segurança, desde a arquitetura de um ambiente IoT até as ameaças e soluções encontradas no mesmo. Foram também apresentados os dois protocolos de comunicação utlizados em soluções IoT, as suas arquiteturas e características, nomeadamente o MQTT e o CoAP, tendo sido também feita uma breve explicação sobre outros protocolos utilizados em IoT. O MQTT foi o protocolo de comunicação escolhido para a realização dos testes, tendo todo o cenário de testes e *software* desenvolvido sido adaptados para o mesmo.

Foram também detalhados os conceitos de *hashing* e de cifragem, e como cada um se aplica à segurança, sendo que o primeiro permite validar a integridade das mensagens através da geração de uma *hash* de comparação, e o segundo permite garantir a confidencialidade das mensagens, escondendo o conteúdo original. Relativamente aos algoritmos utilizados neste projeto, foram utlizados os algoritmos de *hashing* MD5, SHA-256, Blake2 e RIPEMD-160, e os algoritmos de cifragem AES, RC4, DES e Blowfish. Foi também apresentada uma explicação detalhada sobre cada um, visando em aspetos como a história do algoritmo, o tamanho do *hash* ou da chave de cifragem que cada dispositivo gera, a complexidade do algoritmo e a sua segurança de forma geral.

Relativamente à metodologia escolhida, o ambiente de testes definido recorreu à utilização de um Raspberry Pi Model B V1.2 e duas máquinas virtuais para simular a interação de um dispositivo IoT com um *broker* MQTT. Foram detalhadas todas as configurações que permitiram que o ambiente estivesse pronto a receber e visualizar as mensagens enviadas pelo Raspberry Pi, e o *script* padrão que realizava todas as medições dos parâmetros em análise, e de que forma este funcionava.

Por último, foram apresentados os resultados obtidos na realização de testes em quatro cenários diferentes, onde variaram parâmetros como o número de mensagens e o tamanho

de cada mensagem, valores esses que foram definidos para simular o envio de um dispositivo IoT real.

Para cada conjunto de resultados, foi realizada uma comparação entre cada cenário, tendo sido feita uma análise entre o algoritmo que obteve o melhor e o pior desempenho, agrupando estas comparações entre os dois grupos de algoritmos em análise (*hashing* e cifragem).

Os cenários reais IoT atribuídos aos algoritmos que obtiveram os melhores resultados passam por cenários com dispositivos com recursos limitados que enviem mensagens curtas onde se recomenda a utilização do algoritmo Blake2 e cenários com dispositivos com recursos limitados que enviem mensagens longas onde se recomenda a utilização do algoritmo MD5 relativamente ao parâmetro utilização de CPU.

É recomenda a utilização do algoritmo de *hashing* RIPEMD-160 e do algoritmo de cifragem RC4, relativamente ao parâmetro utilização de memória, em cenários de sistemas críticos como por exemplo, cenários de emergência.

Relativamente ao parâmetro da latência, este estudo recomenda a utilização do algoritmo de *hashing* RIPEMD-160 e do algoritmo de cifragem DES para cenários de serviços de *streaming*/video e cenários monitoramento de segurança.

Com a realização deste projeto foi possível compreender melhor como funciona o ambiente IoT, os protocolos de comunicação que sobre ele operam e como é que se lhe pode implementar segurança. Foi uma mais-valia também para a prática realizada uma vez que se aprendeu a realizar um projeto mais complexo do que o que estava habituado, desde a idealização do mesmo, passando pela pesquisa que serviria de base teórica e definição da metodologia, até ao desenvolvimento de *software*, recolha de dados e análise de resultados.

Como trabalho futuro, seria interessante utilizar outros modelos de Raspberry Pi e outro tipo de dispositivos IoT, nomeadamente Arduinos, o que iria acrescentar mais complexidade ao cenário de testes e fornecer muito provavelmente resultados diferentes de equipamento para equipamento, permitindo assim ter uma maior perceção das capacidades e limitações de cada dispositivo.

Referências Bibliográficas

[1] "Aws.Amazon," Amazon, [Online]. Available: https://aws.amazon.com/pt/what-is/iot/.

[2] Tecmundo, "Afinal, o que é a Lei de Moore," Tecmundo, [Online]. Available: https://www.tecmundo.com.br/curiosidade/701-o-que-e-a-lei-de-moore-.htm.

[3] R. F. A. P. D. Oliveira, "Exemplo de uma arquitetura IoT em uma residencia," researchgate, 7 2017. [Online]. Available: https://www.researchgate.net/figure/Figura-3-Exemplo-de-uma-arquitetura-IoT-em-uma-residencia-Fonte-Acessado-em_fig4_318962439.

[4] IoTWorldToday, "IoT Cyberattacks Escalate in 2021, According to Kaspersky," IoTWorldToday, 17 09 2021. [Online]. Available: https://www.iotworldtoday.com/2021/09/17/iot-cyberattacks-escalate-in-2021-according-to-kaspersky/.

[5] K. E. S. &. R. T. Samuel Tweneboah-Koduah, "Cyber Security Threads to IoT Applications and Service Domains," SpringerLink, 2017 Maio 26. [Online]. Available: https://link.springer.com/article/10.1007/s11277-017-4434-6.

[6] Microsoft, "padrão de Publisher-Subscriber," Microsoft, [Online]. Available: https://learn.microsoft.com/pt-pt/azure/architecture/patterns/publisher-subscriber.

[7] R. Patil, "HTTP Request, HTTP Response, Context and Headers : Part III.," https://medium.com/, 29 5 2020. [Online]. Available: https://medium.com/@rohitpatil97/http-request-http-response-context-and-headers-part-iii-5c37bd4cb06b.

[8] Microsoft, "Padrão de Request-Reply assíncrono," Microsoft, [Online]. Available: https://learn.microsoft.com/pt-pt/azure/architecture/patterns/async-request-reply.

[9] mqtt.org, "MQTT: The Standard for IoT Messaging," mqtt.org, [Online]. Available: https://mqtt.org/.

[10] alfacomp, "O que é o Protocolo MQTT?," alfacomp, [Online]. Available: https://alfacomp.net/2022/03/24/mqtt-conheca-o-protocolo-padrao-do-iot/.

[11] C. G.-R. F. A. C. C. Victor Seoane, "Performance evaluation of CoAP and MQTT with security support for IoT environments," vol. 197, 9 10 2021.

[12] M. Rouse, “Three-Way Handshake,” techopedia, 10 11 2020. [Online]. Available: https://www.techopedia.com/definition/10339/three-way-handshake.

[13] Google, “MQTTLens,” chrome web store, [Online]. Available: https://chrome.google.com/webstore/detail/mqttlens/hemojaaeigabkbcookmlgmdigohjobjm.

[14] F. Azzola, “CoAP Protocol: Step-by-Step Guide,” dzone, 8 Novembro 2018. [Online]. Available: https://dzone.com/articles/coap-protocol-step-by-step-guide.

[15] wallarm, “What is CoAP Protocol? Meaning & Architecture,” wallarm, [Online]. Available: https://www.wallarm.com/what/coap-protocol-definition.

[16] F. F. D. C. M. D. N. ARYANE BARROS MACIEL DA SILVA, “Constrained Application Protocol,” gta.ufrj, 1 2019. [Online]. Available: https://www.gta.ufrj.br/ensino/eel878/redes1-2019-1/vf/coap/funcionamento.html.

[17] https://eclipse.dev/, “Eclipse Californium,” https://eclipse.dev/, [Online]. Available: https://eclipse.dev/californium/.

[18] Fromdev, “Top 5 Open Source AMQP (Advanced Message Queing Protocol) Implementations,” Fromdev, 8 Abril 2012. [Online]. Available: https://www.fromdev.com/2012/04/top-5-open-source-amqp-advanced-message.html.

[19] xmpp.org, “Internet of Things (IoT),” xmpp.org, [Online]. Available: https://xmpp.org/uses/internet-of-things/.

[20] XMPP-IoT, “XMPP Internet of Things,” 2017. [Online]. Available: http://www.xmpp-iot.org/.

[21] W. Stallings, Cryptography and Network Security Principles and Practice, 2017.

[22] KUSHAGRA7744, “APLICAÇÕES DE HASHING,” acervolima, [Online]. Available: https://acervolima.com/aplicacoes-de-hashing/.

[23] J. a. H. H. M. a. V. R. Deepakumara, “FPGA implementation of MD5 hash algorithm,” em *Canadian Conference on Electrical and Computer Engineering 2001. Conference Proceedings (Cat. No. 01TH8555)*, 2001, pp. 919-924.

[24] Avast, “What Is the MD5 Hashing Algorithm and How Does It Work?,” Avast, 7 Janeiro 2022. [Online]. Available: https://www.avast.com/c-md5-hashing-algorithm.

[25] N-able, “SHA-256 Algorithm Overview,” 12 9 2019. [Online].

[26] S. a. W. T. binti Suhaili, “Design of high-throughput SHA-256 hash function based on FPGA,” em *2017 6th International Conference on Electrical Engineering and Informatics (ICEEI)*, 2017, pp. 1--6.

[27] T. H. a. P. H. L. a. N. Y. Tran, “A high-performance multimem SHA-256 accelerator for society 5.0,” *IEEE Access,* vol. 9, pp. 39182--39192, 2021.

[28] M. Lepcha, “SHA-256,” Techopedia, 26 7 2023. [Online]. Available: https://www.techopedia.com/definition/sha-256.

[29] H. B. S. d. S. E. &. R. U. L. Heiderscheidt, “A INFLUÊNCIA DA ENTROPIA DAS SENHAS NA DEFESA CONTRA ATAQUES DE FORÇA BRUTA,” *Revista de Extensão e Iniciação Científica da UNISOCIESC,* vol. 10, nº 1, 27 12 2022.

[30] blake2.net, “BLAKE2 — fast secure hashing,” 22 2 2017. [Online]. Available: https://www.blake2.net/.

[31] J. S. Narayan. [Online]. Available: https://gist.github.com/sooryan/8d1b2c19bf0b971c11366b0680908d4b.

[32] L. a. A. J.-P. a. M. W. a. P. R. C.-W. Henzen, “VLSI characterization of the cryptographic hash function BLAKE,” *IEEE transactions on very large scale integration (vlsi) systems,* vol. 19, nº 10, pp. 1746--1754, 2010.

[33] S. a. D. C. a. Z. W. a. Z. J. a. J. H. a. M. B. a. Z. L. Wu, “EaD: ECC-Assisted Deduplication With High Performance and Low Memory Overhead for Ultra-Low Latency Flash Storage,” *IEEE Transactions on Computers,* vol. 72, nº 1, pp. 208--221, 2022.

[34] gluttony777, “RIPEMD Hash Function,” 10 2 2020. [Online]. Available: https://www.geeksforgeeks.org/ripemd-hash-function/.

[35] B. a. B. A. a. D. H. Preneel, *The cryptographic hash function RIPEMD-160,* 1997.

[36] M. J. G. Antunes, “PowerPoint utilizado em aula da Unidade Curricular de Segurança em Redes de Computadores,” 2022.

[37] ssl2buy, “Symmetric vs. Asymmetric Encryption – What are differences?,” ssl2buy, [Online]. Available: https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences.

[38] T. Monteiro, “Criptografia no Telegram,” jusbrasil, 5 2023. [Online]. Available: https://www.jusbrasil.com.br/artigos/criptografia-no-telegram/1824969671.

[39] F. N. NIFE, “A New Modified Version of Caser Cipher Algorithm,” *International Journal of Engineering Research and Development ,* vol. 11, pp. 38-43, 2015.

[40] F. Masoodi, “Symmetric Algorithms I,” em *Emerging Security Algorithms and Techniques*, 2019, pp. 79-95.

[41] A. M. a. o. Abdullah, “Advanced encryption standard (AES) algorithm to encrypt and decrypt data,” *Cryptography and Network Security,* vol. 16, nº 1, p. 11, 2017.

[42] S. D. a. K. A. a. O. S. E. F. Rihan, “A performance comparison of encryption algorithms AES and DES,” *International Journal of Engineering Research \& Technology (IJERT),* vol. 4, nº 12, pp. 151-154, 2015.

[43] Okta, “RC4 Encryption Algorithm Stream Ciphers Defined,” Okta, 14 02 2023. [Online]. Available: https://www.okta.com/identity-101/rc4-stream-cipher/.

[44] P. a. K. P. Prasithsangaree, “Analysis of energy consumption of RC4 and AES algorithms in wireless LANs,” em *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, 2003, pp. 1445--1449.

[45] S. a. P. A. Palma, “Análise Crítica da Implementação da Cifra RC4”.

[46] S. a. R. E. Pavithra, “Throughput analysis of symmetric algorithms,” *International Journal of Advanced Networking and Applications,* vol. 4, nº 2, p. 1574, 2012.

[47] S. D. a. K. A. a. O. S. E. F. Rihan, “A performance comparison of encryption algorithms AES and DES,” *International Journal of Engineering Research \& Technology (IJERT),* vol. 4, nº 12, pp. 151--154, 2015.

[48] A. Einorytè, “What is Blowfish encryption, and how does it work?,” NordVPN, 26 7 2023. [Online]. Available: https://nordvpn.com/pt/blog/what-is-blowfish-encryption/.

[49] C. a. K. S. Haldankar, “Implementation of AES and blowfish algorithm,” *International Journal of Research in Engineering and Technology,* vol. 3, nº 03, pp. 143--146, 2014.

[50] K. Patel, “Performance analysis of AES, DES and Blowfish cryptographic algorithms on small and large data files,” *International Journal of Information Technology,* vol. 11, pp. 813--819, 2019.

[51] A. Szczepaniak, “leorover,” leorover, 15 5 2023. [Online]. Available: https://www.leorover.tech/post/raspberry-pi-or-arduino-when-to-choose-which.

[52] raspberrypi, “Raspberry Pi OS,” raspberrypi, [Online]. Available: https://www.raspberrypi.com/software/.

[53] mosquitto.org, “Eclipse Mosquitto™ An open source MQTT broker,” mosquitto.org, [Online]. Available: https://mosquitto.org/.

[54] docs.python, “hashlib - Secure hashes and message digests,” docs.python, [Online]. Available: https://docs.python.org/3/library/hashlib.html.

[55] SNIGDHA_YAMBADWAR, “O QUE É CRIPTOGRAFIA RC4?,” acervolima, [Online]. Available: https://acervolima.com/o-que-e-criptografia-rc4/.