

Universidade da Beira Interior

Departamento de Informática



Departamento de
Informática

Nº 32 - 2019: *Comparação, uso e aplicação de algoritmos criptográficos para IoT*

Elaborado por:

Daniel António Ferraz Saraiva

Orientador:

Professor Doutor Valderi Reis Quietinho Leithardt

22 de Junho de 2019

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao Professor Valderi Leithardt por ter orientado este projeto. O seu apoio, disponibilidade para ajudar e conhecimentos transmitidos foram imprescindíveis para a realização deste trabalho.

Agradeço também aos meus pais António Saraiva e Gabriela Ferraz, bem como à minha irmã Carolina Saraiva, por terem tornado possível o meu percurso académico, além de todo o carinho, apoio e incentivo prestados ao longo da minha vida.

Deixo um agradecimento especial à Diandre de Paula por todo o carinho, amizade, por ter estado sempre do meu lado desde o início e por todos os momentos que passámos juntos dentro e fora da Universidade, momentos esses que nunca esquecerei. Todo o meu percurso e principalmente estes últimos tempos não teriam sido os mesmos sem a sua presença.

Ao meu grande amigo Gonçalo Mêda, um muito obrigado por estar sempre do meu lado desde que iniciei o meu percurso académico e por todos os trabalhos que desenvolvemos juntos. Agradeço-lhe principalmente por todos os apoios que me prestou a nível pessoal, por ter estado comigo em todos os momentos e por todas as saídas que me forneceram dos melhores momentos nestes últimos anos.

Também não podia deixar de agradecer à Joana Farias e à Marta Caixinha por toda a amizade, companhia e apoio prestado neste último ano. Dificilmente esquecerei as tardes passadas no bar da 6ª fase com estas minhas amigas e que tanto me ajudaram a descontraír e a animar nos momentos menos bons.

Conteúdo

Conteúdo	iii
Lista de Figuras	vii
Lista de Tabelas	ix
Lista de Excertos de Código	xi
1 Introdução	1
1.1 Enquadramento	1
1.2 Motivação	1
1.3 Objetivos	2
1.4 Organização do Documento	3
2 Estado da Arte	5
2.1 Introdução	5
2.2 Trabalhos Relacionados	5
2.3 Conclusões	7
3 Tecnologias e Ferramentas Utilizadas	9
3.1 Introdução	9
3.2 Descrição das Tecnologias Utilizadas	9
3.3 Descrição das Ferramentas Utilizadas	10
3.4 Conclusões	11
4 Engenharia de Software	13
4.1 Introdução	13
4.2 Especificação de Requisitos do Servidor <i>Ubiquitous Privacy</i> (UbiPri)	13
4.2.1 Requisitos Funcionais	13
4.2.2 Requisitos Não Funcionais	14
4.3 Especificação de Requisitos da Aplicação UbiPri	14
4.3.1 Requisitos Funcionais	15

4.3.2	Requisitos Não Funcionais	15
4.4	Conclusões	15
5	Implementação e Testes dos Algoritmos Criptográficos	17
5.1	Introdução	17
5.2	Implementação dos Testes de Criptografia	17
5.2.1	Geração do Pacote	19
5.2.2	Geração das Chaves Simétricas e Vetores de Inicialização	20
5.2.3	Encriptação e Desencriptação do Pacote	20
5.3	Resultados dos Testes	22
5.4	Conclusões	24
6	Implementação do Módulo de Segurança no Servidor UbiPri	25
6.1	Introdução	25
6.2	Organização das Fontes do Servidor	25
6.2.1	<i>Package server</i>	25
6.2.2	<i>Package server.dao</i>	25
6.2.3	<i>Package server.model</i>	26
6.2.4	<i>Package server.model.classify</i>	26
6.2.5	<i>Package server.modules.communication</i>	26
6.2.6	<i>Package server.modules.communication.rest</i>	26
6.2.7	<i>Package server.modules.crypto</i>	26
6.2.8	<i>Package server.modules.privacy</i>	26
6.2.9	<i>Package server.util</i>	27
6.3	Habilitação do protocolo <i>Hypertext Transfer Protocol Secure</i> (HTTPS)	27
6.4	Atualizações Feitas aos Métodos do Servidor	28
6.4.1	Proteção das <i>Passwords</i> e Geração de <i>Tokens</i>	28
6.4.2	Alterações nos Métodos da Classe <i>PrivacyControlUbiquitous</i>	32
6.4.3	Alterações nos Métodos <i>Representational State Transfer</i> (REST)	35
6.5	Conclusões	38
7	Alterações à Aplicação <i>Android</i> do UbiPri	39
7.1	Introdução	39
7.2	Estrutura da Aplicação	39
7.2.1	<i>Package br.ufrgs.inf.ubipri.util</i>	39
7.2.2	<i>Package br.ufrgs.inf.ubipri.client</i>	40
7.2.3	<i>Package br.ufrgs.inf.ubipri.client.model</i>	40
7.2.4	<i>Package br.ufrgs.inf.ubipri.client.dao</i>	40
7.2.5	<i>Package br.ufrgs.inf.ubipri.client.communication</i>	40

7.3	Implementação da Atividade de Registo	40
7.4	Implementação da Classe <i>CryptoServices</i>	41
7.5	Alterações à Atividade de <i>Login</i>	43
7.6	Alterações à Atividade da Localização	44
7.7	Alterações à Atividade do Menu Principal	45
7.8	Implementação do Serviço <i>LocationService</i>	45
7.9	Implementação das Tarefas Assíncronas	47
7.10	Conclusões	48
8	Conclusões e Trabalho Futuro	49
8.1	Conclusões Principais	49
8.2	Trabalho Futuro	51
	Bibliografia	53

Lista de Figuras

1.1	Módulos de privacidade do UbiPri, Leithardt <i>et al.</i> [13].	3
2.1	Número de transmissões pelo tempo de execução dos algoritmos criptográficos testados, Ochôa <i>et al.</i> [16].	6
2.2	Tempo (segundos) e consumo energético (<i>milijoule</i>) para a geração de chaves dos algoritmos <i>Rivest-Shamir-Adleman</i> (RSA) e <i>Elliptic-Curve Diffie-Hellman</i> (ECDH), El-Hajj <i>et al.</i> [9].	7
5.1	Exemplo de uma execução do programa desenvolvido com nível de acesso <i>Advanced</i> e um pacote com tamanho de 10 <i>megabytes</i> . .	19
7.1	Atividade do registo de um utilizador.	41

Lista de Tabelas

2.1	Comparação de alguns dos estudos feitos com este projeto.	8
5.1	Tempo (segundos) de encriptação para o processador Intel® Core™ .	22
5.2	Tempo (segundos) de descriptação para o processador Intel® Core™	22
5.3	Tempo (segundos) de encriptação para o processador <i>Cortex-A15</i> emulado.	23
5.4	Tempo (segundos) de descriptação para o processador <i>Cortex-A15</i> emulado.	24

Lista de Excertos de Código

5.1	Trecho de código da geração de um pacote aleatório.	19
5.2	Trecho de código da geração de chaves simétricas e <i>Initialization Vector</i> (IV).	20
5.3	Trecho de código da encriptação do pacote.	21
6.1	Linhas de código adicionadas à configuração do servidor para suportar HTTPS.	27
6.2	Trecho de código da leitura de uma chave pública.	29
6.3	Trecho de código do método que gera um <i>salt</i>	29
6.4	Trecho de código do método que gera um <i>token</i>	30
6.5	Trecho de código do método que valida um <i>token</i>	30
6.6	Trecho de código do método que cria um novo processo.	31
6.7	Trecho de código do método que verifica se um utilizador existe. .	32
6.8	Trecho de código do método que obtém o <i>salt</i> de um utilizador da base de dados.	32
6.9	Trecho de código do método que verifica o <i>login</i>	33
6.10	Trecho de código do método que regista um utilizador.	34
6.11	Trecho de código do método REST que verifica o <i>login</i>	35
6.12	Trecho de código do método REST que regista um usuário.	36
6.13	Trecho de código do método que trata das ações para um dado ambiente.	36
6.14	Trecho de código dos métodos auxiliares.	37
7.1	Trecho de código do método <code>initKeyStore()</code>	42
7.2	Trecho de código do método <code>encrypt()</code>	43
7.3	Trecho de código do método <code>login()</code>	44
7.4	Trecho de código do método <code>onResume()</code> do menu principal. . .	45
7.5	Trecho de código do método <code>onStartCommand()</code>	46
7.6	Trecho de código do método <code>onDestroy()</code>	47
7.7	Trecho de código do método <code>doInBackground()</code> da tarefa do <i>login</i> . 47	

Acrónimos

IoT *Internet of Things*

3DES *Triple DES*

AES *Advanced Encryption Standard*

AES-NI *Advanced Encryption Standard New Instructions*

API *Application Programming Interface*

CCM *Counter with CBC-MAC*

CTR *Counter*

DDoS *Distributed Denial of Service*

DoS *Denial of Service*

DSA *Digital Signature Algorithm*

EAX *Encrypt then Authenticate then Translate*

ECC *Elliptic-Curve Cryptography*

ECDH *Elliptic-Curve Diffie-Hellman*

ECDSA *Elliptic-Curve Digital Signature Algorithm*

GCM *Galois/Counter Mode*

GDPR *General Data Protection Regulation*

GPS *Global Positioning System*

GPU *Graphics Processing Unit*

HTTP *Hypertext Transfer Protocol*

HTTPS *Hypertext Transfer Protocol Secure*

IDE *Integrated Development Environment*

IV *Initialization Vector*

JDK *Java Development Kit*

JKS *Java KeyStore*

JSON *JavaScript Object Notation*

JWT *JSON Web Token*

MAC *Message Authentication Code*

NDN *Named Data Networking*

RC6 *Rivest Cipher 6*

REST *Representational State Transfer*

RSA *Rivest-Shamir-Adleman*

SHA *Secure Hash Algorithm*

TLS *Transport Layer Security*

UbiPri *Ubiquitous Privacy*

URI *Uniform Resource Identifier*

XML *Extensible Markup Language*

Glossário

C É uma linguagem de programação imperativa e de propósito geral, criada em 1972 por Dennis Ritchie.. 9, 10, 19, 31

C++ É uma linguagem de programação orientada a objetos baseada na linguagem C, criada em 1983 por Bjarne Stroustrup.. 9, 10, 17

C# Lê-se *C Sharp* e é uma linguagem de programação orientada a objectos, desenvolvida pela Microsoft, inicialmente para a plataforma .NET. O C# é inspirado na junção entre as linguagens C++ e Java.. 23

Java É uma linguagem de programação orientada a objectos, desenvolvida pela Sun Microsystems na década de 90. Hoje pertence à empresa Oracle.. 9, 10, 23, 27, 29, 31

middleware Software que se encontra entre o sistema operativo e as aplicações a correr nele. É normalmente utilizado em aplicações distribuídas.. 2, 3, 15, 17, 49–51

smartphone Smartphone é um telefone móvel que contém muitas das principais tecnologias de comunicação e serviços que existem nos computadores pessoais, como acesso a e-mails, serviços de mensagens instantâneas, internet, GPS, entre outros.. 5

Web Services Aplicações modulares auto-descritas e auto-contidas, que permitem a integração de sistemas e a comunicação entre aplicações de diferentes tipos.. 10

Capítulo 1

Introdução

1.1 Enquadramento

O tema deste projeto é a proteção de dados para a *Internet of Things* (IoT). Na IoT, são normalmente usados dispositivos com um poder computacional limitado, o que pode tornar inviável a utilização de protocolos de segurança mais robustos devido à redução na sua eficiência. Torna-se, então, importante analisar os algoritmos criptográficos mais apropriados para estes dispositivos, de maneira a garantir a segurança dos dados enquanto são poupados o maior número de recursos computacionais possíveis.

1.2 Motivação

Nos últimos tempos, a preocupação com privacidade dos dados e criptografia tem vindo a crescer, principalmente com o crescimento exponencial da IoT com cada vez mais dispositivos conectados à *internet*. Num estudo realizado em [19], verificou-se que a maioria dos dispositivos da IoT como *Smart TVs*, *webcams* e impressoras continham vulnerabilidades alarmantes que colocam em risco a segurança e privacidade dos seus utilizadores, o que mostra a maneira como os seus fabricantes ainda prestam pouca atenção à segurança dos seus produtos.

A introdução das normas *General Data Protection Regulation* (GDPR) [5] vieram a mudar a forma como empresas abordam a privacidade dos seus utilizadores na União Europeia, incentivando à pesquisa de novos métodos de proteção de privacidade [15]. Além disso, com cada vez mais informação a ser trocada na rede, torna-se crucial filtrar a informação de interesse. No entanto, estes filtros podem colocar em causa a privacidade dos dados a serem trocados. No estudo feito em [21] desenvolveu-se um algoritmo de *data merging* eficiente sobre informação encriptada, com o principal objetivo de garantir a privacidade e confidencialidade

dos dados.

Novas arquiteturas para a *internet*, como a *Named Data Networking* (NDN), também têm sido alvos de estudo [17]. Modelos e cenários de ataque foram desenvolvidos, colocando em risco a privacidade dos utilizadores, bem como deixando estas arquiteturas suscetíveis a ataques *Denial of Service* (DoS) e *Distributed Denial of Service* (DDoS). Torna-se, então, imperativo estudar este tipo de ataques e como melhor proteger os sistemas informáticos contra eles.

Outra forma de resolver problemas relacionados com privacidade é recorrendo à computação ubíqua, de forma a automatizar os processos de comunicação entre utilizadores e dispositivos. Para tal, foi desenvolvido em [12] o *middleware Ubiquitous Privacy* (UbiPri) para gerir e controlar a privacidade dos utilizadores em ambientes ubíquos de maneira automática, com foco no tipo de ambiente em que os usuários se encontram.

Como tal, a principal motivação para a realização deste trabalho foi investigar novas formas de garantir a segurança e privacidade dos dados em dispositivos para a IoT. Além disso, pretendeu-se aprofundar os conhecimentos básicos de segurança informática adquiridos ao longo da Licenciatura.

1.3 Objetivos

O primeiro passo deste projeto foi analisar o desempenho de diferentes algoritmos criptográficos de chave simétrica, com o objetivo de comparar a sua eficiência em relação ao *Advanced Encryption Standard* (AES). Como dispositivos IoT têm, no geral, poder computacional e recursos energéticos limitados, e tendo em conta que algoritmos de criptografia requerem bastantes destes recursos, é importante optar pelas soluções mais eficientes enquanto que é mantida a segurança dos dados.

Desta forma, começou-se por analisar quais são os algoritmos de chave simétrica *state of the art* para serem testados. Estando esta análise feita, procedeu-se ao desenvolvimento de um programa que utilizasse implementações destes algoritmos para medir o seu tempo de execução em diferentes ambientes de teste.

Com o desempenho dos algoritmos testados avaliado, fez-se um estudo do *middleware* UbiPri desenvolvido em [12]. Este *middleware* encontra-se dividido em vários módulos, cada um responsável por gerir a privacidade de diferentes aspetos do UbiPri. Os módulos encontram-se definidos em [13] e podem ser visualizados na figura 1.1. Como exemplo, o módulo *PRICOM* gere a privacidade das comunicações, enquanto que o módulo *PRIDEV* controla a privacidade dos dispositivos. No âmbito deste projeto decidiu-se desenvolver o módulo *PRISEC*, que é responsável por gerir a criptografia e segurança de todos os dados do *middleware*.

Atualmente, o UbiPri é composto por um servidor onde se encontram armazenados os dados referentes aos seus utilizadores e ambientes. Além disso, o

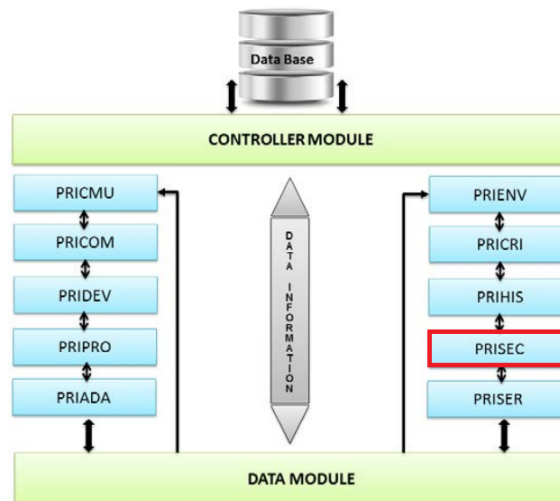


Figura 1.1: Módulos de privacidade do UbiPri, Leithardt *et al.* [13].

servidor tem implementados os métodos necessários ao funcionamento do *middleware*, desde autenticação dos seus utilizadores até ao tratamento dos níveis de acesso e privacidade dos usuários em cada um dos ambientes suportados. Os utilizadores interagem com o servidor do UbiPri através de uma aplicação *Android*, que já tinha implementada algumas das funcionalidades básicas do *middleware*.

Como o módulo *PRISEC* ainda não se encontrava desenvolvido, tanto o servidor como a aplicação não tinham implementados quaisquer aspetos de segurança. Um dos principais objetivos foi, então, corrigir as falhas de segurança no servidor. Feito isto, o que se seguia seria a implementação do módulo de segurança na aplicação *Android*.

Por último, delineou-se fazer uma análise de como se poderiam aplicar no UbiPri os diferentes algoritmos de criptografia com base nos testes feitos no início do projeto, de forma a maximizar a eficiência dos procedimentos criptográficos de todo o ecossistema do *middleware*.

1.4 Organização do Documento

O relatório encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – apresenta o projeto, a motivação para a sua escolha, o enquadramento para o mesmo, os seus objetivos e a respetiva organização do documento.
2. O segundo capítulo – **Estado da Arte** – analisa trabalhos relacionados com o tema deste projeto.

3. O terceiro capítulo – **Tecnologias e Ferramentas Utilizadas** – descreve os conceitos mais importantes no âmbito deste projeto, bem como as tecnologias utilizadas durante o desenvolvimento das aplicações.
4. O quarto capítulo – **Engenharia de Software** – apresenta os requisitos funcionais e não funcionais dos aspetos de implementação deste projeto.
5. O quinto capítulo – **Implementação e Testes dos Algoritmos Criptográficos** – expõe a implementação do programa utilizado para avaliar o desempenho dos algoritmos testados, bem como os resultados dos testes.
6. O sexto capítulo – **Implementação do Módulo de Segurança no Servidor UbiPri** – apresenta o módulo de segurança que foi adicionado ao servidor do UbiPri.
7. O sétimo capítulo – **Alterações à Aplicação *Android* do UbiPri** – mostra o que foi adicionado à aplicação para *Android* do UbiPri.
8. O oitavo capítulo – **Conclusões e Trabalho Futuro** – descreve as principais conclusões que se retiraram deste projeto e o que se pretende desenvolver no futuro.

Capítulo 2

Estado da Arte

2.1 Introdução

Neste capítulo serão apresentados alguns dos estudos mais relevantes relacionados com a área de pesquisa deste projeto. A secção 2.2 apresenta os trabalhos relacionados, onde é exposta a informação de como os estudos foram realizados e os seus resultados. Em 2.3 são mostradas as conclusões deste capítulo.

2.2 Trabalhos Relacionados

Recentemente, vários estudos têm sido feitos para medir o desempenho de diferentes algoritmos criptográficos. É importante ter em consideração não só a segurança do próprio algoritmo, mas também a sua eficiência, principalmente se vamos utilizar em dispositivos com um poder computacional limitado.

Em [6] verificou-se um melhor desempenho do *Rivest Cipher 6* (RC6) em relação ao AES. O ambiente de teste foi uma placa *BeagleBone Black*, um microcontrolador muito utilizado na IoT. Para esta placa, o algoritmo RC6 obteve tempos de execução até 10 vezes mais rápidos do que o AES.

Num estudo feito num dispositivo móvel [18], o consumo de bateria de um *smartphone* foi medido depois de este executar os algoritmos RC6, *Twofish*, *Serpent* e *Mars*, finalistas do concurso para escolher o algoritmo que se tornaria o AES. O RC6 e o *Twofish* obtiveram o melhor desempenho energético, enquanto que o *Serpent* foi o que consumiu mais bateria.

No estudo realizado em [16], foram comparados os algoritmos AES, RC6, *Rivest-Shamir-Adleman* (RSA) e *Triple DES* (3DES) a nível do seu tempo de execução e memória utilizada. Verificou-se que o algoritmo RC6 obteve o melhor tempo de execução e o menor uso de memória para guardar constantes e código. No entanto, o AES teve um menor uso de memória para guardar dados do que

o RC6. A figura 2.1 mostra os resultados do tempo de execução obtidos neste estudo.

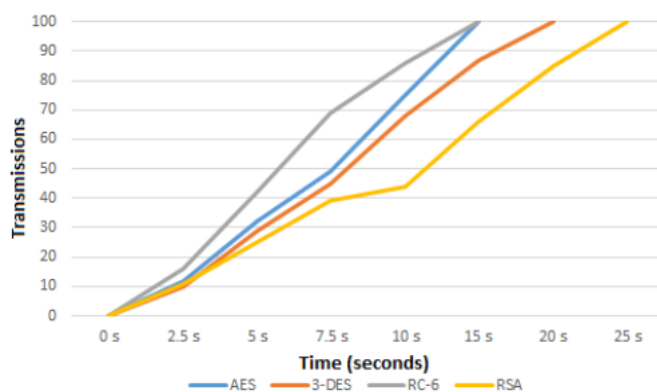


Figura 2.1: Número de transmissões pelo tempo de execução dos algoritmos criptográficos testados, Ochôa *et al.*[16].

O desempenho de diferentes algoritmos de criptografia de chave simétrica foi avaliado em [20], bem como de funções de *hash* da família *Secure Hash Algorithm* (SHA). Estes algoritmos foram executados em placas *Raspberry Pi 3* e *Raspberry Pi Zero*. Os algoritmos de chave simétrica testados incluíram o RC6, AES e *Twofish*, todos em modo *Counter* (CTR) e com tamanho de chave de 128 *bits*. Os resultados mostraram que o RC6 obteve não só a maior taxa de transferência de dados mas também o menor consumo energético para ambas as placas em relação ao AES e *Twofish*.

Em [11] foi avaliada a taxa de transferência de dados e tempo para configurar a chave e *Initialization Vector* (IV) de algoritmos como o *CAST-256*, *Camellia*, *Twofish* e *Serpent*, todos em modo CTR com tamanho de chave de 128 *bits*. Enquanto que o *Twofish* obteve a maior taxa de transferência dos algoritmos aqui mencionados (17 *MiB/s*), também foi o que demorou mais tempo para configurar a sua chave simétrica e IV. O *Camellia* obteve os menores tempos de configuração e a segunda taxa de transferência de dados mais alta (14 *MiB/s*).

Já na criptografia de chave pública, os algoritmos de *Elliptic-Curve Cryptography* (ECC) têm ganho alguma relevância. Estes permitem o mesmo nível de segurança do RSA com tamanhos de chave mais pequenos, aumentando a sua eficiência. No entanto, são mais difíceis de implementar corretamente, o que pode abrir portas para falhas de segurança. Em [9], o algoritmo *Elliptic-Curve Diffie-Hellman* (ECDH) teve tempos de execução mais rápidos e gastos de energia muito mais reduzidos do que o RSA durante o processo de geração de chaves. O tempo necessário para assinar e verificar assinaturas digitais com o *Elliptic-Curve Digi-*

tal Signature Algorithm (ECDSA) também foi notavelmente melhor do que com o *Digital Signature Algorithm* (DSA) e RSA. A figura 2.2 mostra o tempo de execução em segundos e o consumo energético em *milijoule* da geração de pares de chaves dos algoritmos RSA e ECDH.

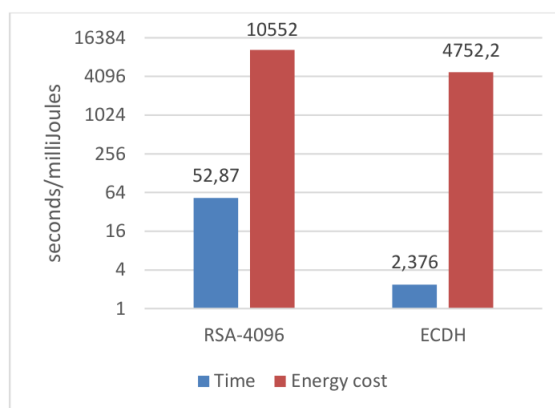


Figura 2.2: Tempo (segundos) e consumo energético (*milijoule*) para a geração de chaves dos algoritmos RSA e ECDH, El-Hajj *et al.* [9].

O algoritmo *ChaCha20* [2] tem vindo a ganhar alguma popularidade, sendo uma melhoria do *Salsa20*, uma cifra de fluxo implementada em 2005. O *ChaCha20* já foi adotado pela *Google* e é uma das cifras suportadas pela versão mais recente do protocolo *Transport Layer Security* (TLS). É normalmente usado em conjunto com o *Message Authentication Code* (MAC) *Poly1305* para autenticar as mensagens encriptadas. De Santis *et al.* [8] analisaram a eficiência do esquema de encriptação autenticada *ChaCha20-Poly1305* em relação ao AES num microprocessador *ARM Cortex-M4*, que é bastante utilizado na IoT. Verificou-se que o *ChaCha20-Poly1305* obteve tempos de execução consideravelmente mais rápidos do que o AES nos modos de cifra testados: *Encrypt then Authenticate then Translate* (EAX), *Galois/Counter Mode* (GCM) e *Counter with CBC-MAC* (CCM).

2.3 Conclusões

Com base nos estudos realizados podemos verificar que, para certos dispositivos, existem alternativas potencialmente melhores do que o AES para proteção dos dados. Algoritmos como o RC6 e o *Twofish* são considerados seguros tal como o AES. Como podem obter um melhor desempenho, parece ser boa ideia utilizá-los quando queremos poupar recursos computacionais. De facto, a poupança de recursos pode ser um aspeto bastante crítico em microcontroladores e dispositivos

móveis que funcionam a bateria. A tabela 2.1 mostra uma comparação simples de alguns dos estudos mencionados neste capítulo com o projeto aqui desenvolvido.

Tabela 2.1: Comparação de alguns dos estudos feitos com este projeto.

Autor/Testes	AES	RC6	Twofish	RSA/ ECC	ChaCha20	Tempo de Execução	Consumo Energético
Soewito <i>et al.</i>	Não	Sim	Sim	Não	Não	Não	Sim
De Santis <i>et al.</i>	Sim	Não	Não	Não	Sim	Sim	Não
El-Hajj <i>et al.</i>	Sim	Não	Não	Sim	Não	Sim	Sim
Yilmaz e Özdemir	Sim	Sim	Sim	Não	Não	Não	Sim
Leithardt <i>et al.</i>	Sim	Sim	Não	Sim	Não	Sim	Não
Este projeto	Sim	Sim	Sim	Não	Não	Sim	Não

Desta forma, a análise destes trabalhos também nos permitiu obter uma ideia do ponto de partida para o projeto aqui desenvolvido e dos métodos criptográficos mais relevantes na atualidade.

Capítulo 3

Tecnologias e Ferramentas Utilizadas

3.1 Introdução

Este capítulo irá apresentar as tecnologias e ferramentas utilizadas durante a realização deste projeto, tais como linguagens de programação e ambientes de desenvolvimento. A secção 3.2 lista todas estas tecnologias, enquanto que a secção 3.3 mostra as ferramentas utilizadas. Na secção 3.4 encontramos a conclusão deste capítulo.

3.2 Descrição das Tecnologias Utilizadas

As tecnologias utilizadas no desenvolvimento deste projeto foram:

1. **Linguagens de programação** – Utilizaram-se as linguagens C/C++ para a implementação do programa de testes dos algoritmos criptográficos. Para as funcionalidades e módulo de segurança do UbiPri, utilizou-se a linguagem Java. Também foi utilizada a *Extensible Markup Language* (XML) para configuração do servidor *Apache Tomcat* e implementação das atividades da aplicação *Android* do UbiPri.
2. **Tecnologias de Segurança** – Foram utilizadas várias tecnologias de segurança, entre as quais:
 - **Argon2id** – É uma função de derivação de chaves que ganhou a *Password Hashing Competition* de 2015 [3]. As suas principais características são resistência a ataques de *Graphics Processing Unit* (GPU) e ataques *side-channel*. Os seus parâmetros permitem controlar tempo

de execução, memória utilizada e paralelismo. É atualmente a função *state of the art* para *hash* de *passwords*.

- **JSON Web Token (JWT)** – Um padrão baseado em *JavaScript Object Notation* (JSON) para criação de *tokens* de autenticação.
3. **Arquitetura Representational State Transfer (REST)** – Um estilo de arquitetura utilizado para a criação de *Web Services*. O servidor do UbiPri foi implementado usando esta arquitetura.

3.3 Descrição das Ferramentas Utilizadas

Como ambientes de desenvolvimento, utilizaram-se os seguintes programas:

1. *Visual Studio Code* – Editor de texto *open source* desenvolvido pela *Microsoft*.
2. *Apache Netbeans 8.2* – Um *Integrated Development Environment* (IDE) para desenvolvimento de aplicações Java. Foi recentemente doado à *Apache Foundation*, tornando-se *open source*.
3. *Android Studio 3.4* – Um IDE para desenvolvimento de aplicações para o sistema operativo *Android* baseado no *IntelliJ IDEA*.

Para testar os algoritmos de criptografia, foi utilizado o *software QEMU* para emular um processador *ARM Cortex-A15*. Escolhemos este processador porque se encontra pronto a ser emulado no *QEMU* e por ser um dos mais recentes da arquitetura *ARMv7-A*, além de ser utilizado em muitos dispositivos IoT. O servidor do UbiPri foi hospedado no *Apache Tomcat 9.0*. Os testes da aplicação *Android* desenvolvida foram feitos no *Android Emulator*. As bases de dados estão implementadas em *PostgreSQL*. Usaram-se ainda bibliotecas e implementações externas *open source* para auxiliar o desenvolvimento das aplicações deste projeto, que estão listadas a seguir:

- *Crypto++* - Uma biblioteca para C++ que disponibiliza implementações de inúmeros algoritmos criptográficos e funções de *hash* [7];
- *LibTomCrypt* - Outra biblioteca para C/C++ que contém implementações de bastantes algoritmos de criptografia e funções de *hash* [14];
- *Argon2* - Implementação oficial do vencedor da *Password Hashing Competition* disponível em [4];
- *JWT* - Implementação do JWT para Java e *Android* [10].

3.4 Conclusões

Com este capítulo foram apresentadas as tecnologias e ferramentas utilizadas na concepção deste trabalho, as quais foram valiosas para atingir os objetivos do mesmo.

Capítulo 4

Engenharia de *Software*

4.1 Introdução

Neste capítulo, serão descritos os requisitos funcionais e não funcionais para o servidor e aplicação do UbiPri nas secções 4.2 e 4.3, respetivamente. A secção 4.4 apresenta as conclusões que se retiram deste capítulo.

4.2 Especificação de Requisitos do Servidor UbiPri

Esta secção irá apresentar os diferentes requisitos funcionais e não funcionais para o módulo de segurança e funcionalidades em falta do servidor do UbiPri.

4.2.1 Requisitos Funcionais

1. **Suporte ao protocolo *Hypertext Transfer Protocol Secure* (HTTPS)** – O servidor deverá ter suporte ao protocolo HTTPS de maneira a tornar todas as mensagens trocadas com os clientes autenticadas e encriptadas. Além disso, todas as requisições *Hypertext Transfer Protocol* (HTTP) devem ser recusadas e redirecionadas para HTTPS.
2. ***Hash de passwords*** – O servidor deverá calcular o *hash* das *passwords* com recurso à função *Argon2id* e guardar este valor na base de dados. Além disso, deverá ser gerado um *salt* que também deverá ser guardado na base de dados para calcular o valor do *hash*.
3. ***Tokens JWT*** – O servidor deverá gerar um *token* JWT para um utilizador quando o *login* é feito. Este *token* terá uma validade de 6 horas, significando que durante esse tempo, o utilizador é autenticado automaticamente no seu

dispositivo sem ser necessário inserir as suas credenciais. O servidor deve ainda poder verificar se o *token* é válido ou não.

4. **Suporte a requisições *POST*** – O servidor deverá operar sobre requisições *POST* em vez de *GET* para operações como *login*, registo e validação de *tokens*. As requisições *GET* colocam os parâmetros no *Uniform Resource Identifier* (URI), o que pode representar uma falha de segurança.
5. **Registo de utilizadores e dispositivos** – O servidor deverá suportar o registo de novos utilizadores e dispositivos. Para tal, deverá ser necessário guardar os seguintes dados:
 - Nome de utilizador;
 - nome completo;
 - *email*;
 - *password*;
 - *salt*;
 - código do dispositivo;
 - nome do dispositivo;
 - tipo de dispositivo;
 - funcionalidades do dispositivo.

4.2.2 Requisitos Não Funcionais

1. **Java Development Kit (JDK)** – Será necessário o JDK versão 8 (no mínimo) para executar os métodos do servidor.
2. **Apache Tomcat** – Será necessário instalar o *Apache Tomcat* versão 9.0 para hospedar o servidor.
3. **Protocolo TLS** – Apenas as versões 1.2 e 1.3 do protocolo TLS deverão ser suportadas.

4.3 Especificação de Requisitos da Aplicação UbiPri

Nesta secção apresentam-se os diferentes requisitos funcionais e não funcionais para o módulo de segurança e funcionalidades em falta da aplicação do UbiPri.

4.3.1 Requisitos Funcionais

1. **Suporte ao protocolo HTTPS** – A aplicação deverá estabelecer conexões HTTPS ao servidor de maneira a garantir a autenticidade e criptografia das mensagens trocadas.
2. **Armazenamento seguro de *tokens*** – A aplicação deverá armazenar de um modo seguro os *tokens* JWT recebidos do servidor através da encriptação dos mesmos. Para tal, utilizar-se-á a *Android KeyStore Application Programming Interface* (API) para guardar as chaves simétricas de encriptação de modo seguro. Os *tokens* serão guardados encriptados nas *Shared Preferences* em modo privado.
3. **Serviço para deteção de localização** – A aplicação deverá detetar a localização do utilizador com base num serviço a correr no *background* mesmo quando a aplicação não está focada.
4. **Suporte a registo de novos utilizadores e dispositivos** – A aplicação deverá permitir o registo de novos utilizadores e dispositivos. Para tal, deverá ser pedido ao utilizador os seguintes dados:
 - Nome de utilizador;
 - nome completo;
 - *email*;
 - *password*;

A aplicação deverá ainda gerar um código e nome único para o dispositivo de forma automática, além de obter as funcionalidades do dispositivo, que serão enviados para o servidor no momento do registo.

4.3.2 Requisitos Não Funcionais

1. **Sistema Operativo** - A aplicação do UbiPri deverá correr em dispositivos *Android* que possuam, no mínimo, a versão 6.0 *Marshmallow*.

4.4 Conclusões

Neste capítulo, apresentaram-se os requisitos funcionais e não funcionais do servidor e aplicação do *middleware* UbiPri. Desta forma, podemos dar início à implementação dos módulos de segurança dos mesmos, bem como das suas funcionalidades em falta.

Capítulo 5

Implementação e Testes dos Algoritmos Criptográficos

5.1 Introdução

Este capítulo irá detalhar a implementação do programa desenvolvido para testar diferentes algoritmos de criptografia a nível do seu tempo de execução, apresentando também os resultados obtidos.

Na secção 5.2 é apresentada a maneira como o programa foi implementado, mostrando trechos de código para exemplificar. Na secção 5.3, são mostrados e discutidos os resultados obtidos nos ambientes de teste. Por fim, em 5.4 são discutidas as conclusões que se tiraram da realização destes testes.

5.2 Implementação dos Testes de Criptografia

O primeiro passo deste projeto foi avaliar o desempenho de diferentes algoritmos de criptografia. Para tal, desenvolveu-se uma aplicação em C++ com recurso à biblioteca *Crypto++*.

No *middleware* UbiPri, são atribuídos aos utilizadores diferentes níveis de acesso para controlar as suas definições de privacidade e de privilégios num dado ambiente. A avaliação do nível de acesso tem em conta vários fatores, tais como se o ambiente é público ou privado, se é dia útil ou não, a hora do dia, entre outros. Os níveis de acesso que foram definidos em [12] são *Guest*, *Basic*, *Advanced* e *Admin*. Como exemplo prático, um professor numa sala de aula (ambiente privado) durante o decorrer da sua aula teria privilégios de Administrador, enquanto que um aluno teria o nível de acesso *Basic*. O nível de acesso também irá definir os algoritmos criptográficos aplicados para proteger os dados do utilizador, bem

como o tamanho máximo de cada pacote a ser encriptado, tendo sido definidos os seguintes:

- *Guest*: *Base64*, 1 a 10 MB de dados;
- *Basic*: *Base64* + AES-128-GCM, 1 a 20 MB de dados;
- *Advanced*: *Base64* + AES-128-GCM + AES-192-GCM, 1 a 30 MB de dados;
- *Admin*: *Base64* + AES-128-GCM + AES-192-GCM + AES-256-GCM, 1 a 50 MB de dados.

Foram então definidos mais dois níveis semelhantes ao *Admin*, mas aplicando os algoritmos RC6 e *Twofish*, dois dos finalistas do concurso para escolher o AES, de forma a comparar o desempenho destes em relação ao algoritmo *Rijndael* que acabou por ganhar o concurso e que é hoje o *standard* na criptografia de chave simétrica.

Estes algoritmos também foram os que obtiveram melhor desempenho no estudo realizado em [18], sendo outra razão importante para a sua escolha.

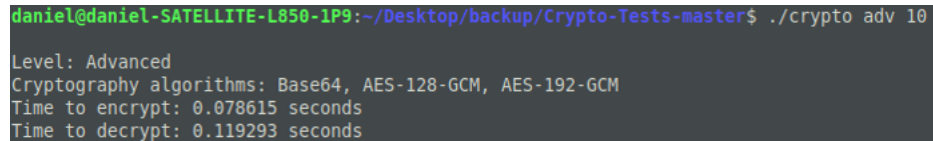
- RC6: *Base64* + RC6-128-GCM + RC6-192-GCM + RC6-256-GCM, 1 a 50 MB de dados.
- *Twofish*: *Base64* + *Twofish*-128-GCM + *Twofish*-192-GCM + *Twofish*-256-GCM, 1 a 50 MB de dados.

Desta forma, o programa recebe como argumentos o nível de acesso do utilizador e o tamanho do pacote em *megabytes*. O *main()* trata apenas de chamar as funções que tratam de cada nível de acesso, passando o respetivo tamanho de pacote. A rotina de uma execução deste programa pode ser resumida da seguinte forma:

- É gerado um pacote aleatório com o tamanho especificado;
- são geradas as chaves simétricas e vetores de inicialização necessários;
- o pacote é encriptado com os algoritmos do respetivo nível de acesso especificado;
- o tempo de encriptação é medido;
- o pacote é desencriptado com os algoritmos do respetivo nível de acesso especificado;
- o tempo de desencriptação é medido;

- os resultados são apresentados no ecrã e o programa termina a sua execução.

A figura 5.1 mostra a execução deste programa com o nível de acesso *Advanced* e com um pacote de tamanho 10 MB, de maneira a exemplificar como a aplicação funciona.



```
daniel@daniel-SATELLITE-L850-1P9:~/Desktop/backup/Crypto-Tests-master$ ./crypto adv 10
Level: Advanced
Cryptography algorithms: Base64, AES-128-GCM, AES-192-GCM
Time to encrypt: 0.078615 seconds
Time to decrypt: 0.119293 seconds
```

Figura 5.1: Exemplo de uma execução do programa desenvolvido com nível de acesso *Advanced* e um pacote com tamanho de 10 *megabytes*.

Serão agora descritos com mais detalhes os pontos listados acima nas subsecções seguintes.

5.2.1 Geração do Pacote

O pacote é gerado aleatoriamente com recurso à função `rand()` da biblioteca nativa do C `rand.h`, tendo o tipo `unsigned char*` para representar um *array* de *bytes*. Para inserir mais entropia a esta função aleatória fraca, usa-se o tempo chamando a função `time()` no método `srand()`. O trecho de código seguinte mostra como um pacote é gerado no programa desenvolvido.

```
void Funcs::Basic(int pack_size)
{
    unsigned long int pt_len = 1024*1024*pack_size; // megabytes
    para bytes
    unsigned char *plain_text = (unsigned char*) malloc ((pt_len
        )*sizeof(unsigned char));

    int i;
    srand (time(NULL));

    for (i = 0; i < pt_len; i++)
    {
        plain_text[i] = rand();
    }
    ...
}
```

Excerto de Código 5.1: Trecho de código da geração de um pacote aleatório.

5.2.2 Geração das Chaves Simétricas e Vetores de Inicialização

As chaves simétricas e os vetores de inicialização são gerados com funções pseudo-aleatórias criptograficamente seguras implementadas na biblioteca *Crypto++*. Para tal, utiliza-se a classe *AutoSeededRandomPool*, um gerador pseudo-aleatório desta biblioteca. A chave terá o tipo *SecByteBlock* do *Crypto++* para ser armazenada em memória de modo seguro. O IV tem o tipo *unsigned char** visto que este não precisa de ser secreto. Todos os IV têm um tamanho de 12 bytes, que é o recomendado pela especificação do modo GCM [1]. O trecho de código seguinte mostra como chaves simétricas e IV são gerados.

```
AutoSeededRandomPool prng;  
SecByteBlock key(16); // chave de 128 bits  
prng.GenerateBlock(key, key.size());  
  
unsigned char iv[12];  
prng.GenerateBlock(iv, sizeof(iv));
```

Excerto de Código 5.2: Trecho de código da geração de chaves simétricas e IV.

5.2.3 Encriptação e Desencriptação do Pacote

Com o pacote, chaves e IV gerados, procede-se à encriptação do pacote. O tempo de encriptação começa a ser medido a partir daqui com a função *clock()* da biblioteca nativa *time.h*.

Para qualquer um dos níveis, começa-se por codificar a mensagem para *Base64*. É utilizado o método *base64_encode()* da biblioteca *Libtomcrypt* para este fim. Esta função recebe como argumentos a mensagem a ser codificada, o tamanho da mensagem, o *array* do tipo *unsigned char** de destino e o tamanho da mensagem codificada. Note-se que se o nível de acesso for o *Guest*, o processo de encriptação termina aqui e chama-se novamente o método *clock()* para medir o tempo de encriptação, procedendo-se à descodificação *Base64* e medindo igualmente o tempo de descodificação.

Como as funções de criptografia da biblioteca *Crypto++* operam sobre o tipo *string*, a mensagem codificada em *Base64* recebe um *cast* de *unsigned char** para *string*. Inicia-se assim o processo de encriptação.

Começamos por instanciar a classe responsável por encriptar o pacote, chamando de seguida o método *SetKeyWithIV()* no objeto instanciado. Esta função define a chave simétrica e o IV a serem usados para encriptação.

Por último, usamos o construtor da classe *StringSource* para redirecionar a mensagem a ser encriptada para o construtor da classe *AuthenticatedEncryptionFilter* que vai tratar da encriptação em si. Todo este processo está dentro de uma

cláusula *try...catch* para permitir tratamento de erros. No final, o tempo de encriptação é medido chamando novamente a função `clock()` e fazendo a diferença dos tempos. Para a desencriptação, o processo é semelhante e feito em reverso. Por outras palavras, se foi feita uma encriptação *Base64* -> *AES-128*, então agora começa-se por desencriptar o segredo aplicando a chave simétrica gerada anteriormente e só depois é que se faz a decodificação *Base64*. A única diferença é que aqui usaremos a classe *AuthenticatedDecryptionFilter* e o método `base64_dec()`. O trecho de código seguinte mostra como é feita a encriptação para o nível de acesso *Basic*.

```
/* Inicio da encriptacao */
clock_t start = clock();

/* Codificar para Base64 */
base64_encode(plain_text , pt_len , base64_enc , &base64_len);

string pdata(reinterpret_cast< char const* >(base64_enc));
string cipher , rpdata;

try
{
    /* Encriptar para AES-128-GCM */
    GCM< AES >::Encryption e;
    e.SetKeyWithIV(key , key.size() , iv , sizeof(iv));

    StringSource ssl(pdata , true ,
        new AuthenticatedEncryptionFilter(e ,
            new StringSink(cipher) , false , TAG_SIZE
        )
    );
}
catch(CryptoPP::Exception& e)
{
    cerr << e.what() << endl;
    exit(1);
}

/* Fim da encriptacao */
clock_t end = clock();
double elapsed_enc = (double)(end - start) / CLOCKS_PER_SEC;
```

Excerto de Código 5.3: Trecho de código da encriptação do pacote.

5.3 Resultados dos Testes

Os testes foram realizados numa máquina com as seguintes características:

- Sistema Operativo: *Linux Mint 19.1 64-bit*;
- RAM: 8 GB;
- Processador: Intel® Core™ i5-3230m 3.2 GHz com 2 núcleos (com tecnologia *Hyperthreading* simulando 4 núcleos).

O programa corre no terminal e são passados como argumentos o nível de acesso e o tamanho do pacote em *megabytes*. Foram testados os tempos de execução de todos os níveis para tamanhos de pacote de 1, 5, 10, 20, 30, 40 e 50 *megabytes*. Note-se que em níveis como o *Guest*, pacotes de tamanho maior que 10 MB não foram testados, já que este é o tamanho máximo de pacote que este nível de acesso permite. As tabelas 5.1 e 5.2 mostram o tempo em segundos de encriptação e desencriptação, respetivamente, para o processador da máquina especificada acima.

Tabela 5.1: Tempo (segundos) de encriptação para o processador Intel® Core™.

Pacote/Nível	Guest	Basic	Advanced	Admin	RC6	Twofish
1 MB	0.001387	0.005009	0.008428	0.011583	0.042762	0.035748
5 MB	0.006695	0.024439	0.038700	0.053514	0.201375	0.172026
10 MB	0.013337	0.048140	0.078615	0.107077	0.403655	0.344725
20 MB		0.096754	0.153608	0.213329	0.819665	0.688903
30 MB			0.244936	0.344732	1.256466	1.058652
40 MB				0.442673	1.641049	1.371701
50 MB				0.615783	2.093196	1.817660

Tabela 5.2: Tempo (segundos) de desencriptação para o processador Intel® Core™.

Pacote/Nível	Guest	Basic	Advanced	Admin	RC6	Twofish
1 MB	0.002370	0.009506	0.012335	0.015596	0.044571	0.038704
5 MB	0.011909	0.045555	0.058880	0.073672	0.223152	0.193471
10 MB	0.023841	0.088987	0.119293	0.148449	0.452738	0.386718
20 MB		0.177205	0.235981	0.295134	0.903632	0.770577
30 MB			0.366507	0.481640	1.368353	1.183474
40 MB				0.616864	1.786420	1.536754
50 MB				0.832803	2.303603	1.996880

Como se pode observar pelas tabelas, os resultados obtidos nos estudos analisados em 2.2 não se verificaram neste ambiente de testes. O algoritmo AES obteve tempos de encriptação e desencriptação melhores do que o RC6 e *Twofish* para todos os parâmetros de teste.

Analisando as características do processador utilizado para avaliar o desempenho destes algoritmos, notámos que este tem aceleração no *hardware* para as operações do algoritmo AES através das instruções *Advanced Encryption Standard New Instructions* (AES-NI). A biblioteca *Crypto++* tira proveito destas instruções no processo de compilação se o processador em que o programa é compilado tiver suporte às mesmas. Encontrámos, então, a principal razão do AES ter tido uma eficiência consideravelmente melhor em relação aos outros algoritmos.

No entanto, muitos dispositivos não têm suporte às instruções AES-NI nos seus processadores. Além disso, implementações que são executadas em máquinas virtuais (como é o caso das linguagens Java ou C#) também podem não tirar partido deste tipo de otimizações de baixo nível.

Bastantes microprocessadores da arquitetura *ARM* não têm as instruções AES-NI implementadas. Como são muito utilizados em dispositivos móveis e microcontroladores da IoT, achou-se relevante correr os mesmos testes para um processador deste tipo.

Na ausência de um microprocessador físico, escolhemos o *software QEMU* para emular um processador *ARM Cortex-A15*. Procedemos então ao processo de *cross-compilation* do programa desenvolvido com recurso ao compilador *arm-linux-gnueabi-hf-g++*. A emulação deste microprocessador e execução do *QEMU* foram feitas na mesma máquina especificada no início da secção 5.3. As tabelas 5.3 e 5.4 mostram o tempo em segundos de encriptação e desencriptação, respetivamente, para o processador *Cortex-A15* emulado.

Tabela 5.3: Tempo (segundos) de encriptação para o processador *Cortex-A15* emulado.

Pacote/Nível	Guest	Basic	Advanced	Admin	RC6	Twofish
1 MB	0.003718	0.095294	0.179608	0.267513	0.200433	0.211463
5 MB	0.017481	0.434503	0.851855	1.294984	0.953222	1.008245
10 MB	0.036054	0.863211	1.699903	2.591663	1.936132	2.004360
20 MB		1.716348	3.383221	5.171729	3.856180	3.994297
30 MB			5.081432	7.781451	5.731962	5.993484
40 MB				10.358439	7.637404	7.975846
50 MB				12.932847	9.786043	10.057823

Tabela 5.4: Tempo (segundos) de descriptação para o processador *Cortex-A15* emulado.

Pacote/Nível	Guest	Basic	Advanced	Admin	RC6	Twofish
1 MB	0.017774	0.111367	0.199604	0.289042	0.215559	0.228209
5 MB	0.087356	0.547343	0.966063	1.410512	1.065690	1.117404
10 MB	0.178384	1.092509	1.941732	2.825305	2.139458	2.243983
20 MB		2.179369	3.851166	5.645727	4.315076	4.464409
30 MB			5.787087	8.458038	6.539588	6.684949
40 MB				11.283066	8.511327	8.901239
50 MB				14.121240	10.829722	11.207440

Como podemos ver nas tabelas acima, o RC6 obteve os melhores tempos de encriptação e descriptação, seguido do *Twofish*. O AES aqui já teve uma eficiência consideravelmente pior, tendo sido até 32% mais lento do que o RC6. Outro resultado interessante foi o facto do RC6 ter obtido um melhor desempenho do que o *Twofish*, o que não se verificou no processador físico da Intel®.

5.4 Conclusões

Neste capítulo, mostrou-se que podem existir boas alternativas ao AES para proteção de dados de forma a poupar recursos computacionais. No entanto, isto pode não ser viável em todos os dispositivos.

Para máquinas com processadores da Intel® comuns, não se justifica utilizar outro algoritmo de chave simétrica para além do AES. Já para dispositivos móveis ou microprocessadores da arquitetura *ARM*, o RC6 e *Twofish* podem fornecer melhorias de eficiência consideráveis, pelo que o seu uso deve ser considerado sempre que possível.

Capítulo 6

Implementação do Módulo de Segurança no Servidor UbiPri

6.1 Introdução

O principal objetivo deste capítulo é descrever o módulo de segurança adicionado ao servidor do UbiPri. Inicialmente, é apresentada a organização dos diferentes ficheiros de código fonte do servidor na secção 6.2. De seguida, procede-se à exposição de como se habilitou o protocolo HTTPS em 6.3. Na secção 6.4 descreve-se as alterações feita aos métodos já implementados no servidor de maneira a cumprir as regras de segurança estabelecidas. Por último, apresentam-se as conclusões que se retiram deste capítulo em 6.5.

6.2 Organização das Fontes do Servidor

Esta secção serve para descrever como os diferentes *packages* e classes do servidor do UbiPri se encontram organizados.

6.2.1 *Package server*

Este *package* contém apenas a classe *Main* do servidor e o método *main()* correspondente.

6.2.2 *Package server.dao*

Contém todas as classes que comunicam com as diferentes tabelas da base de dados do servidor. Os métodos destas classes são responsáveis por obter a informação que desejamos das bases de dados, bem como a realização de inserções e

remoção de registos. São apenas usados *prepared statements* de maneira a impedir ataques de *SQL injection*.

6.2.3 *Package server.model*

Tem a definição das classes que representam os modelos de cada tabela da base de dados. Estes modelos servem para auxiliar durante as operações de inserção, seleção e remoção da base de dados, já que permitem criar objetos temporários semelhantes à informação representada na base de dados.

6.2.4 *Package server.model.classify*

Contém os classificadores utilizados para atribuir um nível de acesso a um dado utilizador tendo em conta o dia da semana, tipo de ambiente, hora do dia, entre outros fatores.

6.2.5 *Package server.modules.communication*

Neste *package* encontramos as classes que estabelecem a comunicação entre diferentes *packages* do servidor. A mais importante aqui é a classe *Communication* que serve de intermediária entre os métodos REST da classe *WebServiceRestCommunication* 6.2.6 e os métodos implementados na classe *PrivacyControlUbiquitous* 6.2.8.

6.2.6 *Package server.modules.communication.rest*

Tem a classe *WebServiceRestCommunication*, cujos métodos são responsáveis por receber as requisições que os clientes fazem ao servidor, bem como o processamento das mesmas.

6.2.7 *Package server.modules.crypto*

Este *package* contém as classes que tratam dos serviços de criptografia e segurança, sendo responsáveis por garantir a proteção dos dados dos utilizadores e da base de dados do servidor.

6.2.8 *Package server.modules.privacy*

Contém a classe *PrivacyControlUbiquitous*, sendo responsável pela implementação das principais funcionalidades do servidor, processando *logins*, registos, deteção dos ambientes em que se encontram os utilizadores, entre outros.

6.2.9 *Package server.util*

As classes deste *package* contêm métodos que auxiliam na conexão à base de dados. Também tem as configurações para aceder à mesma, além de gerir as desconexões para evitar *leaks* de conexão.

6.3 Habilitação do protocolo HTTPS

O primeiro passo no desenvolvimento do módulo de segurança do UbiPri foi adicionar suporte ao protocolo HTTPS, visto que ainda não se encontrava habilitado.

Começou-se por gerar uma chave privada e um certificado *self-signed* para o servidor. Utilizou-se a *Java Keytool* para este fim, emitindo o comando *keytool -genkey -alias ubipri -keyalg RSA* no terminal. Este comando cria uma *Keystore* em formato *Java KeyStore* (JKS).

O próximo passo foi extrair o certificado e a chave privada dessa *keystore* para 2 ficheiros em formato *.crt* e *.pem*, respetivamente. Para o ficheiro do certificado, emitiu-se o comando *keytool -export -alias ubipri -file ubipri.crt -keystore keystore.jks*. De maneira a criar o ficheiro da chave privada, primeiro converteu-se a JKS para uma *PKCS12* *keystore* com *keytool -importkeystore -srckeystore keystore.jks -destkeystore keystore.p12 -srcstoretype jks -deststoretype pkcs12*. Emitiu-se, depois, o comando *openssl pkcs12 -in keystore.p12 -out ubipri.pem* para extrair a chave privada.

Com o certificado e chave privada gerados e extraídos para 2 ficheiros distintos, configurou-se então o *Apache Tomcat* para aceitar o protocolo HTTPS. Para tal, alterou-se o ficheiro *server.xml* que contém as definições do servidor, adicionando-se as seguintes linhas:

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />

<Connector port="8443" protocol="org.apache.coyote.http11.
           Http11AprProtocol" SSLEnabled="true"
           maxThreads="150" scheme="https" secure="true"
           clientAuth="false" sslProtocol="TLSv1.2+TLSv1.3"
           SSLCertificateFile="conf/ubipri.crt"
           SSLCertificateKeyFile="conf/ubipri.pem" />
```

Excerto de Código 6.1: Linhas de código adicionadas à configuração do servidor para suportar HTTPS.

Note-se que, com esta configuração, conexões HTTP são redirecionadas para

HTTPS. Além disso, apenas as versões 1.2 e 1.3 do protocolo TLS são suportadas para garantir as atualizações de segurança mais recentes em todas as comunicações.

Com o HTTPS habilitado no servidor, garantimos a encriptação de todas as mensagens trocadas entre cliente e servidor. Além disso, com o certificado, o cliente tem a garantia que está, de facto, a comunicar com o servidor do UbiPri.

6.4 Atualizações Feitas aos Métodos do Servidor

Embora a maior parte das funcionalidades do servidor já estivessem implementadas, estas continham bastantes falhas de segurança ou uma completa falta dela. Um exemplo disso era o método do *login* a fazer feito por uma requisição *GET* em vez de *POST*. As subsecções seguintes mostram as alterações feitas aos métodos que tratam das requisições dos clientes ao servidor.

6.4.1 Proteção das *Passwords* e Geração de *Tokens*

As *passwords* estavam a ser guardadas em texto limpo na base de dados do servidor. Esta foi uma falha crítica corrigida de imediato. Começou-se por adicionar uma coluna *salt* à tabela dos utilizadores. De seguida, escolheu-se a função de derivação *Argon2id* para calcular o valor do *hash* das *passwords*. Esta função é atualmente a *state of the art* para este fim, visto que ganhou a *Password Hashing Competition* de 2015.

Outro problema que foi verificado é que todas as requisições ao servidor recebiam a *password* do utilizador para autenticar essa requisição. Além de ser uma má prática, isto obrigaria o utilizador a inserir a *password* sempre que fazia uma requisição ao servidor, além de obrigar o servidor a calcular o *hash* da mesma em todas as requisições, gastando bastantes recursos. Assim, foram introduzidos JWTs que são atribuídos a cada utilizador quando este faz o *login* e que terão uma validade por um tempo limitado. A verificação de JWTs também é menos custosa do que calcular valores de *hash*.

Foi, então, criado o *package server.modules.crypto* para tratar destes aspetos da segurança do UbiPri. Este *package* contém as classes *CryptoConfig* e *CryptoServices*.

No *CryptoConfig* encontram-se 2 métodos para ler a chave pública e privada no servidor. Estas chaves servirão para assinar os JWTs. A implementação das mesmas é bastante direta. Lê-se o conteúdo dos ficheiros das chaves para um *array* do tipo *byte*, cria-se um objeto *PKCS8EncodedKeySpec* no caso de ler a chave privada ou um *X509EncodedKeySpec* no caso de ler a chave pública, colocando

nesse objeto o conteúdo do *array* de *bytes*. De seguida, cria-se uma *RSA Key-Factory* e faz-se o *cast* dos *bytes* lidos para um objeto do tipo *RSAPrivateKey* ou *RSAPublicKey*. O trecho de código seguinte mostra a função que lê uma chave pública.

```
public static RSAPublicKey getPubKey() {
    RSAPublicKey publicKey = null;
    try
    {
        byte[] keyBytes = Files.readAllBytes(Paths.get("../conf/
            pubkey.der"));

        X509EncodedKeySpec spec = new X509EncodedKeySpec(
            keyBytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        publicKey = (RSAPublicKey) kf.generatePublic(spec);
    }
    catch (IOException | NoSuchAlgorithmException
        | InvalidKeySpecException e) {
        System.out.println("Error loading public key: " + e);
    }
    return publicKey;
}
```

Excerto de Código 6.2: Trecho de código da leitura de uma chave pública.

Na classe *CryptoServices* encontramos métodos para gerar um *salt*, gerar e verificar um dado JWT e uma função para calcular o *hash* de uma *password*.

A implementação da geração de um *salt* é bastante simples. Instância-se a classe *SecureRandom*, um gerador do Java pseudo-aleatório criptograficamente seguro e cria-se um *array* de *bytes* com tamanho 16 (o recomendado para *hash* de *passwords* [3]). De seguida, chamamos o método *nextBytes()* da classe *SecureRandom* e retornamos o *salt* gerado. O trecho de código seguinte mostra a implementação desta função.

```
public static byte[] generateSalt() {
    SecureRandom rand = new SecureRandom();
    byte[] salt = new byte[16];
    rand.nextBytes(salt);
    return salt;
}
```

Excerto de Código 6.3: Trecho de código do método que gera um *salt*.

Para gerar um *token*, recebemos como argumento o *username* do utilizador para quem é gerado o JWT. Com as classes *Date* e *Calendar* definimos a validade do *token* em 6 horas. Carregamos a chave privada para assinar o *token* com o método `getPrivKey()` do *CryptoConfig* e construímos o JWT. O trecho de código seguinte mostra a implementação da geração de um *token*.

```
public static String generateToken(String userName) {
    // Validade do token
    Date d = new Date();
    Calendar cal = Calendar.getInstance();
    cal.setTime(d);
    cal.add(Calendar.HOUR, 6); // validade de 6 horas
    d = cal.getTime();

    RSAPrivateKey privKey = CryptoConfig.getPrivKey();

    String jws = Jwts.builder()
        .setIssuer("UbiPri")
        .setSubject(userName)
        .setExpiration(d)
        .signWith(privKey)
        .compact();

    return jws;
}
```

Excerto de Código 6.4: Trecho de código do método que gera um *token*.

A validação de um *token* é feita a partir da classe *Jws<Claims>*. Chamando o método `parseClaimsJws()` dentro de uma cláusula *try...catch*, ele atira uma exceção se o *token* for inválido. Retorna *true* se o JWT for válido e *false* caso contrário. O trecho de código seguinte mostra o funcionamento desta função.

```
public static boolean validateToken(String userName, String
token) {
    boolean result;
    try
    {
        Jws<Claims> jws2;

        RSAPublicKey pubKey = CryptoConfig.getPubKey();

        jws2 = Jwts.parser()
            .setSigningKey(pubKey)
            .requireIssuer("UbiPri")
            .requireSubject(userName)
```



```
        .setAllowedClockSkewSeconds(3*60)
        .parseClaimsJws(token);

        result = true;
    }

    catch(JwtException e) {
        // se token nao for valido, entra neste catch
        result = false;
    }

    return result;
}
```

Excerto de Código 6.5: Trecho de código do método que valida um *token*.

Para o cálculo do *hash* de uma *password*, foi utilizado um programa em C adaptado de [4]. O servidor, executado numa máquina virtual Java, chama um novo processo para executar esse programa com a classe *ProcessBuilder* e aguarda o seu *output* (o valor do *hash* calculado). A esse processo, passamos como argumentos a *password* em texto limpo, o *salt* e o tamanho do mesmo. O trecho de código seguinte mostra a implementação da função que cria um novo processo para correr o programa implementado em C.

```
public static String hashPassword(String plainPassword, String
    salt) {
    String line, result = "";
    String[] arg = { "./test", plainPassword, salt, "24" };
    ProcessBuilder pb = new ProcessBuilder(arg);
    pb.directory(new File("/home/daniel/Desktop"));

    try {
        Process p = pb.start();
        p.waitFor();
        BufferedReader bri = new BufferedReader(new
            InputStreamReader(p.getInputStream()));

        while ((line = bri.readLine()) != null)
        {
            result+=line;
        }
    }
    catch (Exception e){}
    return result;
}
```

Excerto de Código 6.6: Trecho de código do método que cria um novo processo.

6.4.2 Alterações nos Métodos da Classe *PrivacyControlUbiquitous*

Esta classe contém a implementação dos métodos chamados pelas funções REST detalhadas em 6.4.3. Como tal, é onde se encontram as funcionalidades principais do servidor. Abaixo serão detalhadas as alterações e adições feitas aos métodos desta classe.

Método `userExists()`

Simplesmente retorna uma *String* em formato JSON a indicar se o utilizador passado como parâmetro existe ou não. Ele chama a função `userExists()` da classe *UserDAO* que faz *queries* à base de dados na tabela dos utilizadores.

```
public String userExists(String userName) {
    if (this.userDAO.userExists(userName)) {
        return "{\"status\":\"YES\"}";
    }

    return "{\"status\":\"NO\"}";
}
```

Excerto de Código 6.7: Trecho de código do método que verifica se um utilizador existe.

Método `getUserSalt()`

Apenas chama o método `getUserSalt()` da classe *UserDAO*. Este, por sua vez, faz uma *query* à base de dados dos utilizadores para obter o *salt* associado ao usuário passado como parâmetro. O trecho de código seguinte mostra a implementação da função na classe *UserDAO*.

```
public String getUserSalt(String userName) {
    String sql = "Select use_name, use_salt From users Where use_name = ?";

    String salt = "failed";

    try
    {
        pstmt = getConnection().prepareStatement(sql);
        pstmt.setString(1, userName);
        rs = pstmt.executeQuery();
    }
```

```
        if (rs.next())
        {
            salt = rs.getString("use_salt");
        }

        rs.close();
        pstmt.close();
    }
    catch (SQLException e) {
        System.out.println("Class: " + this.toString() + ".
            Exception: " + e);
    }
    this.db.disconnect();
    return salt;
}
```

Excerto de Código 6.8: Trecho de código do método que obtém o *salt* de um utilizador da base de dados.

Método `userHasAccessPermission()`

Esta função auxilia a verificação do *login*. Começa por chamar o método `hasAccessPermission()` da classe *UserDAO* para verificar se a combinação do nome de utilizador e *password* está correta. Se não estiver correta, retorna -1, indicando que as credenciais do *login* estão erradas.

Se as credenciais estiverem corretas, é verificado se o dispositivo em que foi feito o *login* está registado na base de dados. Utiliza-se o método `isDeviceRegistered()` da classe *DeviceDAO* para este fim. Se o dispositivo estiver registado, retorna 1. Se não estiver registado, retorna 0. O trecho de código que se segue mostra a implementação desta função.

```
private int userHasAccessPermission(String userName, String
    userPassword, String deviceCode) {
    if (this.userDAO.hasAccessPermission(userName, userPassword)
    ) {
        if (this.devDAO.isDeviceRegistered(deviceCode)) {
            return 1;
        }
        // Dispositivo nao esta registado
        return 0;
    }
    // Se utilizador colocou login incorreto
    return -1;
}
```

Excerto de Código 6.9: Trecho de código do método que verifica o *login*.

Método `userWasRegistered()`

Serve para registar um utilizador na base de dados, bem como o seu dispositivo. Começa por fazer o registo do utilizador, chamando a função `wasRegistered()` da classe *UserDAO*, que vai inserir na base de dados o utilizador. Se esta operação for feita com sucesso, procede-se ao registo do dispositivo.

Começamos por obter o identificador do tipo de dispositivo na base de dados. Estes identificadores já estão registados na base de dados previamente. Como exemplo, o tipo *Android* tem o identificador número 1. De seguida, obtemos o identificador do utilizador. Com isto, podemos chamar o método `registerDevice()` da classe *DeviceDAO*, que vai inserir na base de dados as informações do dispositivo. Por último, registam-se as funcionalidades do dispositivo que vêm em formato JSON. Desta forma, utilizamos a classe *JSONParser* para as processar. O registo das funcionalidades é feito com a chamada ao método `registerDeviceFunctionalities()` da classe *DeviceFunctionalityDAO*. O trecho de código seguinte mostra como o registo de um utilizador está implementado.

```
private boolean userWasRegistered(String userName, String
    userPassword, String fullName, String salt, String email,
    String deviceType, String deviceCode, String deviceName,
    String funcs) {
    if (this.userDAO.wasRegistered(userName, userPassword,
        fullName, salt, email))
    {
        int deviceType_id = this.getDeviceTypeID(deviceType);
        int user_id = this.getUserID(userName);
        if (this.devDAO.registerDevice(deviceType_id, deviceCode
            , deviceName, user_id)) {
            int dev_id = this.devDAO.getDeviceID(deviceCode);
            JSONObject json = new JSONObject();
            try
            {
                JSONParser parser = new JSONParser();
                json = (JSONObject) parser.parse(funcs);
            }
            catch (ParseException e) {
                System.out.println("Parse exception " + e);
            }
            return this.dev_funcDAO.
                registerDeviceFunctionalities(dev_id, json);
        }
    }
    return false;
}
```

Excerto de Código 6.10: Trecho de código do método que regista um utilizador.

6.4.3 Alterações nos Métodos REST

Com o HTTPS habilitado e funções de criptografia implementadas, procedeu-se à alteração dos métodos REST que tratam das requisições do cliente ao servidor. Estes encontram-se na classe *WebServiceRestCommunication* do package *server.modules.communication.rest*.

O primeiro método a ser alterado foi o do *login*. Passou a ser uma *POST request* em vez de um *GET*. Além disso, é necessário agora calcular o valor do *hash* da *password* enviada para o método. Desta forma, começamos por obter o *salt* do utilizador com a função *getUserSalt()*. Fazemos o *hash* da *password* com o método *hashPassword()* da classe *CryptoServices* e chamamos a função *validateRemoteLoginUser()* da classe *Communication*, retornando o seu resultado (uma *String* em formato JSON). O trecho de código seguinte mostra como esta função foi implementada.

```
@POST
@Path("/validate")
@Produces("application/json")
public String validateRemoteLoginUser(
    @FormParam("login") String userName,
    @FormParam("password") String userPassword,
    @FormParam("device") String deviceCode) {

    Communication comm = new Communication(new
        PrivacyControlUbiquitous());
    String salt = comm.getUserSalt(userName);
    userPassword = CryptoServices.hashPassword(userPassword,
        salt);

    return comm.validateRemoteLoginUser(userName, userPassword,
        deviceCode);
}
```

Excerto de Código 6.11: Trecho de código do método REST que verifica o *login*.

Ainda não existia um método para registar um utilizador na base de dados, o qual foi feito no âmbito deste projeto. Para registar um novo usuário, necessitamos do seu *username*, *password*, *email*, nome completo, tipo de dispositivo, código do dispositivo, nome do dispositivo e as funções do dispositivo. Todos estes parâmetros são passados ao método como *POST*. Geramos um *salt* para este utilizador e calculamos o *hash* da *password* introduzida por ele. O registo do novo usuário na base de dados é feito com recurso ao método *registerUser()* da classe *Communication*, retornando o valor de retorno da mesma. O trecho de código seguinte mostra como este método REST foi implementado.

```
@POST
@Path("/register")
@Produces("application/json")
public String registerUser(
    @FormParam("login") String userName,
    @FormParam("password") String userPassword,
    @FormParam("name") String fullName,
    @FormParam("email") String email,
    @FormParam("deviceType") String deviceType,
    @FormParam("deviceCode") String deviceCode,
    @FormParam("deviceName") String deviceName,
    @FormParam("funcs") String funcs) {

    byte[] salt = CryptoServices.generateSalt();
    String b64salt = Base64.getEncoder().encodeToString(salt);
    userPassword = CryptoServices.hashPassword(userPassword,
        b64salt);
    String hexSalt = new String(Base64.getEncoder().encode(salt));

    Communication comm = new Communication(new
        PrivacyControlUbiquitous());
    return comm.registerUser(userName, userPassword, fullName,
        hexSalt, email, deviceType, deviceCode, deviceName, funcs);
}
```

Excerto de Código 6.12: Trecho de código do método REST que registra um usuário.

A função que trata das ações do dispositivo conforme o seu ambiente também teve que ser alterada. Em vez de se enviar a *password* do utilizador para validar a requisição, passou-se a enviar o *token* JWT. Se o *token* for válido, retornamos o valor da função `onChangeCurrentUserLocalizationWithResponse()` da classe *Communication*. Caso contrário, retornamos uma *String* em formato JSON a indicar que a requisição foi recusada. O trecho de código que se segue mostra as alterações feitas a esta função.

```
@POST
@Path("/change/location/json/response")
@Consumes("application/json")
@Produces("application/json")
public String onChangeCurrentUserLocalizationWithResponse(
    @FormParam("environmentID") String environmentId,
    @FormParam("userName") String userName,
    @FormParam("token") String token,
```

```

        @FormParam ("deviceCode") String deviceCode) {
    if (CryptoServices.validateToken(userName, token))
    {
        Communication comm = new Communication(new
            PrivacyControlUbiquitous());
        return comm.onChangeCurrentUserLocalizationWithResponse(
            Integer.parseInt(environmentId), userName,
            deviceCode, false);
    }
    return "{ \"status\": \"DENY\" } ";
}

```

Excerto de Código 6.13: Trecho de código do método que trata das ações para um dado ambiente.

Foram, ainda, adicionados mais 3 métodos REST que auxiliam o funcionamento do UbiPri. Um regista um dispositivo, outro verifica se o utilizador já existe e o último valida um *token*. A implementação destas funções é bastante simples e segue a mesma lógica dos métodos descritos anteriormente, como se pode ver pelo trecho de código seguinte.

```

@POST
@Path("/checkuser")
@Produces("application/json")
public String userExists(
    @FormParam("username") String userName) {

    Communication comm = new Communication(new
        PrivacyControlUbiquitous());
    return comm.userExists(userName);
}

@POST
@Path("/registerDevice")
@Produces("application/json")
public String registerDevice(
    @FormParam("username") String userName,
    @FormParam("token") String token,
    @FormParam("deviceType") String deviceType,
    @FormParam("deviceCode") String deviceCode,
    @FormParam("deviceName") String deviceName,
    @FormParam("funcs") String funcs) {

    if (!CryptoServices.validateToken(userName, token))
    {
        return "{ \"status\": \"DENY\" } ";
    }
}

```

```
        Communication comm = new Communication(new
            PrivacyControlUbiquitous());
        return comm.registerDevice(userName, deviceType, deviceCode,
            deviceName, funcs);
    }

    @POST
    @Path("/validateToken")
    @Produces("application/json")
    public String validateToken(
        @FormParam("username") String userName,
        @FormParam("token") String token) {

        if (CryptoServices.validateToken(userName, token))
        {
            return "{\"status\":\"OK\"}";
        }

        return "{\"status\":\"DENY\"}";
    }
}
```

Excerto de Código 6.14: Trecho de código dos métodos auxiliares.

6.5 Conclusões

Neste capítulo, foi descrita a implementação do módulo de segurança no servidor do UbiPri de maneira a proteger os dados dos utilizadores e do próprio servidor. As falhas de segurança detetadas foram corrigidas, o que nos permite agora implementar a aplicação *Android* do UbiPri com a garantia de que as comunicações entre a aplicação e o servidor cumprem as regras básicas de encriptação e segurança.

Capítulo 7

Alterações à Aplicação *Android* do UbiPri

7.1 Introdução

Neste capítulo vão ser descritas todas as alterações feitas à aplicação para *Android* do UbiPri, que já estava implementada com funcionalidades ainda muito básicas. A secção 7.2 apresenta a estrutura da aplicação, falando dos seus componentes e organização do código. A secção 7.3 explica como a atividade do registo foi implementada. De igual modo, na secção 7.4 é exposta a implementação dos serviços de criptografia. As secções 7.5, 7.6 e 7.7 explicam as alterações feitas às atividades do *login*, da localização e do menu principal. A secção 7.8 apresenta a implementação do serviço de localização, enquanto que a 7.9 fala sobre as tarefas assíncronas da aplicação. Por último, a secção 7.10 apresenta as conclusões.

7.2 Estrutura da Aplicação

A aplicação é composta por 6 componentes, das quais 5 são atividades e 1 é um serviço. Existem atividades para fazer o *login*, registo, menu principal, visualizar informações do ambiente atual e visualizar informações do dispositivo. O serviço corre no *background* depois de fazer o *login*, detetando mudanças no ambiente em que o utilizador se encontra. As seguintes subsecções apresentam a organização dos *packages* que constituem a aplicação.

7.2.1 *Package br.ufrgs.inf.ubipri.util*

Tem a classe *Config* que disponibiliza variáveis estáticas que auxiliam nas funcionalidades da aplicação. A classe *CryptoServices* também se encontra neste

package, onde se encontram implementados métodos criptográficos para proteção de dados.

7.2.2 *Package br.ufrgs.inf.ubipri.client*

Contém as classes relativas às atividades e ao serviço implementados. Também tem a classe *BDHelper* que gere a base de dados local da aplicação.

7.2.3 *Package br.ufrgs.inf.ubipri.client.model*

Contém modelos de classe para representar informação útil disponível na base de dados do servidor. Semelhante ao *package server.model* descrito em 6.2.3.

7.2.4 *Package br.ufrgs.inf.ubipri.client.dao*

Semelhante ao *package server.dao* descrito em 6.2.2. Contém classes que comunicam com as bases de dados locais da aplicação.

7.2.5 *Package br.ufrgs.inf.ubipri.client.communication*

Um dos *packages* mais importantes da aplicação. Contém todas as classes que estabelecem a conexão ao servidor, processando as requisições necessárias e lendo os resultados retornados pelo UbiPri.

7.3 Implementação da Atividade de Registo

A aplicação não possuía uma atividade para registar um utilizador, pelo que teve de ser implementada. A atividade contém campos para introduzir o *username*, nome completo, *email*, *password* e confirmação de *password*, além de 2 botões para efetuar o registo e para voltar à atividade de *login*.

Para fazer a operação de registo, começa-se por obter os valores nos campos da atividade. É chamado o método *checkFields()* que simplesmente verifica se existe algum campo vazio. Em seguida, a aplicação verifica se o nome de utilizador já existe chamando a função *checkUserExists()* da classe *Communication*. Depois verifica se a *password* foi inserida de modo igual em ambos os campos. Se estiverem iguais, é confirmado se a *password* é forte o suficiente com recurso a uma expressão regular. Antes de estabelecer a conexão ao servidor, a aplicação verifica ainda se um *email* válido foi inserido com recurso ao *android.util.Patterns*.

Se nenhuma destas verificações falhar, é feito o registo do utilizador com o método *remoteRegister()* da classe *Communication*. Se não ocorrer nenhum erro

durante o processo de registo, a atividade é finalizada e é chamada a atividade do *login*. Caso contrário, aparece uma *toast* a dizer que o registo falhou. A figura 7.1 mostra a atividade do registo.

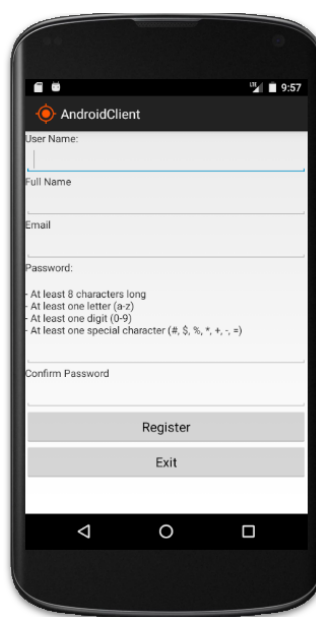


Figura 7.1: Atividade do registo de um utilizador.

7.4 Implementação da Classe *CryptoServices*

Esta classe foi implementada de maneira a termos funcionalidades criptográficas na aplicação do UbiPri.

Começou-se por definir 2 variáveis de classe, uma pública e uma privada. A privada tem o tipo *KeyStore* enquanto que a pública é um *byte array* de tamanho 12 que representa um IV para encriptação.

O principal objetivo desta classe é encriptar o *token* JWT gerado pelo servidor e enviado para o utilizador. O *token* tem que ser guardado no dispositivo do utilizador, e como permite a autenticação do mesmo, decidiu-se recorrer à sua encriptação. No entanto, também tem que ser guardada a chave da sua encriptação. Para isto, recorreremos à API do *Android Keystore* que serve para guardar chaves de encriptação de um modo seguro.

Assim, foi definido o método `initKeyStore()` que inicializa a *keystore* da aplicação. É também gerada uma chave com a qual serão encriptados os *tokens*. O *token* será encriptado com o algoritmo AES em modo GCM. Infelizmente, a API

Android Keystore ainda não tem suporte a outros algoritmos de chave simétrica. O excerto de código que se segue mostra a implementação da função agora descrita.

```
public static void initKeyStore() {
    try {
        ks = KeyStore.getInstance("AndroidKeyStore");
        ks.load(null);

        if (!ks.containsAlias("tokenKey"))
        {
            KeyGenerator kg = KeyGenerator.getInstance(
                KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

            KeyGenParameterSpec kgps = new KeyGenParameterSpec.
                Builder("tokenKey", KeyProperties.PURPOSE_ENCRYPT
                    | KeyProperties.PURPOSE_DECRYPT)
                .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
                .setEncryptionPaddings(KeyProperties.
                    ENCRYPTION_PADDING_NONE)
                .setRandomizedEncryptionRequired(false)
                .build();

            kg.init(kgps);
            SecretKey sk = kg.generateKey();

            KeyStore.SecretKeyEntry ske = new KeyStore.
                SecretKeyEntry(sk);
            ks.setKeyEntry("tokenKey", ske.getSecretKey().
                getEncoded(), null);
        }
    }
    catch (KeyStoreException | IOException |
        CertificateException | NoSuchAlgorithmException |
        NoSuchProviderException |
        InvalidAlgorithmParameterException e) { }
```

Excerto de Código 7.1: Trecho de código do método `initKeyStore()`.

Definiu-se também um método para gerar um IV, que simplesmente gera *bytes* aleatórios com a classe *SecureRandom*.

Por último, temos 2 funções que encriptam e desencriptam uma dada mensagem. Ambas estão implementadas de forma semelhante. Primeiro, obtém-se a chave da *Keystore* com o método `getKey()`. Esta chave é guardada num objeto do tipo *SecretKey*. De seguida, declaramos um objeto *Cipher* que é inicializado com uma instância do AES em modo GCM sem *padding*. Por último, atribui-se o IV

a ser utilizado e chama-se o método `init()` para fazer a encriptação ou desencriptação. A mensagem é então retornada encriptada ou desencriptada. O trecho de código seguinte mostra o método que trata da encriptação de uma mensagem.

```
public static String encrypt(String msg) {
    try
    {
        SecretKey sk = (SecretKey) ks.getKey("tokenKey", null);

        Cipher cp = Cipher.getInstance("AES/GCM/NoPadding");
        GCMParameterSpec gcm = new GCMParameterSpec(128, iv);
        cp.init(Cipher.ENCRYPT_MODE, sk, gcm);

        return Base64.encodeToString(cp.doFinal(msg.getBytes()),
            Base64.DEFAULT);
    }
    catch (KeyStoreException | NoSuchAlgorithmException |
        UnrecoverableEntryException | NoSuchPaddingException |
        InvalidKeyException | BadPaddingException |
        IllegalBlockSizeException |
        InvalidAlgorithmParameterException e) {
        Log.e("Encryption error: ", e.getMessage());
    }

    return "error";
}
```

Excerto de Código 7.2: Trecho de código do método `encrypt()`.

7.5 Alterações à Atividade de *Login*

O código relacionado à atividade inicial da aplicação do UbiPri teve que sofrer algumas alterações, visto que continha algumas falhas de segurança ou funcionalidades mal implementadas.

Foi implementada uma função `initRoutines()` que define rotinas de inicialização quando a aplicação é iniciada. Ela é responsável por instanciar *Shared Preferences*, gerar um IV para encriptação e desencriptação de *tokens* e tentar fazer o *login* de forma automática se já existir um *token* e utilizador associado ao dispositivo.

Quando a aplicação é iniciada pela primeira vez, também são mostrados diálogos a pedir ao utilizador para permitir acesso à localização, visto que nas versões mais recentes do *Android* este tipo de permissões têm de ser atribuídas explicitamente em tempo de execução. A aplicação é terminada se o utilizador não

conceder esta permissão, já que a aplicação só funciona se souber a localização do utilizador.

No método que faz a operação de *login*, foram feitas algumas alterações importantes. Em primeiro lugar, a aplicação estava a guardar numa base de dados local o nome de utilizador e a sua *password* em texto limpo para fazer *logins* sem se conectar ao servidor. Isto foi completamente removido visto que pode introduzir graves problemas de segurança. A base de dados local do utilizador não é mais utilizada e será removida em versões futuras da aplicação.

Outra alteração a este método foi a introdução da opção de registar o dispositivo se for a primeira vez que o utilizador faz *login* a partir dele. Se o utilizador não quiser registar este dispositivo em seu nome, não poderá utilizar a aplicação nele. Por último, se o *login* é bem sucedido, o dispositivo recebe um *token* do servidor. Este é encriptado com as funções da classe *CryptoServices* e guardado nas *Shared Preferences* em modo privado para poder ser usado em requisições futuras. O trecho de código seguinte mostra parte da função que realiza o *login* de um utilizador.

```
try {
    result = communication.remoteLogin(userName, userPassword,
        Config.DEVICE_CODE);
    status = (JSONObject) new JSONTokener(result).nextValue();
    login_result = status.getString("status");

    switch(login_result)
    {
        case "DENY":
            break;
        case "CHECK":
        case "OK":
            Config.token = status.getString("token");
            token = CryptoServices.encrypt(Config.token);
            editor.putString("token_enc", token);
            editor.commit();
            break;
    }
}
```

Excerto de Código 7.3: Trecho de código do método *login()*.

7.6 Alterações à Atividade da Localização

Anteriormente, esta atividade implementava um *LocationListener* que detetava mudanças de localização. Continha ainda todos os métodos que tratavam da mu-

dança de ambiente, incluindo as ações a serem feitas caso isso acontecesse.

O problema disto é que a atividade necessitava de estar focada para o dispositivo detetar mudanças no ambiente. Isto tornaria o uso da aplicação bastante limitado. Desta forma, as funcionalidades de mudança de localização foram colocadas num serviço. Esta atividade agora limita-se apenas a mostrar a informação do ambiente atual.

7.7 Alterações à Atividade do Menu Principal

O menu principal apenas servia para aceder às atividades da localização e da informação do dispositivo, para além de permitir fazer o *logout*. Foi alterado de maneira a despoletar o serviço de localização e a verificar a validade do *token* sempre que a atividade é resumida. O excerto de código que se segue mostra o *override* feito ao método *onResume()*.

```
@Override
protected void onResume() {
    super.onResume();
    try
    {
        if (!communication.tokenLogin(Config.LOGGED_USER_NAME,
            Config.token)) { // token nao e valido
            btnLogout.callOnClick(); // simula logout
        }
    }
    catch (InterruptedException | ExecutionException |
        JSONException e) {
        Log.e("Token login ", "Error: ", e);
    }
}
```

Excerto de Código 7.4: Trecho de código do método *onResume()* do menu principal.

7.8 Implementação do Serviço *LocationService*

O serviço implementa um *LocationListener* para podermos detetar mudanças na localização do utilizador pelo *Global Positioning System* (GPS). O método *onStartCommand()* cria uma nova *thread* onde é corrido num ciclo infinito a deteção da localização atual do utilizador. Dentro deste ciclo está ainda um *Runnable* que executa a verificação da localização. A *thread* dorme por 5 segundos ao fim de

cada verificação. O trecho de código seguinte mostra a implementação do `onStartCommand()`.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId)
{
    new Thread() {
        public void run() {
            while(running)
            {
                Runnable oRun = new Runnable() {
                    @Override
                    public void run() {
                        int access_fine = checkSelfPermission(
                            Manifest.permission.
                            ACCESS_FINE_LOCATION);
                        int access_coarse = checkSelfPermission(
                            Manifest.permission.
                            ACCESS_COARSE_LOCATION);
                        if (access_fine == PackageManager.
                            PERMISSION_GRANTED && access_coarse
                            == PackageManager.PERMISSION_GRANTED)
                        {
                            locationManager.
                                requestLocationUpdates(provider,
                                    400, 1, LocationService.this);
                            locationManager.getLastKnownLocation(
                                provider);
                        }
                    }
                };
                oHandler.post(oRun);

                try
                {
                    Thread.sleep(5000);
                }
                catch (InterruptedException e) {}
            }
        }
    }.start();
    return Service.START_STICKY;
}
```

Excerto de Código 7.5: Trecho de código do método `onStartCommand()`.

Caso a localização mude, o método `onLocationChanged()` é chamado automaticamente pelo *LocationListener*. O *override* deste método já se encontrava implementado na atividade da localização, tendo sido apenas inserido na classe do serviço juntamente com os métodos `applyActions()`, `applyAction()` e `isActionFunctionality()`.

A função `onDestroy()` do serviço foi alterada de forma a parar o ciclo infinito do `onStartCommand()`. A variável *running*, que controla esse ciclo, é colocada a *false*. Além disso, chamamos o método `removeUpdates()` para prevenir o *LocationListener* de continuar a verificar se o utilizador mudou de localização. O excerto de código que se segue mostra como foi implementado o `onDestroy()`.

```
@Override
public void onDestroy() {
    locationManager.removeUpdates(this);
    running = false;
    super.onDestroy();
}
```

Excerto de Código 7.6: Trecho de código do método `onDestroy()`.

7.9 Implementação das Tarefas Assíncronas

A aplicação dispõe de *AsyncTasks* para estabelecer conexões com o servidor. As tarefas já existentes foram alteradas para usarem *HttpsURLConnection* em vez de *HttpURLConnection*. Além disso, foram adicionadas as tarefas *TaskCheckUserExists*, *TaskRegisterDevice* e *TaskTokenLogin* visto que se encontravam em falta e eram necessárias para suportar todas as funcionalidades da aplicação.

A implementação destas tarefas seguem todas a mesma lógica. Constrói-se uma *String* com os parâmetros a passar ao servidor, define-se o *URL* da requisição a fazer ao servidor e instância-se uma conexão do tipo *HttpsURLConnection*. Escrevem-se os parâmetros para a conexão através de uma *DataOutputStream* e, por fim, lê-se o retorno do servidor com um *BufferedReader*. O excerto de código seguinte mostra parte do código da tarefa que trata do *login*.

```
try
{
    String login_params = "login="+params[0]+"&password="+params
        [1]+"&device="+params[2];
    uri = Config.SERVER_HOST+"webresources/rest/validate";
    URL url = new URL(uri);
    SSLContext context = TrustCert.trustMyCert();
```

```
    HttpURLConnection conn = (HttpURLConnection) url.
        openConnection();
    conn.setSSLSocketFactory(context.getSocketFactory());
    conn.setDoOutput(true);
    conn.setRequestMethod("POST");
    conn.setRequestProperty("Accept", "application/json");
    conn.setRequestProperty("Content-Type", "application/
        json");
    conn.connect();

    DataOutputStream wr = new DataOutputStream(conn.
        getOutputStream());
    wr.writeBytes(login_params);
    wr.flush();
    wr.close();
    ...
    BufferedReader br = new BufferedReader(new InputStreamReader
        ((conn.getInputStream())));

    String output;

    while ((output = br.readLine()) != null) {
        result += output;
    }
    conn.disconnect();
}
...
return result;
}
```

Excerto de Código 7.7: Trecho de código do método `doInBackground()` da tarefa do *login*.

7.10 Conclusões

Este capítulo mostrou todas as alterações e adições feitas à aplicação *Android* do UbiPri. Foram corrigidas falhas de segurança, adicionados módulos de segurança, novas funcionalidades e implementou-se um serviço para tornar a detecção de localização mais flexível. A aplicação do UbiPri tem agora todas as funcionalidades básicas implementadas, estando ainda garantida a segurança dos dados trocados entre o utilizador e o servidor.

Capítulo 8

Conclusões e Trabalho Futuro

8.1 Conclusões Principais

Os testes realizados na primeira fase deste projeto demonstraram que a eficiência dos algoritmos criptográficos pode depender bastante do tipo de dispositivo em que são utilizados. Para criptografia de chave simétrica, algoritmos como o RC6 e o *Twofish* parecem ser boas alternativas ao AES em certos ambientes, tais como microprocessadores da arquitetura *ARM*. No contexto de uma aplicação ubíqua, esta deve detetar as características do dispositivo e atribuir o algoritmo a ser utilizado conforme o seu *hardware* e linguagem de programação utilizada para implementar os algoritmos criptográficos.

Quanto aos esquemas de múltipla encriptação definidas nos níveis de acesso do UbiPri, ainda se pensou aplicar estes nas comunicações entre o servidor e aplicação *Android* do *middleware*. No entanto, chegámos à conclusão que isto não iria trazer quaisquer benefícios a nível de segurança.

Em primeiro lugar, tanto o servidor como o cliente teriam que ter as chaves simétricas necessárias para encriptar e desencriptar as mensagens trocadas. Estas teriam que ser trocadas entre o servidor e o cliente à semelhança do que acontece no protocolo TLS. Ambos geravam um par de chaves pública e privada, encriptando as chaves simétricas e trocando-as de modo seguro entre eles. No entanto, se a comunicação é, de alguma forma, comprometida, o atacante teria acesso a todas as chaves trocadas entre o cliente e o servidor. Assim, é fácil de perceber que o esquema de múltipla encriptação não tornaria mais seguro a troca das mensagens. Este esquema também tornaria as comunicações mais lentas sem necessidade, visto que cada mensagem teria que ser encriptada e desencriptada duas ou mais vezes.

Um contexto onde a encriptação múltipla poderia ser útil seria se o servidor necessitasse de armazenar dados muito sensíveis do utilizador. Deste contexto,

observávamos dois cenários possíveis:

1. Apenas o servidor necessitaria de aceder a estes dados no futuro. Desta forma, guardavam-se as chaves simétricas necessárias para descriptar os dados sem nunca as trocar com o cliente ou outra entidade externa do servidor.
2. Apenas o cliente necessitaria de aceder a estes dados futuramente. O utilizador guardaria no seu dispositivo as chaves simétricas necessárias, nunca as trocando com o servidor ou outra entidade externa. Quando necessitasse de aceder aos dados, ele pedia-os ao servidor e procedia à descriptação dos mesmos localmente.

Alternativamente, para evitar guardar as chaves num meio de armazenamento permanente, estas podiam ser derivadas a partir de uma função de derivação com um segredo (por exemplo, uma *password*). No entanto, surgia a desvantagem de que se o segredo fosse esquecido, a informação nunca mais podia ser descriptada novamente.

Ainda sobre a encriptação múltipla, também acabou por se concluir que seria melhor ideia utilizar uma mistura dos algoritmos como *AES+RC6+Twofish* do que utilizar um esquema do tipo *Twofish+Twofish+Twofish*. Por exemplo, se fosse encontrada uma falha de segurança no RC6 que o quebrasse, ainda tínhamos o *Twofish* e o AES a proteger os dados encriptados.

Por último, se formos utilizar diferentes tamanhos de chave numa mistura de algoritmos criptográficos, prevê-se que para obter a melhor eficiência, o algoritmo com tempo de execução mais rápido deve ficar com o tamanho de chave maior. No processador da Intel[®] utilizado para correr os testes, isto significa que atribuíamos uma chave de 256 *bits* ao AES, visto que foi o algoritmo mais rápido. O *Twofish* ficaria com uma chave de 192 *bits*. Já ao RC6 atribuía-se um tamanho de chave de 128 *bits*, visto que foi o mais lento.

Assim, não parecem haver muitas razões e situações onde a encriptação múltipla se justifique. Se vamos atribuir diferentes níveis de criptografia conforme os privilégios de um utilizador num dado ambiente, parece ser mais vantajoso variar somente o tamanho de chave. Quanto ao algoritmo a utilizar, isso dependeria do dispositivo a realizar as operações criptográficas, tal como referido anteriormente.

Quanto à implementação do módulo de segurança no UbiPri, puderam-se explorar muitas maneiras de proteger os dados. A habilitação do protocolo HTTPS, a utilização da função *state of the art Argon2id* para *hash* de *passwords* e o uso de *tokens* JWT garantiram uma base robusta para a segurança da informação no *middleware* UbiPri.

8.2 Trabalho Futuro

No futuro, pretende-se avaliar outros aspetos do desempenho dos algoritmos criptográficos testados, principalmente o consumo energético. Além disso, seria interessante correr os testes num microprocessador *ARM* ou noutros tipos de microcontroladores físicos para comparar ou confirmar os resultados obtidos na emulação.

Outro aspeto importante a ser analisado seria avaliar a cifra contínua *ChaCha20* e comparar com as cifras de bloco testadas. Desta forma, pretendemos verificar se e quando é vantajoso utilizar cifras contínuas em vez de cifras de bloco.

Para o *middleware* UbiPri, ainda existem módulos e muitas outras funcionalidades a serem implementadas. Pretendemos continuar o desenvolvimento deste *middleware*, adicionando novas camadas de privacidade e novos métodos de a proteger. No servidor e aplicação, ficaram por implementar alguns mecanismos, nomeadamente recuperações e alterações de *password*, eliminação de contas, desativação de dispositivos já registados e uma maneira de invalidar *tokens*. Este último é absolutamente necessário se uma *password* for alterada ou se um dispositivo for desativado, visto que o *token* não pode ser mais válido nesses contextos.

Na aplicação para *Android* do UbiPri, apenas o algoritmo AES é utilizado para encriptação dos dados. Pretendem-se explorar alternativas no futuro, aplicando os algoritmos com base nos testes realizados e nível de acesso do utilizador. A razão pela qual outros algoritmos não foram utilizados com base no dispositivo e privilégios do usuário foi devido à falta de tempo para encontrar outros métodos de os aplicar, visto que a *Android KeyStore* API apenas tem suporte para o AES. Por último, será considerada a possibilidade de implementar a aplicação UbiPri para outros sistemas operativos, nomeadamente o *iOS*.

Bibliografia

- [1] David A. McGrew and John Viega. The Galois/counter mode of operation (GCM), 2005. [Online] <https://pdfs.semanticscholar.org/b4c4/66e7158c158fb513b729d6302521017d72fa.pdf>. Último acesso a 23 de Março de 2019.
- [2] D. J. Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*, 2008.
- [3] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 292–302, March 2016.
- [4] Password Hashing Competition. Argon2, 2016. [Online] <https://github.com/p-h-c/phc-winner-argon2>. Último acesso a 10 de Maio de 2019.
- [5] Intersoft Consulting. General Data Protection Regulation (GDPR), 2016. [Online] <https://gdpr-info.eu/>. Último acesso a 28 de Maio de 2019.
- [6] B. A. da Silva, I. S. Ochôa, and V. Leithardt. Estudo de algoritmos criptográficos simétricos na placa beaglebone black, 2018. [Online] https://www.setrem.com.br/erad2019/data/pdf/forum_ic/192077.pdf. Último acesso a 20 de Maio de 2019.
- [7] Wei Dai. Crypto++, 2019. [Online] <https://www.cryptopp.com/>. Último acesso a 20 de Março de 2019.
- [8] F. De Santis, A. Schauer, and G. Sigl. ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 692–697, March 2017.
- [9] M. El-Hajj, M. Chamoun, A. Fadlallah, and A. Serhrouchni. Analysis of Cryptographic Algorithms on IoT Hardware platforms. In *2018 2nd Cyber Security in Networking Conference (CSNet)*, pages 1–5, Oct 2018.

- [10] Jwtk. JWT, 2014. [Online] <https://github.com/jwtk/jjwt>. Último acesso a 14 de Maio de 2019.
- [11] N. Khan, N. Sakib, I. Jerin, S. Quader, and A. Chakrabarty. Performance analysis of security algorithms for iot devices. In *2017 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*, pages 130–133, Dec 2017.
- [12] V. Leithardt. Ubipri : middleware para controle e gerenciamento de privacidade em ambientes ubíquos, 2015. Dissertação de Doutorado, Universidade Federal do Rio Grande do Sul, Instituto de Informática [Online] <http://hdl.handle.net/10183/147774>. Último acesso a 27 de Maio de 2019.
- [13] Valderi Leithardt, Guilherme Borges, Anubis Rossetto, Carlos Rolim, Claudio Geyer, Luiz Correia, David Nunes, and Jorge Sá Silva. A privacy taxonomy for the management of ubiquitous environments. 12 2013.
- [14] Team libtom. LibTomCrypt, 2019. [Online] <https://www.libtom.net/LibTomCrypt/>. Último acesso a 16 de Março de 2019.
- [15] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-centered and privacy-driven process mining system design for iot. In Cinzia Cappiello and Marcela Ruiz, editors, *Information Systems Engineering in Responsible Information Systems*, pages 194–206, Cham, 2019. Springer International Publishing.
- [16] I. S. Ochôa, V. R. Q. Leithardt, C. A. Zeferino, and J. S. Silva. Data transmission performance analysis with smart grid protocol and cryptography algorithms. In *2018 13th IEEE International Conference on Industry Applications (INDUSCON)*, pages 482–486, Nov 2018.
- [17] Vishwa Pratap Singh and R. L. Ujjwal. Privacy attack modeling and risk assessment method for name data networking. In Sanjiv K. Bhatia, Shailesh Tiwari, Krishn K. Mishra, and Munesh C. Trivedi, editors, *Advances in Computer Communication and Computational Sciences*, pages 109–119, Singapore, 2019. Springer Singapore.
- [18] B. Soewito, F. E. Gunawan, Diana, and A. Antonyová. Power consumption for security on mobile devices. In *2016 11th International Conference on Knowledge, Information and Creativity Support Systems (KICSS)*, pages 1–4, Nov 2016.

-
- [19] R. Williams, E. McMahon, S. Samtani, M. Patton, and H. Chen. Identifying vulnerabilities of consumer internet of things (iot) devices: A scalable approach. In *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 179–181, July 2017.
 - [20] B. Yilmaz and S. Özdemir. Performance comparison of cryptographic algorithms in internet of things. In *2018 26th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4, May 2018.
 - [21] Yandong Zheng, Rongxing Lu, Beibei Li, Jun Shao, Haomiao Yang, and Kim-Kwang Raymond Choo. Efficient privacy-preserving data merging and skyline computation over multi-source encrypted data. *Information Sciences*, 498, 05 2019.