Pós-Graduação em Ciência da Computação

THIAGO DE OLIVEIRA CAVALCANTE

SISTEMA DE COMUNICAÇÃO SEGURA PARA DISPOSITIVOS CONECTADOS À INTERNET DAS COISAS COM UTILIZAÇÃO DE SMART CARDS

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2017

Thiago de Oliveira Cavalcante

Sistema de Comunicação Segura para Dispositivos Conectados à Internet das Coisas com Utilização de Smart Cards

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.

ORIENTADOR: **Prof. Djamel Fawzi Hadj Sadok**

RECIFE
2017

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

C376s Cavalcante, Thiago de Oliveira
Sistema de comunicação segura para dispositivos conectados à Internet das coisas com utilização de smart cards / Thiago de Oliveira Cavalcante. – 2017.
104 f.: il., fig., tab.

Orientador: Djamel Fawzi Hadj Sadok.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências e apêndice.

1. Redes de computadores. 2. Internet das coisas. I. Sadok, Djamel Fawzi Hadj (orientador). II. Título.

004.6 CDD (23. ed.) UFPE- MEI 2017-247

Thiago de Oliveira Cavalcante

Sistema de Comunicação Segura para Dispositivos Conectados à Internet das Coisas com Utilização de Smart Cards

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 14/09/2017.

BANCA EXAMINADORA

_____

Prof. Dr. Odilon Maroja da Costa Pereira Filho
Centro de Informática/UFPE

_____

Prof. Dr. Carmelo José Albanez Bastos Filho
Escola Politécnica de Pernambuco / UPE

_____

Prof. Dr. Djamel Fawzi Hadj Sadok
Centro de Informática / UFPE
(Orientador)

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Rúben e Gicelma, e às minhas irmãs, Ana Letícia e Ana Clara, pelo amor, carinho e suporte incondicionais durante toda minha vida.

Agradeço também à Lidiane por ser minha companheira em todas as horas e por ter me ajudado em alguns dos momentos mais complicados dessa jornada.

Sou muito grato ao Prof. Djamel e à Profª. Judith por me darem essa oportunidade, pela paciência, compreensão, preocupação, pelos conselhos e pela confiança que eles depositaram em mim e em meu trabalho. Agradeço também à Andrea por estar sempre disposta a ajudar e por ter sido o ponto de partida dessa empreitada. Por fim, agradeço aos meus colegas de trabalho do GPRT pela amizade e suporte e também pelas piadas, lanchinhos e caronas.

Science is a cooperative enterprise, spanning the generations. It's the passing of a torch from teacher, to student, to teacher. A community of minds reaching back to antiquity and forward to the stars.

—NEIL DEGRASSE TYSON

RESUMO

Este trabalho tem como objetivo proteger a comunicação entre dispositivos conectados à Internet das Coisas, do inglês *Internet of Things* (IoT), através da integração entre microcontroladores e *Smart Cards* (SCs), cartões de plástico nos quais estão embutidos *chips* criptográficos invioláveis, atualmente utilizados em aplicações que exigem um alto nível de segurança (e.g., bancos). É proposta uma arquitetura, a qual envolve projetos de *hardware* e *software*, para um sistema que estabelece uma comunicação autenticada e criptografada, baseada no Protocolo *Transport Layer Security* (TLS), entre dispositivos IoT e um servidor. O foco do trabalho está em placas de desenvolvimento de baixo custo. Testes foram realizados inicialmente no *Arduino UNO* e o dispositivo final possui o microcontrolador *ESP8266* (em específico, o módulo ESP-12E), que possui Wi-Fi integrado, o que facilita a sua inclusão na IoT, e é simples de programar. Adicionalmente, é utilizado um SC com a tecnologia *Java Card*, que torna mais simples o desenvolvimento e a instalação de programas (conhecidos como *applets*) no cartão. Nele está instalada uma versão modificada do *IsoApplet*, um programa *open source* em desenvolvimento que permite a execução de tarefas criptográficas, implementado de acordo com os padrões ISO7816. Assim, a execução de operações essenciais na implementação de uma infra-estrutura de segurança como geração de chaves, cifragem e decifragem (em ambas criptografias simétrica e assimétrica), assinatura digital e armazenamento seguro de dados (e.g., chaves secretas, certificados) é delegada pelo microcontrolador ao cartão, que possui *hardware* especializado. O microcontrolador, por sua vez, pode ser ligado a sensores e se conectar de forma autenticada com um servidor, enviando informações criptografadas. Por fim, demonstra-se que é possível construir um dispositivo conectado à Internet das Coisas, capaz de enviar mensagens de forma segura, a partir da integração entre microcontroladores de baixo custo e *Smart Cards*. Uma análise de custo do dispositivo construído, mostra que o mesmo pode ter um preço compátivel com o mercado, se produzido em larga escala. Uma segunda análise, relativa ao consumo de energia da placa, mostra que, a depender do tipo de aplicação, o dispositivo pode funcionar com bateria por dias. As contribuições deste trabalho, além da fabricação do próprio dispositivo IoT, incluem o desenvolvimento de bibliotecas que habilitam a comunicação entre microcontroladores (compatíveis com Arduino) e *Smart Cards* e a expansão de um *software open-source* para Java Cards com funções criptográficas associadas ao TLS.

Palavras-chave: Internet das Coisas. Segurança. Smart Card. Java Card. Microcontroladores

ABSTRACT

This work aims to secure the communication between devices connected to the *Internet of Things* (IoT) by integrating microcontrollers and *Smart Cards* (SCs), plastic cards in which are embedded cryptographic tamper-resistant chips, currently used in applications that require a high level of security (e.g., banks). An architecture, which involves hardware and software projects, is proposed for a system that establishes an encrypted and authenticated communication, based on *Transport Layer Security* (TLS) Protocol, between IoT devices and a server, focusing on low-cost development boards. Tests were performed initially on *Arduino UNO* boards and the final device has an *ESP8266* microcontroller (specifically, an ESP-12E module), which has integrated Wi-Fi capabilities and is simple to program. Additionally, the SC used is *Java Card*-based, which simplifies the development and installation of programs (known as applets) on the card. It contains a modified version of *IsoApplet*, an open source program under development that allows the realization of cryptographic tasks, implemented according to ISO7816 standards. Thus, the execution of essential operations in the implementation of a security infrastructure such as key generation, encryption and decryption (in both symmetric and asymmetric cryptography), digital signature and secure data storage (e.g., secret keys, certificates) is delegated by the microcontroller to the card, which has specialized hardware. The microcontroller, in turn, can be connected to sensors and connects in an authenticated way to a server, sending encrypted data. Finally, it is shown that it is possible to build a device connected to the Internet of Things, which is able to send messages safely, by integrating low-cost microcontrollers and *Smart Card*s. A cost analysis of the device shows that it can have a market-compatible price if produced on a large scale. A second analysis, regarding the power consumption of the board, shows that, depending on the type of application, the device can run on battery for days. The contributions of this work, in addition to the manufacture of the IoT device itself, include the development of libraries that enable communication between microcontrollers (compatible with Arduino) and *Smart Card*s and the expansion of open-source software for Java Cards by adding cryptographic functions associated with TLS.

Keywords: Internet of Things. Security. Smart Card. Java Card. Microcontrollers

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

3DES *Triple DES*

AES *Advanced Encryption Standard*

AEAD *Authenticated Encryption with Associated Data*

AODF *Authentication Object Directory File*

APDU *Application Protocol Data Unit*

API *Application Programming Interface*

ASCII *American Standard Code for Information Interchange*

ASN.1 *Abstract Syntax Notation One*

ATR *Answer-to-Reset*

BER *Basic Encoding Rules*

BOM *Bill of Materials*

CA *Certificate Authority*

CBC *Cipher Block Chaining*

CDF *Certificate Directory File*

CI Circuito Integrado

CMAC *Cipher-based Message Authentication Code*

CPU *Central Processing Unit*

CRC *Cyclic Redundancy Check*

CRT *Control Reference Template*

DDoS *Distributed Denial-of-service*

DES *Data Encryption Standard*

DF *Dedicated File*

DH *Diffie-Hellman*

DODF *Data Object Directory File*

DSA *Digital Signature Algorithm*

DTLS *Datagram Transport Layer Security*

ECAD *Electronics Computer-aided Design*

ECB *Electronic Codebook*

ECDSA *Elliptic Curve Digital Signature Algorithm*

EEPROM *Electrically Erasable Programmable Read-only Memory*

EF *Elementary File*

ETU *Elementary Time Unit*

FCI *File Control Information*

FIPS *Federal Information Processing Standard*

GPIO *General-purpose Input/Output*

HMAC *Hash-based Message Authentication Code*

IDE *Integrated Development Environment*

IETF *Internet Engineering Task Force*

IoT *Internet of Things*

IP *Internet Protocol*

ISO *International Organization for Standardization*

ISoc *Internet Society*

ITU *International Telecommunication Union*

ITU-T *ITU Telecommunication Standardization Sector*

JC *Java Card*

JCDK *Java Card Development Kit*

JCRE *Java Card Runtime Environment*

JCVM *Java Card Virtual Machine*

MAC *Message Authentication Code*

MCU *Microcontroller Unit*

MF *Master File*

MQTT *Message Queue Telemetry Transport*

MUSCLE *Movement for the Use of Smart Cards in a Linux Environment*

NDEF *NFC Data Exchange Format*

NFC *Near-field Communication*

NIST *National Institute of Standards and Technology*

ODF *Object Directory File*

OS *Operating System*

OTA *Over-the-air*

PCI Placa de Circuito Impresso

PGP *Pretty Good Privacy*

PIC *Peripheral Interface Controller*

PIN *Personal Identification Number*

PKCS *Public Key Cryptography Standard*

PKI *Public Key Infrastructure*

PRF *Pseudorandom Function*

PrKDF *Private Key Directory File*

PUF *Physical Unclonable Function*

PuKDF *Public Key Directory File*

RAM *Random-access Memory*

RF Rádiofrequência

RFC *Request for Comments*

RIPEMD *RACE Integrity Primitives Evaluation Message Digest*

RISC *Reduced Instruction Set Computer*

ROM *Read-only Memory*

RSA *Rivest Shamir Adleman*

SC *Smart Card*

SCP *Secure Channel Protocol*

SHA *Secure Hash Algorithm*

SIM *Subscriber Identity Module*

SKDF *Secret Key Directory File*

SMD *Surface-mount Device*

SMT *Surface-mount Technology*

SoC *System-on-Chip*

SP *Special Publication*

SPI *Serial Peripheral Interface*

TCP *Transport Control Protocol*

TLS *Transport Layer Security*

TLV *Tag-length-value*

USB *Universal Serial Bus*

WDT *Watchdog Timer*

SUMÁRIO

1

INTRODUÇÃO

Internet das Coisas (também conhecida por sua sigla em inglês, IoT) é um termo cunhado em 1999 nas instalações do *Massachusetts Institute of Technology* (MIT) e refere-se à fusão entre objetos, sejam eles aparelhos eletrônicos ou não, e sistemas computacionais como processadores, sensores e, principalmente, módulos de comunicação *wireless*. Tais objetos, anteriormente passivos, estão agora conectados à *Internet* e são capazes de, entre outras coisas, trocar informações entre si e com outros objetos, coletar dados relativos à sua função, processá-los e tomar decisões de forma autônoma. Além disso, também podem ser monitorados remotamente através da Internet.

Existe um impacto massivo associado à IoT atualmente. Infográficos elaborados pelas empresas Intel (2015) e SAS (2016) mostram o quão grande é a rede da IoT, com bilhões de dispositivos conectados, e prevêem um crescimento de duas a dez vezes nesse número até o ano de 2020. Um relatório da Microsoft (EDSON, 2015) lista alguns dos fatores que aceleraram a adoção da IoT: queda dos custos de componentes de *hardware*, como *chips* e sensores, além de avanços nas suas arquiteturas; valor de aplicações industriais muito maior que o valor de aplicações para consumidores; progressos na área de *software* para análise de dados; aumento da conectividade de dispositivos com a introdução das redes de celular; vantagens dos serviços na nuvem (essenciais para a IoT), como baixo custo, escalabilidade e flexibilidade; enorme potencial econômico.

Uma extensa análise de mercado realizada pelo McKinsey Global Institute (MANYIKA *et al.*, 2015) estima que o impacto econômico da IoT pode ser de US$3,9 a US$11,1 trilhões por ano em 2025. Segundo o texto, as aplicações com maior valor em potencial estão nas categorias de Humanos (e.g., monitoramento de doenças, melhoria da qualidade de vida), Fábricas (e.g., otimização de operações, manutenção preventiva) e Cidades (e.g., segurança e saúde pública, controle de tráfego). No entanto, o alcance do máximo potencial econômico depende do desenvolvimento de fatores-chave como tecnologia, interoperabilidade, privacidade e confidencialidade, segurança, propriedade intelectual, organizações e políticas públicas.

Uma segunda análise produzida pela Gartner (VELOSA; SCHULTE; LHEUREUX, 2015) identifica tecnologias relevantes utilizando como ferramentas o *Ciclo de Expectativas* (tradução livre do termo em inglês, *Hype Cycle*) e a *Matriz de Prioridades*, relacionando maturidade,

expectativas de mercado, tempo para estabelecimento e benefícios. A própria IoT se encontra em um lugar de altas expectativas, com um benefício capaz de revolucionar a indústria e uma previsão de estabelecimento de cinco a dez anos. Tecnologias relacionadas a segurança da IoT, como *Autenticação da IoT*, *Segurança Digital* e *Segurança de Sistemas e Softwares Embarcados* são muito recentes e possuem um grande potencial para inovação, com benefícios significativos para o mercado.

Preocupações com segurança e privacidade são, sem dúvida, alguns dos maiores obstáculos relacionados ao desenvolvimento da IoT. A transformação de objetos comuns em pequenos computadores conectados à Internet significa que esses objetos podem ser hackeados e suas informações acessadas por pessoas não autorizadas. Existem relatos sobre falhas de segurança em câmeras (HILL, 2014)(KREBS, 2016), preocupações com *wearables*, aparelhos médicos, carros e até mesmo utilização de aparelhos na IoT em ataques DDoS (TAYLOR, 2016)(GALLAGHER, 2015). É possível, inclusive, utilizar um mecanismo de busca especializado para dispositivos conectados à Internet das Coisas (*Shodan*) para encontrar e observar a gravação de câmeras desprotegidas em todo o mundo (PORUP, 2016). Wisniewski (2016) enumera uma série de medidas de segurança para dispositivos IoT, algumas delas restritivas ao ponto de impedir o acesso à Internet ou serviços da nuvem. Schneier (2017), especialista em segurança e criptografia e autor dos livros *Applied Cryptography* e *Cryptography Engineering*, lista cinco "obviedades" sobre a segurança da IoT:

1. Na Internet o ataque é mais fácil que a defesa, dada a grande complexidade dos sistemas, o que aumenta a chance de vulnerabilidades. Além disso, atacantes não precisam se preocupar com leis, moral ou ética e fazem uso de novas tecnologias de maneira mais ágil;
2. A maioria dos *softwares* no mercado é mal escrita, por ser mais rápido e barato de produzir. Isso aumenta a quantidade de erros (*bugs*), os quais podem representar uma vulnerabilidade no sistema;
3. A conexão entre todas as coisas pode expôr novas vulnerabilidades. Dois sistemas seguros, se conectados de forma insegura, podem ser atacados;
4. Na Internet, todos os usuários estão ao alcance de todos os atacantes, dos piores aos melhores;
5. Existem leis que impedem o progresso da pesquisa na área de segurança, por motivos de quebra de *copyright*.

Tendo como principais motivações a carência de segurança em dispositivos IoT, bem como a ausência de dispositivos de baixo custo com segurança integrada e o alto potencial para inovação nessa área, este trabalho propõe arquiteturas de *hardware* e *software*, através da integração entre um microcontrolador, também conhecido como *Microcontroller Unit* (MCU), e um *Smart Card* (SC), *Elemento Seguro* capaz de armazenar informações e executar algoritmos de criptografia. Adicionalmente, dado o potencial econômico da IoT, este trabalho é focado em

sistemas de baixo custo, acessíveis à maioria dos usuários e muito utilizados na prototipação de projetos, baseados em plataformas conhecidas como *Arduino* e *ESP8266*. Suas principais contribuições estão ligadas à conexão entre o *Microcontroller Unit* (MCU) e o *Smart Card* (SC), entre elas: desenvolvimento e fabricação de circuitos que realizam a conexão física entre os diversos componentes; elaboração de bibliotecas que habilitam o MCU para requisição de operações criptográficas, instalação e desinstalação de programas e manipulação de dados armazenados no cartão; expansão de um *software open source* para SCs com a implementação de novas funções criptográficas e funções relativas ao protocolo de comunicação TLS.

1.1 Objetivos

O objetivo principal deste trabalho é a criação de uma arquitetura para IoT, englobando *hardware* e *software*, composta por dispositivos contendo um microcontrolador de baixo custo conectado à Internet das Coisas e integrado com um *Smart Card*, os quais se comunicam de forma criptografada e autenticada com um servidor, a partir da utilização das capacidades criptográficas deste cartão. Estes dispositivos também são capazes de se conectar com sensores e enviar seus dados ao servidor. A realização deste objetivo principal envolve a realização de objetivos específicos. Como será visto no Capítulo 4, a descrição do sistema está dividida em plataformas de *hardware* e *software*, e o mesmo pode ser feito com esses objetivos. Com relação ao desenvolvimento de *hardware*, os objetivos específicos são:

- Estabelecer uma conexão física entre um *Smart Card* e um microcontrolador comum e realizar a troca de mensagens entre eles com sucesso;
- Projetar e prototipar uma placa com um *Smart Card* e um módulo de comunicação sem fio que possa ser acoplada a um microcontrolador, ou que contenha um microcontrolador, e seja adequada para aplicações em IoT (a partir de critérios como tamanho, alimentação, conexão com sensores, etc.);

No que diz respeito ao desenvolvimento de *software*, são definidos os seguintes objetivos específicos:

- Elaborar um *software* para o microcontrolador capaz de executar um protocolo de comunicação segura baseado em requisições de operações criptográficas feitas ao *Smart Card*;
- Desenvolver um *software* para o *Smart Card* capaz de utilizar suas habilidades criptográficas e armazenar informações no próprio cartão, em conformidade com os padrões internacionais vigentes;
- Criar um *software* que agirá como o servidor do dispositivo IoT, comunicando-se de forma segura com ele e recebendo suas mensagens relativas à uma determinada aplicação;

- Avaliar o sistema sob diferentes métricas, como consumo de energia, custos de fabricação, performance, entre outras, para verificar sua compatibilidade com aplicações em IoT.

1.2 Estrutura do Trabalho

O trabalho está organizado em seis capítulos, incluindo este. O Capítulo 2 apresenta conceitos fundamentais para o entendimento do trabalho, entre eles: segurança de sistemas e criptografia, placas de desenvolvimento, *Smart Cards* e padrões internacionais relacionados. O Capítulo 3 referencia trabalhos que estão relacionados ao tema da dissertação, como propostas de integração entre *Smart Cards* e microcontroladores e métodos de segurança para IoT. O Capítulo 4 apresenta a proposta principal do trabalho e descreve todas as etapas do desenvolvimento de *hardware* e *software* do dispositivo IoT. O Capítulo 5 mostra avaliações feitas após a finalização do trabalho, com estudos do custo total e do consumo de energia do *hardware* fabricado. Por fim, o Capítulo 6 apresenta as conclusões sobre o trabalho, dificuldades encontradas ao longo de sua elaboração e atividades futuras que podem ser realizadas para melhoria do projeto.

2

FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta conceitos considerados fundamentais para o entendimento do trabalho realizado. Não é o objetivo deste capítulo, no entanto, se aprofundar excessivamente em todos os temas escolhidos, mas sim explicar de maneira sucinta (e, em alguns momentos, simplificada) os pontos importantes que constituem uma base teórica sobre a qual se apóia o projeto desenvolvido.

Muitos desses conceitos estão definidos em especificações e padrões internacionais de criptografia e segurança. Stallings (2011) identifica as organizações mais importantes nesse aspecto:

- *National Institute of Standards and Technology* (NIST): Agência federal dos Estados Unidos, responsável pela publicação dos *Federal Information Processing Standards* (FIPSs) e das *Special Publications* (SPs). Apesar de ser uma organização de atuação em escala nacional, seus padrões tem impacto mundial. Publicou os algoritmos *Secure Hash Algorithm* (SHA) e *Advanced Encryption Standard* (AES);
- *Internet Society* (ISoc): Organização americana sem fins lucrativos, com mais de 140 membros organizacionais e 80.000 membros individuais, cujo objetivo é liderar o desenvolvimendo de padrões relacionados à Internet. É a organização-mãe da *Internet Engineering Task Force* (IETF), organização que publica especificações voluntárias conhecidas como *Requests for Comments* (RFCs). Um desses RFCs especifica o protocolo TLS, discutido na Seção 2.4;
- *ITU Telecommunication Standardization Sector* (ITU-T): Setor da *International Telecommunication Union* (ITU), agência internacional das Nações Unidas, responsável por produzir padrões relacionados à área de telecomunicações. Seus padrões são chamados de Recomendações;
- *International Organization for Standardization* (ISO): Órgão internacional, independente e não-governamental composto por representantes de organizações de padronização mundiais em mais de 160 países (entre elas, ISoc e ITU), os quais desenvolvem, em conjunto, os padrões ISO.

Vale a pena ressaltar também a RSA Laboratories, responsável pela criação dos padrões

Public Key Cryptography Standard (PKCS), discutidos na Seção 2.5. Esses padrões, apesar de serem de uma empresa privada, são importantes no campo da criptografia de chave pública.

2.1 Segurança de Sistemas

Um glossário com definições de termos associados à Segurança na Internet é apresentado no RFC 2828 (SHIREY, 2000). Entre elas, são dadas três definições para o termo *segurança*:

1. Providências tomadas para proteger um sistema;
2. Condição de um sistema que resulta do estabelecimento e manutenção de medidas tomadas para protegê-lo;
3. Condição de recursos de um sistema de estarem isentos de acessos não autorizados e de mudanças, destruições ou perdas não autorizadas ou acidentais.

A Recomendação X.800 (ITU-T, 1991) estabelece uma arquitetura de segurança de sistemas baseada na definição de dois tipos de elementos: *serviços de segurança* e *mecanismos de segurança*. Ambos estão definidos no RFC 2828:

- *Serviço de segurança*
    1. Serviço de processamento ou comunicação que é fornecido por um sistema para dar um tipo específico de proteção a recursos do sistema;
    2. Serviço, fornecido por umas das camadas de comunicação de sistemas, que garante a segurança adequada dos sistemas ou das transferências de informações;
- *Mecanismo de segurança*: Processo (ou dispositivo que incorpora tal processo) que pode ser usado em um sistema para implementar um serviço de segurança que é fornecido por tal sistema, ou para implementá-lo dentro do próprio sistema.

O serviço de segurança é, portanto, um conceito abstrato que se refere a um objetivo de segurança do sistema, enquanto que o mecanismo de segurança é o procedimento aplicado na prática que permite o fornecimento de determinado serviço.

2.1.1 Serviços de Segurança

São definidos os seguintes serviços de segurança na Recomendação X.800:

- *Controle de acesso*: Proteção contra o uso não autorizado de recursos do sistema;
- *Autenticação de Entidade*: Garantia de que a entidade com a qual está se comunicando é quem ela diz ser;
- *Autenticação de Mensagem*: Garantia de que o remetente da mensagem é autêntico;
- *Confidencialidade*: A informação é mantida secreta para todas as partes não autorizadas a acessá-la;

- *Integridade*: Garantia de que a informação não foi modificada durante seu trajeto;
- *Não-repúdio*: Proteção contra a negação de envio/recebimento de mensagens por uma entidade participante da comunicação.

Paar e Pelzl (2010) classificam os quatro últimos serviços listados acima como os mais importantes na maioria dos sistemas e citam alguns serviços adicionais:

- *Disponibilidade*: Garantia de que o sistema está disponível sempre que requisitado;
- *Auditoria*: Fornecimento de evidências sobre atividades de segurança relevantes do sistema, a partir da criação de registros de eventos;
- *Segurança Física*: Proteção contra adulterações físicas no sistema;
- *Anonimato*: Proteção contra descoberta e mau uso de identidades.

O padrão FIPS 199 (NIST, 2004) utiliza a definição de "segurança da informação" presente no Código de Leis dos Estados Unidos: "*Proteção da informação e de sistemas de informação de acesso, uso, divulgação, disrupção, modificação ou destruição não autorizados a fim de prover confidencialidade, integridade e disponibilidade*", onde o serviço de integridade abrange também autenticação de mensagens e não-repúdio.

É importante ressaltar que diferentes aplicações requerem diferentes serviços de segurança. Para este trabalho, foram considerados indispensáveis os serviços que garantissem a segurança do *transporte de dados* entre as partes, dado que dispositivos IoT enviam, em geral, dados coletados de sensores, os quais podem conter informações sensíveis sobre seus usuários.

2.1.2 Mecanismos de Segurança

A Recomendação X.800 define os seguintes mecanismos de segurança:

- *Cifragem*: Utilização de algoritmos matemáticos para transformação de uma informação que se deseja transmitir, também chamada de texto claro ou legível, em uma informação que só pode ser interpretada mediante a utilização de uma chave, também conhecida como texto cifrado ou ilegível. Implementa os serviços de confidencialidade, autenticação de entidade, autenticação de mensagem e integridade;
- *Assinatura Digital*: Informação anexada ao final de uma mensagem que permite ao recipiente verificar a origem dos dados. Implementa os serviços de autenticação de entidade, autenticação de mensagem e não-repúdio;
- *Mecanismos de Controle de Acesso*: Utilização de informações sobre uma entidade para reforçar os seus diretos de acessos (e.g., identidade e senha). Implementa o serviço de controle de acesso;
- *Mecanismos de Integridade de Dados*: Informação anexada ao final de uma mensagem que permite ao recipiente verificar que os dados não foram modificados. Implementa os serviços de autenticação de mensagem e não-repúdio;

- *Troca de Autenticação*: Maneira de verificar a identidade de uma entidade a partir da troca de informações. Implementa o serviço de autenticação de entidade;
- *Padding de Tráfego*: Adição de informações aleatórias ou sem significado a uma mensagem, com o objetivo de dificultar a análise de tráfego. Implementa o serviço de confidencialidade;
- *Controle de Roteamento*: escolha de rotas fisicamente seguras ou com um determinado nível de proteção para troca de informações. Implementa o serviço de confidencialidade, pois garante que a transmissão de dados apenas por rotas consideradas seguras;
- *Notarização*: Utilização de uma terceira entidade confiada pelas partes, para garantir as propriedades de uma troca de informações, como integridade, origem, destino e tempo. Implementa o serviço de não-repúdio.

Os mecanismos considerados mais importantes para o trabalho são descritos nas próximas seções. Descrições detalhadas desses mecanismos podem ser encontradas nos livros de Katz e Lindell (2014), Stallings (2011) e Paar e Pelzl (2010).

Cifragem. A cifragem de informações pode ser feita de duas maneiras: *reversível*, quando é possível aplicar uma operação de *decifragem* sobre o texto cifrado para obter o texto claro novamente; e *irreversível*, quando não é possível obter o texto claro a partir do texto cifrado.

A cifragem reversível de dados pode ser dividida em duas categorias, de acordo com o tipo de chave que é utilizado:

- *Simétrica* ou de *Chave Secreta*: quando uma única chave é utilizada tanto para cifrar quanto para decifrar o texto. Por este motivo, ela precisa ser distribuída de forma segura para todas as entidades autorizadas a acessar a informação, e também deve ser armazenada de maneira segura. Algoritmos que realizam esse tipo de cifragem incluem o *Advanced Encryption Standard* (AES) e o *Data Encryption Standard* (DES);
- *Assimétrica* ou de *Chave Pública*: quando é utilizado um par de chaves, onde uma delas é utilizada para cifrar o texto e outra para decifrar o texto. A chave usada para cifrar é de acesso público (chave pública) e a chave usada para decifrar o texto é armazenada de forma segura pelo usuário (chave privada). Dessa forma, qualquer entidade pode utilizar a chave pública para enviar uma mensagem confidencial ao proprietário da chave privada e só ele poderá decifrá-la. Essas chaves não precisam ser distribuídas, pois cada usuário pode gerá-las individualmente. Exemplos de algoritmos de cifragem com chave pública são o *Rivest Shamir Adleman* (RSA) e o *Diffie-Hellman* (DH);

A Figura 2.2 mostra um diagrama comparando os dois tipos de cifragem. Em geral, algoritmos de chave secreta oferecem um nível de segurança similar aos algoritmos de chave

pública, mas com chaves bem menores e execução mais rápida. Por este motivo, muitos sistemas utilizam uma abordagem híbrida, onde a cifragem de chave pública serve apenas para estabelecer uma chave secreta comum entre as entidades em uma comunicação e, a partir daí, toda troca de informação é cifrada com a chave secreta (o protocolo TLS é um exemplo disso). A principal finalidade da cifragem é fornecer confidencialidade dos dados. No entanto, algoritmos de cifragem com chave secreta também podem ser utilizados em mecanismos de integridade de dados. Alguns algoritmos de cifragem de chave pública, por sua vez, são usados na criação de assinaturas digitais e em esquemas de estabelecimento de uma chave secreta, como é o caso do protocolo *Diffie-Hellman*. Este algoritmo, ilustrado de maneira simplificada na Figura 2.1, é baseado no *problema do logaritmo discreto* e é utilizado exclusivamente para o estabelecimento de um valor secreto comum entre duas ou mais entidades, o qual geralmente é utilizado como uma chave simétrica (ou como ponto de partida para a obtenção de uma chave simétrica, como no Protocolo TLS).

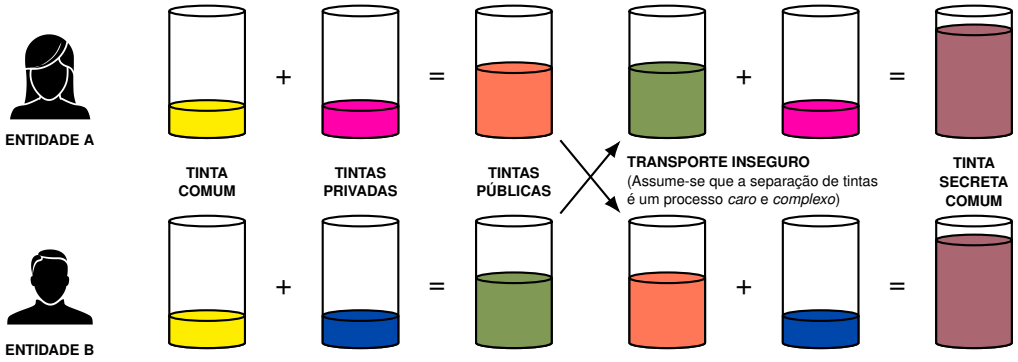


Figura 2.1: Protocolo *Diffie-Hellman* simplificado, onde as chaves são representadas por tintas e o problema do logaritmo discreto é representado pelo processo de separação das tintas. *Fonte das Entidades A e B: Noun Project, Dirk Rowe*

A cifragem irreversível de dados é representada pelas funções de *hash*. Essas funções recebem uma quantidade arbitrária de dados e produzem uma quantidade fixa e pequena de dados na saída. Suas principais característas são a dificuldade em se encontrar *colisões* (duas mensagens que resultam em um mesmo *hash*) e a impossibilidade de se obter a mensagem original a partir do seu *hash*. Por este motivo, essas funções são utilizadas na criação de assinaturas digitais e no controle de integridade de dados, com o objetivo de criar dados comprimidos e únicos baseados em outros dados maiores. Exemplos de algoritmos de *hash* incluem o SHA e o SHA256.

Assinatura Digital. Esquemas de assinatura digital utilizam algoritmos de criptografia baseados em chaves assimétricas em conjunto com algoritmos de *hash* para calcular, a partir de uma mensagem, uma segunda informação atrelada ao remetente desta mensagem, sua assinatura. Da mesma forma que assinaturas escritas em um papel, as assinaturas digitais são utilizadas para identificar a origem de uma determinada informação (autenticação de entidade), e também impedem o remetente de negar o seu envio (não-repúdio). Adicionalmente, as assinaturas digitais

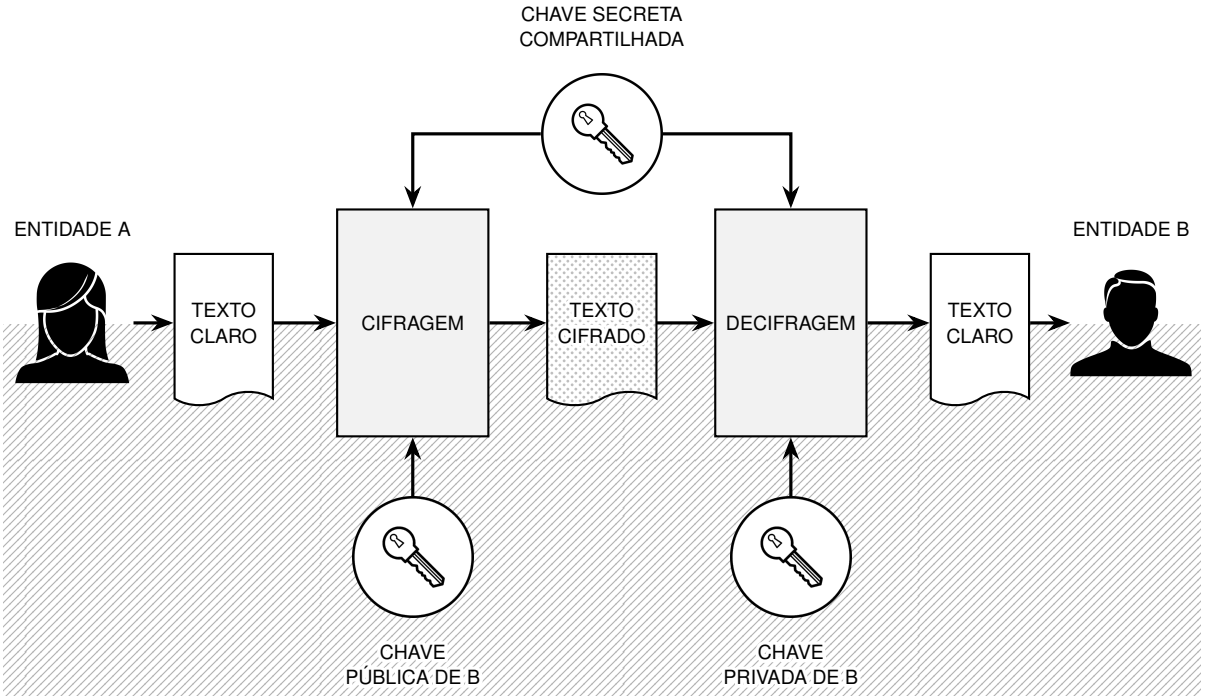


Figura 2.2: Comparação entre cifragem simétrica (acima) e assimétrica (abaixo). *Fontes: Entidades A e B: Noun Project, Dirk Rowe; Chave: Noun Project, Jemis Mali*

garantem a integridade da informação, pois qualquer alteração nos dados modifica drasticamente o conteúdo da assinatura.

Nos algoritmos de assinatura digital, a chave privada é utilizada pelo remetente para gerar a assinatura a partir da informação, visto que só ele tem acesso a essa chave. Qualquer outra entidade que queira verificar esta assinatura utiliza a chave pública do remetente. É possível gerar assinaturas digitais com algoritmos de cifragem, como o RSA, mas também existem algoritmos específicos para criação de assinaturas digitais, como o *Digital Signature Algorithm* (DSA).

Mecanismos de Integridade de Dados. O principal mecanismo utilizado para garantir integridade e autenticação de mensagens é o *Message Authentication Code* (MAC). Os MACs são análogos às assinaturas digitais, mas utilizam algoritmos com chave secreta. Nesse caso, uma única chave compartilhada entre as partes serve tanto para calcular o MAC quanto para verificá-lo. Por ser um código único associado a uma mensagem, o MAC implementa a autenticação e integridade da mensagem, dado que qualquer alteração na mensagem altera o MAC e só quem possui a chave secreta é capaz de calculá-lo. No entanto, não implementa a não-repudiação pois não é possível provar qual das partes em uma comunicação calculou um determinado MAC. Existem MACs baseados em algoritimos de *hash*, os HMACs, e em algoritimos de cifragem, os CMACs. Os primeiros são utilizados no TLS e os segundos são utilizados na especificação GlobalPlatform (GlobalPlatform, 2003) e, portanto, ambos são utilizados neste trabalho.

Os algoritmos de MAC, assim como os algoritmos de cifragem com chave secreta, possuem uma execução muito mais rápida e chaves menores do que os algoritmos de assinatura

digital, que utilizam chaves assimétricas. Por esse motivo, uma vez que as assinaturas digitais são utilizadas em um esquema para estabelecer chaves secretas (como o TLS), as mensagens subsequentes são "assinadas" com MACs. A Figura 2.3 mostra um diagrama comparando MACs e assinaturas digitais.

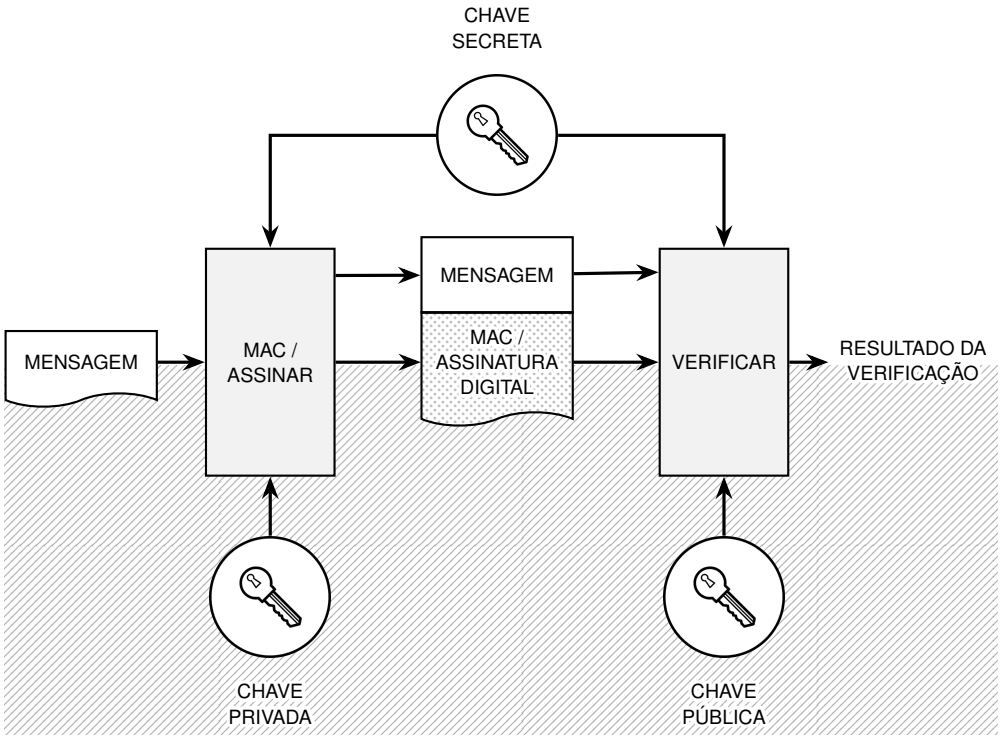


Figura 2.3: Comparação entre MAC (acima) e assinatura digital (abaixo). *Fonte: Chave: Noun Project, Jemis Mali*

Troca de Autenticação. Uma maneira de realizar a troca de autenticação é através de protocolos de "aperto de mão", onde as entidades trocam informações que podem ser utilizadas para verificar sua autenticidade mutualmente. No TLS, por exemplo, as entidades trocam *certificados*: documentos que associam uma identidade (nome, país, estado, endereço, etc.) a uma chave pública e são assinados digitalmente por uma terceira entidade, na qual as primeiras confiam. Na Internet, um certificado é assinado por uma determinada entidade, e o certificado dessa entidade é assinado por outra, e assim sucessivamente formando uma cadeira de certificados, até se chegar na entidade raíz, a qual não precisa ser certificada por ninguém e na qual todos confiam. Essas entidades responsáveis pela emissão de certificados são chamadas de *Certificate Authority* (CA) ou Autoridades de Certificação. No sistema proposto neste trabalho, o servidor atua como uma CA e emite certificados para os clientes, os quais se autenticam com o próprio servidor posteriormente, em um protocolo de aperto de mão.

2.2 Ambientes de Desenvolvimento de Baixo Custo

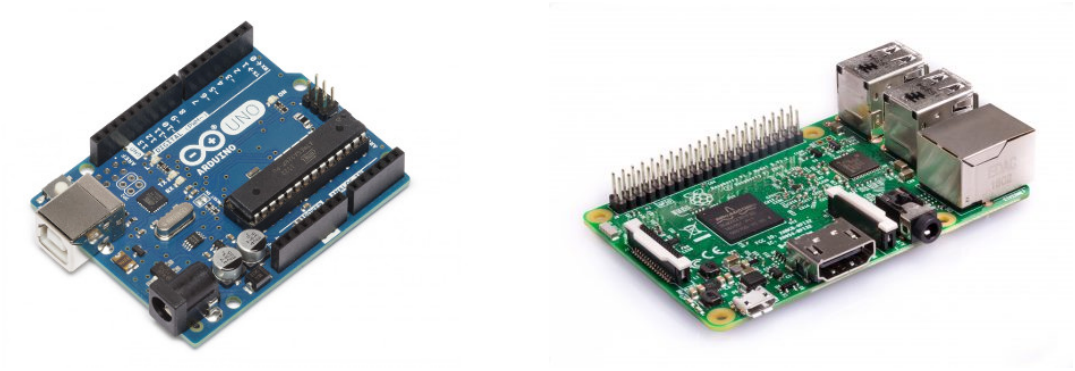
A prototipação de aplicações em *Internet of Things* (IoT) é baseada na utilização de plataformas de *hardware* economicamente acessíveis que fornecem uma maneira simples e rápida de programar um microcontrolador. Tais plataformas são chamadas *Ambientes de Desenvolvimento de Baixo Custo*. Em geral, são circuitos eletrônicos que dão ao usuário acesso a várias interfaces de entrada/saída do MCU, como portas digitais e analógicas e interfaces de comunicação, e que podem ser conectados a um computador e programados de maneira fácil. Essas interfaces podem, por sua vez, ser conectadas com os mais diversos tipos de sensores e módulos de comunicação sem fio para criação de projetos em IoT. Existem desde circuitos mais simples, com processadores de dezenas de MHz e memórias na ordem de kB (e.g., Arduino UNO, MSP430 Launchpad), a circuitos que são verdadeiros computadores, com processamento e memória comparáveis aos de um *smartphone*, capazes de rodar sistemas operacionais como Linux e Android (e.g., Raspberry Pi, BeagleBone). A aplicação que se deseja desenvolver determina qual ambiente deve ser escolhido.

O relatório da Gartner sobre tecnologias em IoT (VELOSA; SCHULTE; LHEUREUX, 2015) cita os ambientes de desenvolvimento de baixo custo como uma das tecnologias emergentes de alto benefício para o mercado, principalmente por fomentar a inovação em IoT realizada por desenvolvedores individuais. Ele destaca ainda algumas vantagens, como:

- Versatilidade, pois estes circuitos podem ser utilizadas por uma vasta gama de usuários que vai desde de pessoas com pouco conhecimento de eletrônica e/ou computação elaborando projetos básicos (inclusive com propósitos educacionais) a profissionais e startups desenvolvendo projetos complexos e, por vezes, comerciais, como *drones* e impressoras 3D;
- Flexibilidade, dando ao desenvolvedor a possibilidade de alteração do *design* original para se encaixar nos requisitos de seus projetos.

Recursos relacionados a elaboração de projetos com esses ambientes, como bibliotecas, exemplos e tutoriais, podem ser encontrados nas diversas comunidades *online* de desenvolvedores de *hardware*. Uma *survey* realizada pelo site Hackster (2016) destaca algumas comunidades mais acessadas pelos usuários, entre elas: Arduino, Instructables, Adafruit e SparkFun. Essa mesma pesquisa também aponta as placas Arduino e Raspberry Pi (Figura 2.4) como sendo as mais utilizadas por desenvolvedores, para prototipação de sistemas.

Este trabalho teve como foco os ambientes de desenvolvimento mais simples. Em específico, foi utilizada a placa *Arduino UNO*, no início do projeto, e os módulos do *chip ESP8266*, do meio ao final. Os detalhes do desenvolvimento de *hardware* são especificados na Seção 4.1.



(a) **(b)**

Figura 2.4: Arduino UNO (a) e Raspberry Pi 3 Modelo B (b). *Fontes: store.arduino.cc e raspberrypi.org*

2.3 Smart Cards

Os *Smart Cards* (SCs) são cartões de plástico nos quais estão embutidos Circuitos Integrados (CIs), usualmente chamados de *chips*. Esses cartões podem ser classificados de acordo com três características principais:

- Tamanho;
- Tipo de *chip*;
- Método de transmissão de dados.

Nos padrões ISO que especificam as características dos SCs, eles são categorizados como *Cartões de Identificação – Cartões com Circuitos Integrados*. O padrão ISO7810 (ISO/IEC, 2003), em particular, especifica as características físicas dos cartões de identificação, entre elas as suas dimensões. São definidos quatro tamanhos diferentes para os cartões, sendo os mais relevantes o ID-1 e o ID-000, mostrados na Figura 2.5. O primeiro é o tamanho utilizado em cartões de crédito e o segundo é o tamanho utilizado em cartões SIM para celular. Inicialmente, foram utilizados cartões ID-1 no desenvolvimento deste trabalho, os quais foram transformados em cartões ID-000, para favorecer a portabilidade do sistema.

Quanto ao tipo de *chip* contido, Rankl e Effing (2010) definem dois tipos de *Smart Card*:

- *Cartões de Memória*: Possuem apenas uma memória não-volátil (i.e., que mantém os dados armazenados, mesmo após ser desligada e ligada novamente), tipicamente uma EEPROM, e podem conter uma lógica de segurança associada ao acesso dessa memória. Em geral, são otimizados para aplicações mais simples e específicas e, por isso, são mais baratos;
- *Cartões de Processador*: Possuem um processador (ou CPU), uma memória ROM (apenas leitura) com o seu sistema operacional, uma memória EEPROM onde podem ser armazenados dados e códigos de aplicações que são executadas pelo cartão e uma

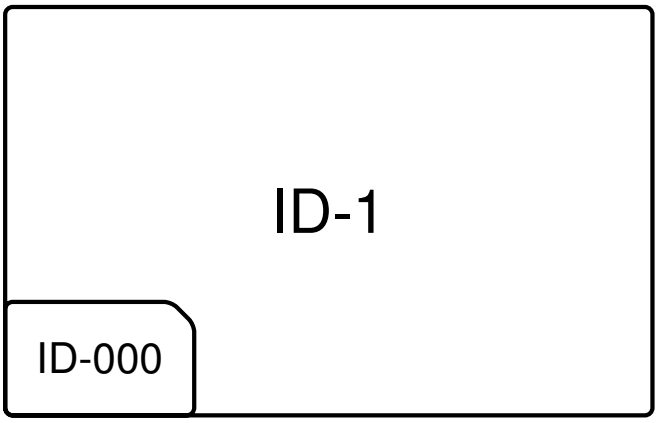


Figura 2.5: Comparação entre os tamanhos de cartão ID-1 e ID-000

memória RAM (volátil, apaga quando é desligada) para armazenar os dados durante a execução de uma aplicação. Em geral, cartões de processador possuem também um segundo processador (também chamado de *criptoprocessador*), cuja função é acelerar a execução de algoritmos de criptografia. São cartões mais versáteis, pois podem conter várias aplicações. Foi utilizado um cartão desse tipo no desenvolvimento do trabalho.

Por fim, a transmissão de dados pode ser feita por contatos elétricos ou sem contatos, por meio de uma interface de Rádiofrequência, de forma que o cartão não precisa ser inserido em um leitor, apenas aproximado a uma distância ao alcance da antena. Os cartões podem ser somente de contato, somente sem contato, de interface dupla (um único *chip* que suporta os dois métodos de transmissão) ou híbridos (um cartão com dois *chips* distintos, cada um suportando um tipo de transmissão). Como o objetivo do trabalho era conectar um SC a um MCU, foi utilizado um cartão com interface de contatos, como o da Figura 2.6.

Figura 2.6: Cartão ID-1 com interface de contatos elétricos. *Fonte: ic0nstrux.com*

2.3.1 Segurança dos Smart Cards

Os *Smart Cards* geralmente são referidos como sistemas invioláveis ou resistentes à adulteração, capazes de armazenar dados de maneira segura. Rankl e Effing (2010) dedicam um capítulo inteiro do seu livro à descrição das diferentes estratégias de proteção que são implementadas em um SC. Os autores atribuem a segurança do cartão a quatro componentes principais:

- Corpo;
- *Hardware* do CI;
- Sistema Operacional;
- Aplicação.

A segurança do corpo do cartão é relativa a características que podem ser checadas visualmente por pessoas, como marcações, impressões e coisas semelhantes. Em aplicações como a deste trabalho, em que o cartão não passa por nenhuma verificação humana, apenas os três últimos parâmetros são importantes. São definidas no texto algumas categorias de ataques que podem ser realizados em *Smart Cards*:

- *Ataques Sociais*: Têm como alvo as pessoas que utilizam os cartões, e não os próprios cartões em si;
- *Ataques Físicos*: Direcionados ao *hardware* do cartão, geralmente necessitam de uma quantidade considerável de recursos técnicos. Podem ser classificados como *estáticos*, quando o cartão não precisa estar ligado, ou *dinâmicos*, quando são feitas observações sobre o estado físico do cartão durante o seu funcionamento;
- *Ataques Lógicos*: Baseados em fraquezas do *software* do cartão, seu sistema operacional e suas aplicações, e também em criptanálise tradicional (análise de algoritmos de criptografia em busca de falhas). Podem ser classificados como *passivos*, quando apenas são observadas trocas de mensagens e feitas medidas no CI, ou *ativos*, quando existe a manipulação da troca de dados e do dispositivo.

São tomadas medidas de proteção em todas as fases da vida de um cartão: no seu desenvolvimento, produção e utilização, onde de fato se concentra a maior parte dos ataques. Em específico, a realização de ataques físicos no CI do cartão, segundo Rankl e Effing (2010), requer diversos equipamentos especializados, tais como: microscópio, cortador a laser, micromanipuladores, feixes de íon focalizados, equipamentos de fresagem química e computadores de alta velocidade. Antes de qualquer ataque, o *chip* precisa ser removido do cartão. O mesmo é protegido por uma camada de resina, a qual também deve ser removida sem danificar o circuito. Com isso, o semicondutor está exposto e pode ser manipulado. No entanto, existe uma série de medidas de segurança implementadas no *hardware* que precisam ser transpassadas, entre elas:

- O tamanho das estruturas dentro do *chip*, da ordem de centenas de nm, dificulta a análise do circuito;
- O *design* dos CIs é exclusivo para *Smart Cards* e não é utilizado para outros tipos de dispositivos, onde segurança não é um fator crítico;
- Presença de estruturas falsas no *chip*, sem função alguma, apenas para dificultar a localização das estruturas reais;
- Os barramentos que conectam a CPU às memórias são posicionados em camadas de difícil acesso dentro do semicondutor e, em alguns casos, são embaralhados para mascarar sua função;
- Utilização de "escudos" sobre a superfície do *chip*, que impedem a medição de tensões em regiões específicas. Alguns desses escudos são usados para alimentar o cartão e acabam por inutilizá-lo se forem removidos, outros são feitos especificamente para detecção de ataques físicos;
- Módulos de monitoração de tensão dentro do *chip*, que impedem que o cartão funcione fora da sua região de operação, onde podem ocorrer falhas que permitam o vazamento de dados.

Rankl e Effing (2010) também detalham mecanismos de proteção adicionais associados a ataques ao *software* do cartão, como a utilização de controle de acesso para arquivos (presente na PKCS#15), autenticação entre cartão e entidade externa, troca de mensagens seguras (ambos presentes na especificação GlobalPlatform), entre outros. Em suma, *Smart Cards* não são perfeitamente seguros, mas são desenvolvidos de forma que um ataque direto ao seu sistema seja uma tarefa árdua e custosa, mesmo para adversários especializados.

2.3.2 Padrão ISO7816

A especificação internacional de todos os parâmetros relacionados aos *Smart Cards* é realizada pelo padrão ISO7816. Ele consiste em quatorze partes diferentes (1 a 13 e 15), onde cinco delas (1 a 3, 10 e 12) são específicas para cartões com contatos elétricos e as restantes independem do método de transmissão de dados. Apenas algumas partes do padrão foram utilizadas como referência para este trabalho: 1 a 4, 8 e 9, pois são as partes que tratam das características físicas, dos protocolos de comunicação e dos comandos que o *Smart Card* pode receber.

As partes 1 e 2 tratam de características físicas dos cartões com contatos. A primeira delas (ISO/IEC, 1998) é mais focada no cartão em si, estendendo o conteúdo da ISO7810 citada anteriormente com propriedades adicionais que levam em consideração a presença dos contatos, como proteção à eletricidade estática e interferência eletromagnética, temperatura de operação, entre outras. A segunda parte (ISO/IEC, 2007) trata especificamente dos contatos elétricos, define uma quantidade de oito contatos (C1 a C8), suas dimensões e localização em um cartão de tamanho ID-1.

A terceira parte do padrão ISO7816 (ISO/IEC, 2006) define as características elétricas e os protocolos de transmissão de dados dos *Smart Card*s (SCs) com contatos. Incialmente, são especificadas as classes de operação de acordo com a tensão que deve ser aplicada ao cartão para que ele funcione:

- *Classe A*: Tensão de operação igual a 5 V;
- *Classe B*: Tensão de operação igual a 3 V;
- *Classe C*: Tensão de operação igual a 1,8 V.

Depois, são especificados os contatos elétricos, seus nomes e funções:

- VCC (C1): Usado para alimentar o cartão com a tensão de operação;
- RST (C2): Usado para fornecer o sinal de *reset* para o cartão, que o faz retornar ao estado inicial de operação;
- CLK (C3): Usado para fornecer o sinal de relógio (ou de *clock*) para o cartão. Este sinal coordena todas as ações executadas pelo cartão e tem como valores mínimo e máximo 1 MHz e 5 MHz, respectivamente;
- GND (C5): Tensão de referência para o cartão;
- I/O (C7): Usado para envio e recebimento de dados, de forma serial.

A Figura 2.7 mostra uma representação dos contatos em um *Smart Card*, com seus respectivos nomes. Os contatos C4 e C8 não possuem funções definidas na Parte 3 da ISO7816 (NC = *Not Connected*, Não Conectado), mas podem ser usados como os terminais de dados de uma interface USB para o cartão, de acordo com a Parte 12 da ISO7816 (ISO/IEC, 2005a). O contato C6, por sua vez, é de aplicação geral (SPU = *Standard or Proprietary Use*, Uso Padronizado ou Proprietário), como entrada ou saída, mas não possui função definida na ISO7816 (é reservado para uso futuro). O cartão não possui interface USB, portanto, esses três contatos (C4, C8 e C6) nunca são utilizados no projeto. A ISO7816-3 especifica, então, quatro procedimentos que devem ser realizados durante a operação do cartão. Na ordem de execução, os procedimentos são:

1. *Ativação*: RST deve ser colocado em nível baixo, VCC deve ser ligado, o I/O do leitor deve ser colocado em modo de *recepção* e um sinal de relógio deve ser aplicado a CLK;
2. *Cold Reset*: após o sinal de relógio ser aplicado, RST deve permanecer em nível baixo por pelo menos 400 ciclos do relógio e então ser colocado em nível alto;
3. *Troca de informações*: após um tempo que varia entre 400 e 40000 ciclos de relógio, o SC envia o *Answer-to-Reset* (ATR) (resposta fixa, que contém alguns parâmetros de comunicação), negocia um protocolo de transmissão e seus parâmetros com o leitor e a transferência de mensagens entre as partes é iniciada;

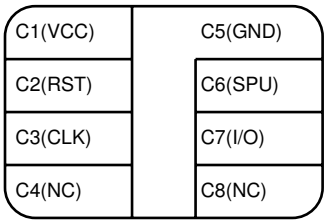


Figura 2.7: Contatos de um *Smart Card*

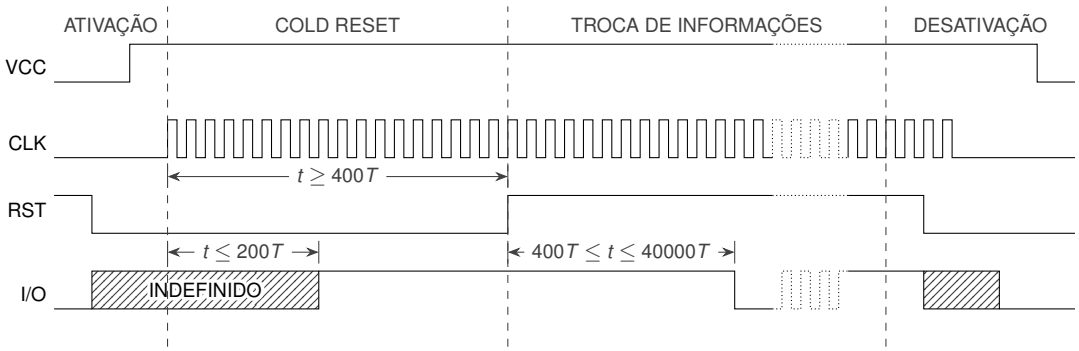


Figura 2.8: Procedimentos de operação do *Smart Card*

4. *Desativação*: RST, CLK e I/O devem ser colocados, nessa ordem, em nível baixo e VCC deve ser desligado.

A Figura 2.8 detalha o primeiro, segundo e quarto procedimentos. O terceiro procedimento, a troca de informações, é realizado através do envio e recebimento de caracteres. Cada caractere é formado por 10 *bits* e cada *bit* tem duração de 1 *Elementary Time Unit* (ETU), unidade básica de tempo na qual o protocolo de transmissão é baseado. O valor padrão do ETU é de 372 ciclos do relógio (e.g., se o relógio tem frequência de 2 MHz, o ETU é 372×0,5 ns = 186 ns) e pode ser reconfigurado após o recebimento do ATR. Antes do início de qualquer comunicação, o I/O deve estar em nível alto. O primeiro *bit* marca o início do caractere e possui valor sempre igual a "0". Os próximos 8 *bits* codificam um *byte* de dados que se deseja transmitir. O décimo e último *bit* é a paridade do caractere, utilizado como uma maneira simples para detecção de erros. A paridade está correta se existe uma quantidade par de valores "1" entre o segundo e décimo *bits* do caractere. Se a paridade estiver correta, existe uma pausa até o envio do próximo caractere e se estiver errada, o receptor deve enviar um sinal de erro e esperar o reenvio do mesmo caractere. A Figura 2.9 exemplifica o comportamento do I/O durante o envio de um caractere.

Uma cadeia de caracteres forma um comando. O protocolo mais simples de comunicação é o T=0, onde o microcontrolador funciona como um dispositivo mestre e envia comandos para o cartão, que os processa (um por vez) e envia a resposta. Os comandos e respostas são enviados em um ou mais "pacotes" chamados *Application Protocol Data Units* (APDUs), os quais são divididos em:

- *APDUs de Comando*: possuem um cabeçalho obrigatório de quatro bytes contendo a

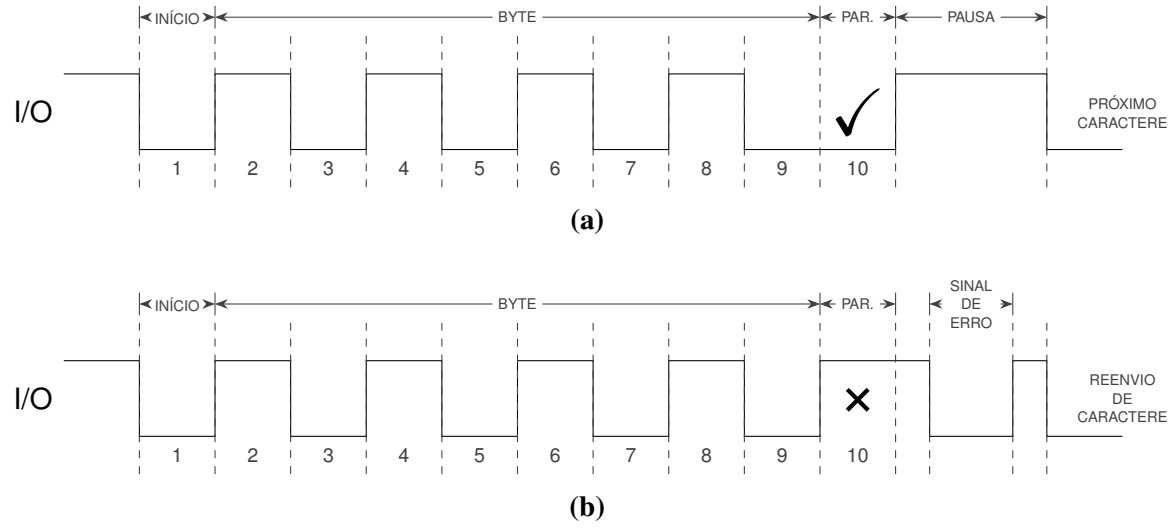


Figura 2.9: Envio de um caractere com paridade correta (a) e incorreta (b)

classe do comando (CLA), o tipo de instrução (INS) e dois parâmetros relacionados à instrução (P1 e P2), e um corpo condicional de tamanho variável que pode conter o número de bytes que estão sendo enviados no comando (L_c) juntamente com os bytes de dados e/ou o número de bytes esperados na resposta (L_e). Caso não seja necessário enviar nenhuma informação com o comando, nem seja esperada nenhuma informação na resposta, o corpo não existe;

- *APDUs de Resposta*: possuem um corpo condicional de tamanho variável que pode conter os bytes da resposta ao comando (caso sejam necessários) e um rodapé obrigatório de dois bytes contendo o status do processamento do comando (SW1 e SW2).

A Figura 2.10 ilustra o formato de cada tipo de APDU. A classe do comando determina se ele é interindustrial, padronizado de acordo com a ISO7816, ou proprietário, criado pelos desenvolvedores de uma aplicação específica. A instrução informa ao cartão que comando deve ser executado, sob as condições especificadas nos parâmetros P1 e P2 (quando necessários). Os bytes de status SW1 e SW2 informam ao leitor de cartão se o comando foi executado corretamente ou não. Os valores mais frequentes de SW1 e SW2 durante uma troca de mensagens entre leitor e SC estão listados na Tabela 2.1.

A parte 4 do padrão ISO7816 (ISO/IEC, 2005b) define os conceitos de organização, segurança e a maioria dos comandos aceitos pelo SC. A organização diz respeito a:

- Pares de comando/resposta (também definidos na parte 3) e os significados dos bytes CLA, INS, SW1 e SW2;
- Formato dos objetos armazenados no cartão, de acordo com as regras de codificação da ASN.1 (Seção 2.6);
- Estrutura dos dados do cartão, baseada em *Dedicated Files* (DFs) e *Elementary Files* (EFs), os quais também são utilizados na especificação PKCS#15 (Seção 2.5);

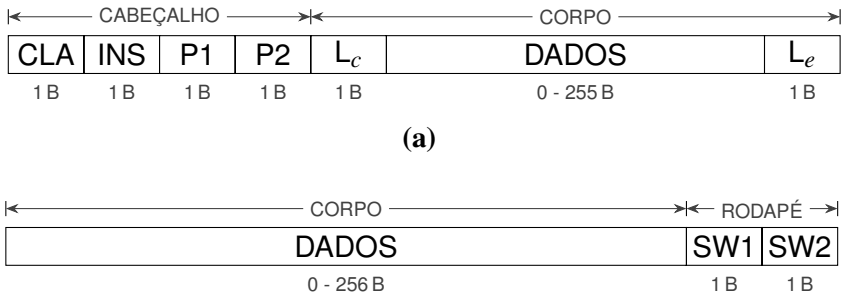


Figura 2.10: Estrutura das APDUs de comando (a) e resposta (b)

SW1/SW2	STATUS
0x9000	Processamento normal.
0x61XX	Processamento normal. O byte SW2 codifica a quantidade de bytes ainda disponíveis para serem recebidos pelo leitor (entre 1 e 256, onde 0x00 = 256), o qual deve enviar um comando de recebimento de dados.
0x6CXX	Erro de verificação. A quantidade de bytes esperados (L_e) enviada pelo leitor é incorreta. O mesmo comando deve ser enviado novamente com a quantidade certa, a qual está codificada em SW2.

Tabela 2.1: Valores comuns para os bytes SW1 e SW2

- Arquitetura de segurança do cartão, onde são definidas, entre outras informações: *status* de segurança para aplicações, arquivos e comandos; mecanismos de segurança utilizados pelo cartão para se autenticar e comunicar de maneira segura com uma entidade externa; condições de acesso para dados armazenados no cartão. A arquitetura de segurança é complementada com uma especificação de comunicação com *mensagens seguras*, construídas a partir dos mecanismos definidos previamente.

A maioria dos comandos definidos na parte 4 é relativa à manipulação de arquivos no *Smart Card*. Por exemplo, são definidos os comandos de seleção de arquivos (`SELECT`), leitura (`READ BINARY`) e atualização (`UPDATE BINARY`) de dados dentro dos arquivos. São definidos também alguns comandos de segurança básica, como os de autenticação com uma entidade externa (`EXTERNAL AUTHENTICATE`), geração de dados aleatórios para utilização em esquemas de autenticação (`GET CHALLENGE`) e verificação de uma informação interna do cartão, como um PIN (`VERIFY`). É definido ainda o comando utilizado para obtenção dos dados restantes de uma resposta (`GET RESPONSE`), necessário no segundo caso da Tabela 2.1. As definições dos comandos envolvem quais valores devem ser utilizados para CLA, INS, P1, P2, L_c, L_e e o conteúdo do corpo no APDU de comando, bem como o que é esperado no corpo do APDU de resposta e os possíveis valores para SW1 e SW2.

As partes 8 e 9 também definem comandos aceitos pelo cartão. A parte 8 (ISO/IEC, 2004a), especificamente, define comandos que devem ser usados para operações criptográficas, como geração de chaves assimétricas (`GENERATE ASYMMETRIC KEY PAIR`), cifragem, decifragem, assinatura digital, verificação de assinatura e *hash* (`PERFORM SECURITY OPERATION`, variando P1 e P2 de acordo com a operação desejada). A parte 9 (ISO/IEC, 2004b), por sua vez, define comandos relativos ao gerenciamento dos arquivos no cartão, como os comandos para criar (`CREATE FILE`) e remover (`DELETE FILE`) arquivos.

2.3.3 Java Card

A tecnologia Java Card, desenvolvida pela Sun Microsystems (atualmente uma parte da Oracle), permite que *Smart Cards* e outros dispositivos com restrições de processamento e memória executem programas escritos em um subconjunto da linguagem Java. Tais programas são chamados de *applets*. Essa tecnologia é baseada em três elementos principais: a *Java Card Virtual Machine* (JCVM), o *Java Card Runtime Environment* (JCRE) e a *Java Card Application Programming Interface* (API).

A máquina virtual (JCVM) e o ambiente de execução (JCRE) possuem todas as ferramentas necessárias para habilitar um processador com a capacidade de rodar *applets* Java Card. A API (Sun Microsystems, 2006a), por sua vez, consiste em um conjunto de elementos (e.g., rotinas, classes, métodos, constantes) utilizados na criação do código Java que será transformado em um *applet* para o cartão. O objetivo da API é simplificar o desenvolvimento de *software*, abstraindo os detalhes da implementação e fornecendo "blocos de montagem" com aquilo que é necessário para o desenvolvedor. A API Java Card possui diferentes pacotes com inúmeros recursos, entre eles:

- Métodos para controle de execução e gerenciamento de recursos de memória;
- Métodos para manipulação de *arrays*;
- Métodos para envio e recebimento de APDUs;
- Definições de constantes relacionadas ao padrão ISO7816;
- Definições de algoritmos de segurança e criptografia (e.g., cifragem/decifragem, geração de chaves, *hash*, geração de números aleatórios, assinatura digital e verificação, troca de chaves).

Uma vez que o código Java é finalizado, ele precisa ser transformado em um *applet* que possa ser executado pelo cartão. As ferramentas necessárias para realizar essa conversão fazem parte do *Java Card Development Kit* (JCDK). O kit de desenvolvimendo consiste em um pacote de *softwares* e documentos necessários para o desenvolvimento de aplicações Java Card, incluindo as especificações dos elementos citados no início dessa seção. O guia do usário do JCDK (Sun Microsystems, 2006b) descreve os passos necessários para criar um arquivo de *applet* que será instalado no cartão. Primeiramente, o código-fonte em Java é compilado com a ferramenta `javac`, o que resulta em um arquivo de Classe. Esse arquivo de Classe é, por sua

vez, convertido para um arquivo CAP (*Compiled Applet*) com a ferramenta `converter`, o qual pode ser instalado no cartão.

Uma *datasheet* da Oracle (2012) sobre a tecnologia Java Card lista alguns dos seus principais benefícios:

- *Interoperabilidade*: *Applets* desenvolvidos para Java Card são compatíveis com qualquer cartão que possua essa tecnologia, independentemente do fornecedor e do *hardware*;
- *Segurança*: Tecnologia desenvolvida com um processo aberto, que faz uso de implementações comprovadas pela indústria e avaliações de segurança de alto nível;
- *Capacidade de múltiplas aplicações*: Várias aplicações podem coexistir de forma segura em um cartão com Java Card;
- *Natureza dinâmica*: Novas aplicações podem ser instaladas no cartão mesmo após a sua emissão para o cliente;
- *Compatibilidade com padrões existentes*: A API Java Card é compatível com padrões como a ISO7816 e GlobalPlatform.

Em um infográfico publicado no site do Java Card Forum (2017), é estimada uma produção de mais de 3 bilhões de cartões com a tecnologia Java Card por ano desde 2015, nas mas diversas áreas de aplicação. É antecipada também uma nova versão do Java Card, com caracteristicas específicas para IoT.

2.3.4 Especificação GlobalPlatform

A GlobalPlatform é uma associação industrial sem fins lucrativos com mais de 100 empresas participantes, cujo objetivo é desenvolver e publicar especificações relacionadas com a tecnologia de *chips* seguros (e.g, *Smart Cards*). A sua especificação para cartões, *GlobalPlatform Card Specification* (GlobalPlatform, 2003), estabelece uma arquitetura para a criação de cartões capazes de conter e executar múltiplas aplicações.

O conteúdo do cartão é separado em diferentes contêineres, chamados de Arquivos de Carregamento Executáveis (*Executable Load Files*), os quais contém os códigos executáveis de cada aplicação, chamados de Módulos Executáveis (*Executable Modules*). Essas aplicações rodam em um ambiente de execução de escolha do desenvolvedor, e são criadas com a API associada à esse ambiente. Por exemplo, o cartão pode funcionar com aplicações que são executadas no *Java Card Runtime Environment* (JCRE), e desenvolvidas a partir da *Java Card* API (Subseção 2.3.3).

A administração das aplicações no cartão é feita pelo Domínio de Segurança (*Security Domain*), uma aplicação com privilégios de acesso superiores que atua como o representante do desenvolvedor da aplicação dentro do cartão. Através dele é possível instalar e desinstalar aplicações, com os comandos definidos na especificação. A maioria dos comandos definidos

na especificação GlobalPlatform é proprietária, ou seja, não está na ISO7816. Os principais comandos são aqueles utilizados para carregar os dados da aplicação no cartão (`LOAD`), instalá-la (`INSTALL`) e removê-la (`DELETE`). Também é definido um comando para a extração de informações sobre o cartão (`GET DATA`) e também sobre o Domínio de Segurança e as aplicações instaladas no cartão (`GET STATUS`).

O Domínio de Segurança também é responsável por criar um canal seguro de comunicação entre o cartão e uma entidade externa. Essa ação é realizada através do *Secure Channel Protocol* (SCP). Ele provê os seguintes serviços de segurança:

- Autenticação de Entidade: a entidade externa autentica o cartão e o cartão autentica a entidade externa;
- Integridade e Autenticação de Mensagens: As mensagens trocadas possuem um MAC;
- Confidencialidade: As mensagens enviadas para o cartão são cifradas.

Os serviços de segurança que serão utilizados no SCP são escolhidos pela entidade externa, mas o mínimo requerido na especificação é a autenticação entre entidades e a presença de MACs nas mensagens enviadas para o cartão. A autenticação das entidades é possível através do cálculo de dados baseados em uma *chave base* de 16 B contida no cartão, a qual só é conhecida pelo desenvolvedor da aplicação. Em geral, essa chave tem um valor padrão 0x40414243444546474849, o qual pode ser alterado. O resultado dessa troca de autenticação é a geração de chaves simétricas usadas na criação e verificação dos MACs e na cifragem e decifragem das mensagens. Dessa forma, apenas alguém com conhecimento da *chave base* é capaz de alterar o conteúdo do cartão e, caso ocorram muitas tentativas sem sucesso, o cartão pode ser desabilitado permanentemente, impedindo qualquer comunicação subsequente.

2.4 Protocolo de Comunicação TLS

A comunicação entre o dispositivo e o servidor desenvolvidos nesse trabalho é baseada no protocolo *Transport Layer Security* (TLS). Este protocolo, definido na RFC 5246 (ALLEN *et al.*, 2008), tem como objetivo principal prover privacidade e integridade dos dados transmitidos em uma comunicação entre duas partes. Ele é composto basicamente de duas camadas: o Protocolo de Registro (*Record Protocol*) e o Protocolo de Aperto de Mão (*Handshake Protocol*).

2.4.1 Protocolo de Registro

Segundo o texto, o protocolo de registro é a camada mais básica, utilizada para encapsular outros protocolos (entre eles o de *handshake)*, e sua segurança está baseada em dois fatores

- Confidencialidade: as mensagens são criptografadas com criptografia simétrica, utilizando como chave um segredo compartilhado entre as partes;

- Integridade e Autenticação de Mensagem: as mensagens incluem um *Message Authentication Code* (MAC).

Cada mensagem do protocolo de registro é composta por: um byte contendo o tipo de mensagem, dois bytes com a versão do TLS que está sendo utilizada, dois bytes com o tamanho do conteúdo da mensagem e o próprio conteúdo da mensagem. No texto, é utilizada uma construção de cifragem autenticada conhecida como *MAC-then-encrypt*, onde é criado um MAC com o conteúdo em texto claro (não cifrado) da mensagem e ambos, texto claro e MAC, são cifrados. Essa construção, contudo, é conhecida por possuir alguns problemas, entre eles: vulnerável ao ataque de oráculo de *padding* (*padding oracle attack*) (KATZ; LINDELL, 2014); não é genericamente segura, ou seja, não independe das funções de cifragem e MAC escolhidas (KRAWCZYK, 2001); insegura quando analisada sob diferentes noções de privacidade e integridade (BELLARE; NAMPREMPRE, 2000). Para resolver esse problema, foi definida no RFC 7366 (GUTMANN, 2014) uma extensão para o TLS que possibilita a utilização da construção *encrypt-then-MAC*, onde o texto claro é cifrado e o MAC é criado com a cifra. Essa construção não possui os problemas da anterior. Adicionalmente, cada MAC é criado com um número de sequência, o que garante proteção contra ataques baseados no reenvio de mensagens. A Figura 2.11 ilustra o formato de uma mensagem.

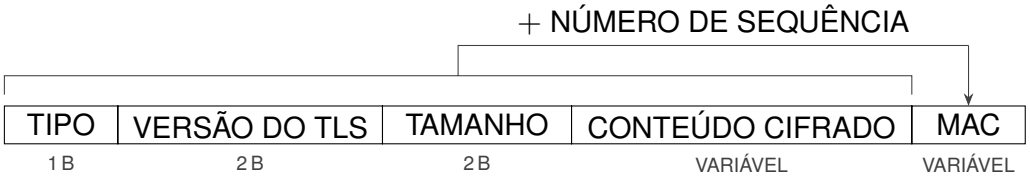


Figura 2.11: Estrutura de uma mensagem do Protocolo de Registro com cifragem autenticada

2.4.2 Protocolo de Aperto de Mão

O protocolo de *handshake*, por sua vez, estabelece uma comunicação mutualmente autenticada entre servidor e cliente, onde são definidos os algoritmos de criptografia e as chaves que serão utilizadas. A autenticação das partes é feita através de certificados e os algoritmos são determinados com a escolha de uma suíte de criptografia. Cada suíte é composta por:

- Um algoritmo de troca de chaves com criptografia assimétrica (e.g., RSA, DH), para determinação da chave secreta que será utilizada na cifragem dos dados;
- Um algoritmo de cifragem com criptografia simétrica (e.g., AES, 3DES), para cifrar os dados transmitidos;
- Um algoritmo que será utilizado nos MACs (e.g., HMAC-SHA, HMAC-SHA256), para garantir a integridade dos dados.

Função PRF. Antes de descrever como é realizado o *handshake*, é preciso definir a função PRF (do inglês, *Pseudorandom Function*), a qual é utilizada em várias etapas do protocolo em questão. Ela é uma função que tem como objetivo gerar uma quantidade arbitrária de números pseudoaleatórios a partir de uma função de *hash* e três parâmetros: *secret*, uma informação secreta; *label*, uma *string* em formato ASCII; *seed*, uma cadeia de bytes que serve como origem para o cálculo. Ela é dada por:

$$\begin{aligned}\mathsf{PRF}(secret, label, seed) = \mathsf{HMAC\text{-}hash}(secret,\ A(1) \parallel label \parallel seed) \parallel \\ \mathsf{HMAC\text{-}hash}(secret,\ A(2) \parallel label \parallel seed) \parallel \\ \mathsf{HMAC\text{-}hash}(secret,\ A(3) \parallel label \parallel seed) \parallel \ldots,\end{aligned}$$

com:

$$\begin{aligned}A(0) &= label \parallel seed \\ A(i) &= \mathsf{HMAC\text{-}hash}(secret,\ A(i-1)).\end{aligned}$$

É possível gerar uma quantidade arbitrária de bytes pseudoaleatórios com essa função. A quantidade de bytes pseudoaleatórios necessários e a quantidade de bytes que resultam da execução do HMAC (a qual varia com o algoritmo de *hash* escolhido) definem até que valor de $A(i)$ a função precisa ser executada, ou seja, definem o *critério de parada* do algoritmo. No caso do `master_secret`, por exemplo, são necessários apenas os 48 primeiros bytes obtidos. A Figura 2.12 ilustra o algoritmo, onde é possível ver seus diferentes elementos, entre eles: o conjunto de bytes $A(0)$, que serve como ponto de partida para o algoritmo; os conjuntos de bytes $A(i)$ seguintes, obtidos a partir da execução iterativa do HMAC em conjunto com o *secret*; os blocos de HMAC e concatenação; os blocos de bytes de saída, os quais são concatenados e utilizados como resultado final, após o alcance do critério de parada do algoritmo.

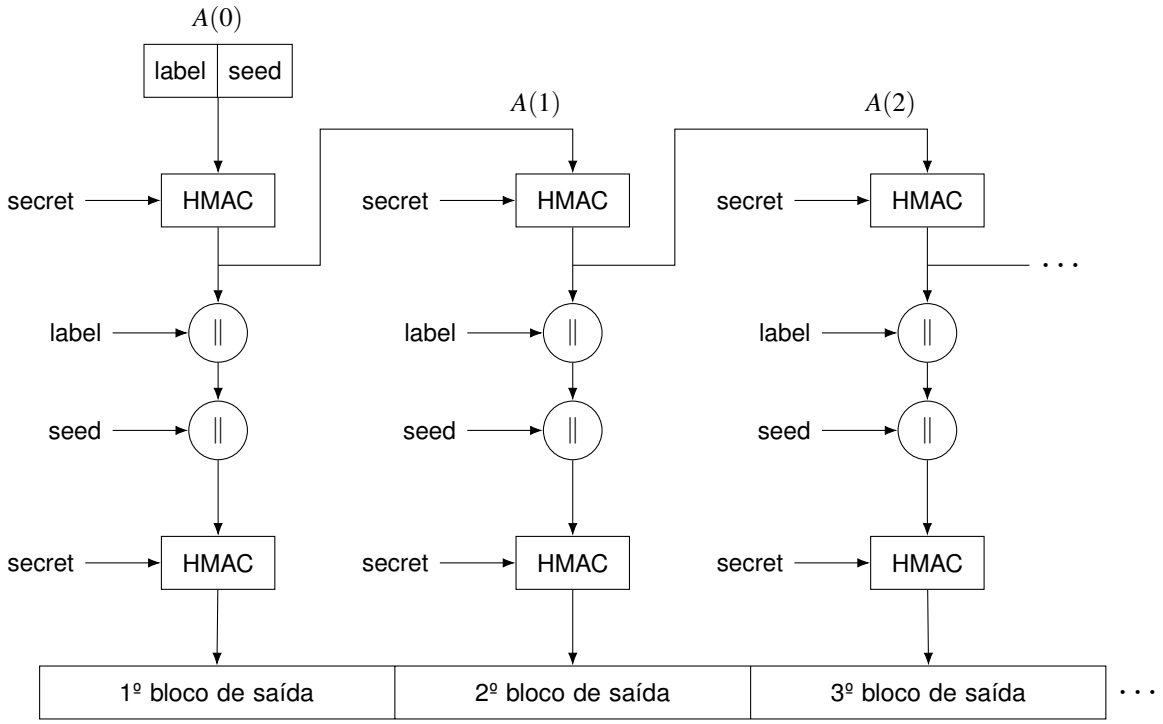


Figura 2.12: Diagrama de funcionamento da função PRF

Troca de Mensagens durante o *Handshake*. Cada mensagem do *handshake* é composta por: um byte contendo o tipo de mensagem (associado às diferentes etapas do *handshake*), três bytes com o tamanho do conteúdo e o próprio conteúdo, que varia de acordo com a etapa do *handshake*. Essa mensagem é, então, encapsulada em uma mensagem da Figura 2.11, com o tipo da mensagem correspondente ao valor definido na RFC 5246 para mensagens de *handshake*, igual a 0x16. Quando é iniciada uma nova conexão, nenhuma suíte de criptografia foi decidida, então o conteúdo das mensagens não é cifrado nem possui MAC. A Figura 2.13 ilustra o formato de uma mensagem do *handshake*.

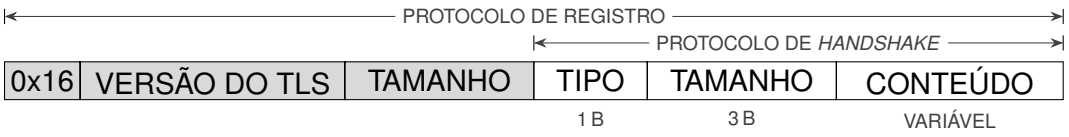


Figura 2.13: Estrutura de uma mensagem do Protocolo de *Handshake*, encapsulada no Protocolo de Registro

O *handshake* é iniciado com uma mensagem do cliente para o servidor, a *ClientHello*. Essa mensagem contém, entre outros dados: o `client_random`, valor aleatório de 32 bytes que será utilizado em alguns cálculos durante o protocolo; uma lista com as suítes de criptografia suportadas pelo cliente; uma lista com as extensões que o cliente deseja utilizar, como a extensão do *encrypt-then-MAC* ou as extensões definidas para o uso do TLS com curvas elípticas (BLAKE-WILSON *et al.*, 2006).

O servidor responde esta mensagem com a *ServerHello*, mensagem contendo também um valor aleatório de 32 bytes (`server_random`), a suíte criptográfica e as extensões que serão utilizadas nessa comunicação, escolhidas das listas enviadas pelo cliente. Após o envio da *ServerHello*, o servidor envia a *Certificate*, mensagem com um certificado (ou cadeia de certificados) para o cliente realizar sua autenticação. Caso o algoritmo de troca de chaves definido na suíte seja o DH com Chaves Efêmeras (geradas apenas para aquela conexão), o servidor envia a *ServerKeyExchange*, que consiste em uma mensagem com sua chave pública efêmera assinada digitalmente com a chave privada relativa ao seu certificado. A assinatura é gerada a partir da concatenação do `client_random`, do `server_random` e da própria chave efêmera:

$$\mathsf{ServerKeyExchange} = \mathit{chave} \parallel s(\mathit{client\_random} \parallel \mathit{server\_random} \parallel \mathit{chave}),$$

onde $\parallel$ denota concatenação e s é a função de assinatura digital. O servidor pode, opcionalmente, requisitar a autenticação do cliente enviando a mensagem *CertificateRequest*. Por fim, o servidor envia a *ServerHelloDone*, uma mensagem sem conteúdo, apenas para sinalizar a finalização das suas mensagens e esperar a resposta do cliente.

O cliente, por sua vez, se tiver recebido uma *CertificateRequest*, responde com uma mensagem *Certificate*, contendo um ou mais certificados que o servidor usará para autenticá-lo. Esta mensagem é seguida pela *ClientKeyExchange*, que será utilizada para estabelecer o `pre_master_secret`, valor secreto prévio usado no cálculo do `master_secret`, valor secreto final de 48 bytes utilizado no cálculo das chaves simétricas que irão cifrar e assinar os dados. Caso o algoritmo de troca de chaves seja o RSA, o cliente gera um `pre_master_secret` composto por sua versão do TLS (dois bytes) e mais 46 bytes aleatórios. O `pre_master_secret` é, então, cifrado com a chave pública do servidor e enviado. Caso o algoritmo de troca de chaves seja o DH com Chaves Efêmeras, o cliente (que já possui a chave pública do servidor) apenas enviará sua chave pública, de forma que o `pre_master_secret` será o segredo comum que resulta da execução do algoritmo. Após o envio da *ClientKeyExchange*, ambas as partes possuem o `pre_master_secret` e o `master_secret` pode ser obtido através da seguinte expressão:

$$\mathit{master\_secret} = \mathsf{PRF}(\mathit{pre\_master\_secret}, \text{“master secret”}, \mathit{client\_random} \parallel \mathit{server\_random}).$$

Caso o cliente tenha enviado a mensagem *Certificate*, ele agora envia a mensagem *CertificateVerify*, a qual contém uma assinatura gerada com a concatenação de todas as mensagens de *handshake* trocadas até o momento desde a *ClientHello*, utilizando a chave privada relativa ao certificado enviado pelo cliente para o servidor. Segundo o texto, essa mensagem provê a verificação explicita do certificado do cliente. Após o envio desta mensagem, todos os parâmetros de segurança da conexão foram estabelecidos. O cliente então envia a mensagem *ChangeCipherSpec*, sem nenhum conteúdo, que apenas sinaliza que a troca de mensagens dali em diante será realizada com os novos parâmetros de segurança definidos. Para finalizar o

handshake é necessário o envio da mensagem *Finished*, a qual tem como conteúdo 12 bytes que precisam ser verificados pelo recipiente, chamados de `verify_data`. Ele é calculado a partir da seguinte expressão:

$$verify\_data = \mathsf{PRF}(master\_secret,\ finished\_label,\ h(mensagens\_handshake)),$$

onde o *finished_label* é "client finished" para a mensagem enviada pelo cliente e "server finished" para a mensagem enviada pelo servidor e o *h(mensagens_handshakes)* é um *hash* de todas as mensagens de *handshake* trocadas até o momento concatenadas (tal qual no envio da mensagem *CertificateVerify*). Essa mensagem é enviada no formato da Figura 2.11, utilizando as chaves determinadas no *handshake*. No total, são geradas quatro chaves: duas para o cliente e duas para o servidor, uma delas para cifrar o texto (*write_key*) e outra para gerar os MACs (*write_MAC_key*). Dessa forma, as mensagens enviadas pelo cliente após o *handshake* são cifradas apenas com a *write_key* do cliente e autenticadas com a *write_MAC_key* do cliente, e o mesmo acontece com o servidor. Essas chaves são obtidas de um bloco de dados chamado `key_block`, o qual é calculado da seguinte maneira:

$$key\_block = \mathsf{PRF}(master\_secret,\ \text{“key expansion”},\ server\_random \parallel client\_random),$$

onde o número de bytes necessários depende dos algoritmos decididos no *handshake*. Após o recebimento da mensagem *Finished* do cliente, o servidor verifica se os dados estão corretos e responde com uma *ChangeCipherSpec* e uma *Finished* também, que será verificada pelo cliente. Com isso, o *handshake* é finalizado e podem ser trocadas mensagens relativas à aplicação do sistema. A Figura 2.14 resume a sequência de valores que são calculados no *handshake* até serem obtidas as chaves simétricas e a Figura 2.15 ilustra e sumariza o processo completo do *handshake* descrito nesta seção.

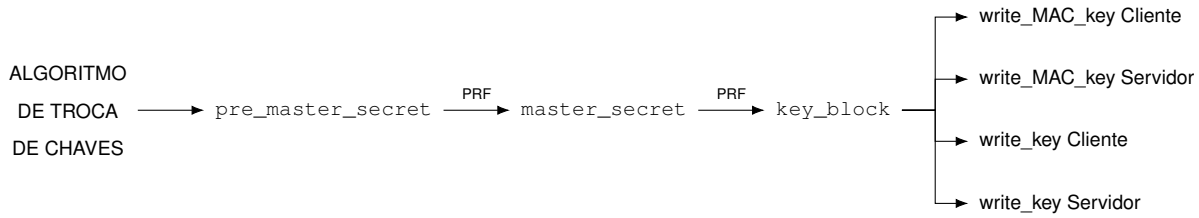


Figura 2.14: Sequência de obtenção das chaves simétricas no Protocolo de *Handshake*

Figura 2.15: Diagrama da troca de mensagens entre Servidor e Cliente, durante o Protocolo de *Handshake* do TLS

2.5 Padrão PKCS#15

Os PKCS são padrões elaborados pela empresa RSA Laboratories em conjunto com outras entidades relacionadas a segurança de sistemas, com o objetivo de fomentar o desenvol-

vimento de aplicações na área de Criptografia de Chave Pública. Entre eles, estão definidos os algoritmos RSA (PKCS#1) e DH (PKCS#3), fundamentais até hoje em aplicações que utilizam esse tipo de criptografia.

O padrão PKCS#15 (RSA Laboratories, 2000), em particular, especifica um formato para armazenamento de dados em um *token* criptográfico (e.g., *Smart Card*). A estrutura de dados definida neste padrão se baseia em três tipos de arquivos:

- *Master File* (MF): Arquivo obrigatório único, representa a raiz de toda a estrutura;
- *Dedicated Files* (DFs): Arquivos que podem conter outros arquivos, sejam eles outros DFs ou EFs;
- *Elementary Files* (EFs): Podem ser arquivos que contem informações e ponteiros para objetos armazenados no cartão (e.g., chaves, certificados), ou podem ser os próprios objetos em si. No caso de objetos que se relacionam, como um par de chaves pública e privada, seus EFs compartilham de um mesmo identificador. Esses arquivos não podem conter outros arquivos, como os DFs.

Existem quatro tipos de objetos que podem ser armazenados no cartão, de acordo com o PKCS#15: chaves (as quais podem ser públicas, privadas ou secretas), certificados, objetos de autenticação (e.g., PINs, senhas e padrões biométricos) e objetos externos genéricos. A formatação de arquivos segundo o PKCS#15 é análoga à formatação de arquivos em um computador pessoal, onde o MF é o Disco Local, os DFs são as pastas no disco e os EFs são os diferentes arquivos dentro de cada pasta.

Em uma estrutura padrão PKCS#15, o ponto de partida é o MF, o DF principal que contém todos os outros arquivos armazenados no dispositivo. Dentro dele, existe pelo menos um arquivo, o DF(PKCS#15) (chamado de Diretório da Aplicação PKCS#15), o qual contém todos os arquivos e objetos relacionados ao PKCS#15. Opcionalmente, podem ser criados outros DFs relativos a outras aplicações (inclusive outras aplicações PKCS#15) e o EF(DIR), um arquivo que lista todas as aplicações presentes no dispositivo.

Dentro do DF(PKCS#15), existem os EFs chamados Arquivos de Diretório (*Directory Files*), os quais guardam informações e ponteiros para os diferentes tipos de arquivos armazenados no dispositivo. O principal Arquivo de Diretório é o EF(ODF) (obrigatório), o qual guarda ponteiros para outros Arquivos de Diretório. Os Arquivos de Diretório restantes guardam ponteiros para diferentes objetos. São eles: EF(AODF), para objetos de autenticação; EF(PrKDF), para chaves privadas; EF(PuKDF), para chaves públicas; EF(SKDF), para chaves secretas; EF(CDF), para certificados; EF(DODF), para objetos externos. Esses EFs são todos opcionais, a depender do que será armazenado. É importante ressaltar que os Arquivos de Diretório guardam apenas referências aos objetos, e não os próprios objetos em si, estes ficam armazenados em outros endereços no dispositivo. Além dos Arquivos de Diretório, existem ainda os EF(TokenInfo) e EF(UnusedSpace). O primeiro é obrigatório e contém informações gerais sobre o *token* e o segundo é opcional e contém informações sobre os espaços livres no *token*.

Um ponto importante da especificação PKCS#15 é que todos os arquivos na sua estrutura possuem condições de acesso, as quais limitam quais arquivos podem ser criados, modificados ou removidos e quem pode executar tais operações. São definidas quatro condições, sumarizadas na Tabela 2.2. Dessa forma, arquivos com informações mais sensíveis, como chaves, só podem ser alterados ou utilizados em operações criptográficas mediante autenticação do usuário.

CONDIÇÃO	SIGNIFICADO
NEV	A operação nunca é permitida.
ALW	A operação sempre é permitida.
CHV	A operação só é permitida após autenticação do usuário.
SYS	A operação só é permitida com a apresentação de uma chave do sistema, geralmente disponível apenas para o fornecedor do *token*.

Tabela 2.2: Condições de acesso em uma estrutura de arquivos PKCS#15

A Figura 2.16 ilustra uma estrutura de arquivos PKCS#15 em um *token* criptográfico. As conexões pontilhadas representam arquivos opcionais, as tracejadas representam referências que um arquivo faz a outro (ou outros) e as cheias, por sua vez, representam os arquivos obrigatórios.

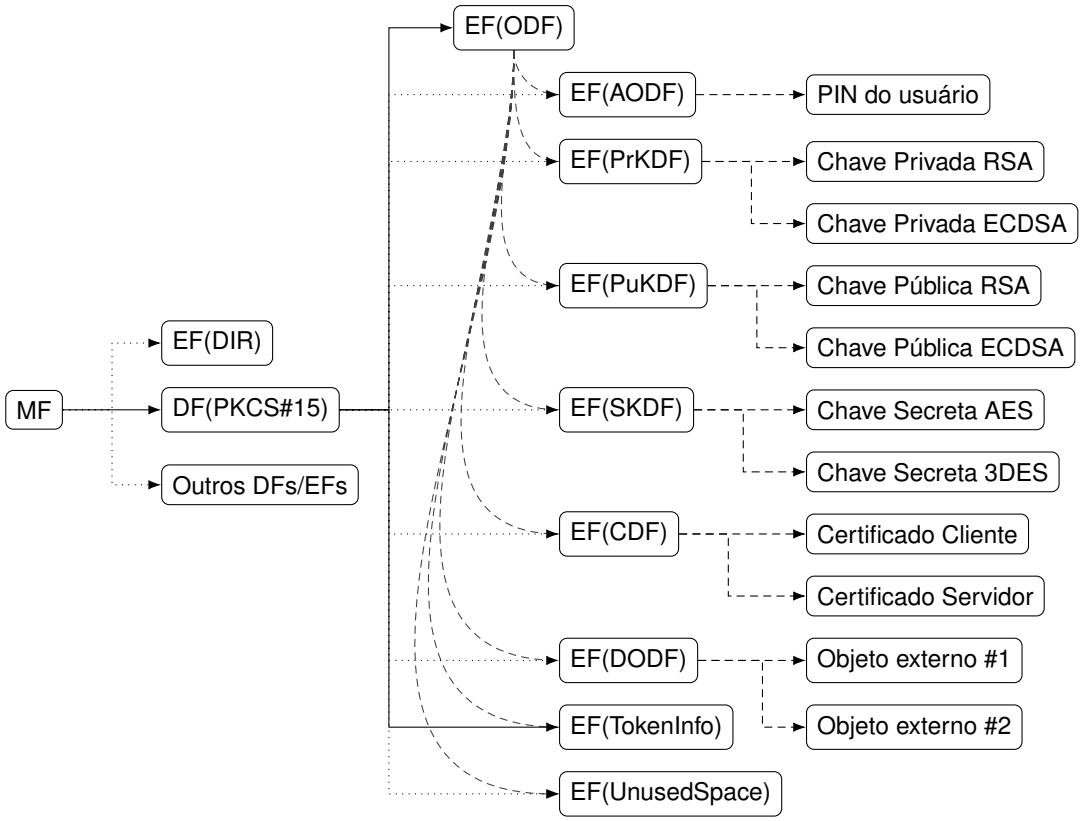


Figura 2.16: Representação gráfica do sistema de arquivos PKCS#15

2.6 Notação ASN.1

A *Abstract Syntax Notation One* (ASN.1) é uma maneira de representar diferentes tipos de dados de forma abstrata, através de regras de escrita bem definidas. Esta notação é especificada na Recomendação X.680 (ITU-T, 2008) e é utilizada em outros padrões da área de tecnologia como, por exemplo, o PKCS#15 e o TLS. O padrão X.680 define vários tipos simples de dados, como inteiros, booleanos, cadeias de bits, cadeias de bytes, sequências de objetos, entre outros, os quais podem ser utilizados para a criação de tipos mais complexos, relativos a uma determinada aplicação. Na Tabela 2.3 estão demonstrados alguns exemplos de definições em ASN.1, retirados do PKCS#15.

EXEMPLO	**DESCRIÇÃO**
`pkcs15-ub-reference INTEGER ::= 255`	Declaração de uma constante nomeada *pkcs15-ub-reference*, com um valor inteiro igual a 255.
`Reference ::= INTEGER (0..pkcs15-ub-reference)`	Declaração de um novo tipo de dado chamado *Reference*, o qual é um inteiro de valor no intervalo de 0 a *pkcs15-ub-reference*, a constante definida previamente.

`CommonKeyAttributes ::= SEQUENCE {`
`  iD  Identifier,`
`  usage  KeyUsageFlags,`
`  native  BOOLEAN DEFAULT TRUE,`
`  accessFlags  KeyAccessFlags OPTIONAL,`
`  keyReference Reference OPTIONAL,`
`  startDate  GeneralizedTime OPTIONAL,`
`  endDate  [0] GeneralizedTime OPTIONAL,`
`}`
	Declaração de um segundo tipo de dado, *CommonKeyAttributes*, composto por uma sequência de várias informações, cada uma de um tipo diferente. Entre essas informações está a *keyReference*, que é do tipo *Reference*. É possível ver outros tipos de dados definidos pelo desenvolvedor, como *Identifier*, *KeyUsageFlags* e *KeyAccessFlags*, dos quais só é possível saber o significado olhando sua declaração no documento, como foi feito com o tipo *Reference*.

Tabela 2.3: Exemplos de definições escritas na notação ASN.1

Quando esses dados são utilizados em sistemas digitais, eles precisam ser traduzidos para conjuntos de bytes, os quais podem ser lidos, modificados e transmitidos. O método para codificar em bytes objetos escritos com a ASN.1 está definido na Recomendação X.690 da ITU (ITU-T, 2002). Ele define um conjunto de regras principais conhecidas como *Basic Encoding Rules* (BER). Essas regras utilizam a construção *Tag-length-value* (TLV), onde cada objeto é representado com três campos distintos: *tag*, comprimento e valor. A *tag* identifica o tipo do dado e contém três informações:

- A classe do dado: universal (reservada para os tipos de dados definidos na ASN.1), aplicação, específico do contexto e privado (escolhidas pelo usuário);
- Se o dado é primitivo, o campo valor representa o dado diretamente, ou construido, o campo valor representa um conjunto de outros tipos de dados;
- O número identificador relativo ao tipo do dado.

A *tag* pode ser codificada em um ou mais bytes, dependendo do tamanho do número identificador. A Tabela 2.4 exemplifica algumas *tags* de dados definidas no padrão X.680.

TAG	CARACTERÍSTICAS	TIPO
0x01	Universal, Primitivo, Número 1	Booleano (`BOOLEAN`)
0x02	Universal, Primitivo, Número 2	Inteiro (`INTEGER`)
0x03	Universal, Primitivo, Número 3	Cadeia de Bits (`BITSTRING`)
0x04	Universal, Primitivo, Número 4	Cadeia de Bytes (`OCTETSTRING`)
0x30	Universal, Construído, Número 16	Sequência de Dados (`SEQUENCE`)

Tabela 2.4: Tipos de dados da notação ASN.1

O comprimento dos dados representa a quantidade de bytes presentes no campo valor. Ele pode ser codificado em um ou mais bytes, dependendo do tamanho dos dados. Na sua forma curta, com apenas um byte, o oitavo bit é sempre igual a 0 e os sete bits restantes guardam o tamanho, o qual pode variar de 0 a 127. Na sua forma longa, com vários bytes, o primeiro byte possui o oitavo bit sempre igual a 1 e os sete bits restantes guardam a quantidade de bytes subsequentes necessários para armazenar o tamanho. Por exemplo, um campo comprimento dado por 0x08 guarda um tamanho de 8 bytes, enquanto que um campo comprimento dado por 0x820100 guarda um tamanho de 256 bytes.

Para exemplificar a codificação de um determinado dado representado com ASN.1, pode-se usar a constante *pkcs15-ub-reference* definida previamente. Ela é um inteiro de valor 255 (0xFF em hexadecimal e 11111111 em binário). Como os inteiros em ASN.1 possuem sinal (representação em complemento de dois), são necessários dois bytes para representar o valor 255 (um único byte de valor 11111111 representaria o inteiro -1). Sua *tag* é a que está definida na Tabela 2.4 para inteiros e o tamanho é igual a 2. Portanto, a representação de acordo com as BER é dada por 0x020200FF.

Uma ferramenta muito utilizada durante o desenvolvimento deste trabalho para verificação de dados codificados dessa forma foi a asn1js (LUCHINI, 2017). Este *software* recebe como entrada os dados codificados em formato hexadecimal, processa-os e retorna a representação em ASN.1.

2.7 Considerações Finais

Cada aspecto do desenvolvimento do sistema proposto neste trabalho é especificado em padrões desenvolvidos e adotados pela indústria e pela academia. Assegurar a conformidade com tais padrões significa garantir a interoperabilidade do sistema com outros sistemas e projetos que também estejam de acordo com os mesmos documentos. Outra questão particularmente importante quando se trata de segurança é a necessidade de se utilizar métodos e algoritmos de conhecimento publico, os quais estão sob constante averiguação da sua eficácia. Tais métodos e algoritmos estão, também, detalhados em especificações citadas neste capítulo.

3

TRABALHOS RELACIONADOS

Nesta seção são apresentados trabalhos acadêmicos que se relacionam com o tema deste trabalho. As pesquisas dos textos foram realizadas nas bibliotecas digitais *IEEE Xplore* e *ACM*. Os termos de busca utilizados foram as palavras-chave definidas no resumo deste trabalho e termos relacionados (e.g., sinônimos, plurais, diferentes formas de escrever). A Tabela 3.1 sumariza os termos citados.

PALAVRAS-CHAVE E TERMOS RELACIONADOS
Internet of Things, Internet-of-Things
Security, Secure, Privacy, Authentication
Smart Card, Smart Cards, Smartcard, Smartcards
Java Card, Java Cards, Javacard, Javacards
Microcontroller, Microcontrollers, Microprocessor, Microprocessors

Tabela 3.1: Termos de pesquisa utilizados na busca de trabalhos relacionados

As pesquisas consistiram em diferentes combinações dos termos da tabela, de forma a obter resultados mais específicos ou mais generalizados. As combinações são feitas nas expressões de busca, as quais permitem a utilização de operações lógicas como AND e OR para agrupar os termos. Os textos foram selecionados entre os 100 primeiros de cada pesquisa, e todas as pesquisas foram limitadas a publicações datadas de no máximo 15 anos. É necessário um intervalo de tempo maior, pois apesar de IoT ser um assunto muito recente, pesquisas com *Smart Card*s e *Java Cards* existem há mais tempo.

A Figura 3.1 mostra diagramas de Venn, os quais destacam as diferentes buscas realizadas. As diferentes tonalidades representam a quantidade de termos agrupados. É importante ressaltar que a área de cada região nos diagramas não tem relação com a quantidade de textos encontrados, é apenas uma representação para visualizar as diferentes combinações de termos que foram utilizadas. Como os Java Cards fazem parte de um subgrupo dos Smart Cards, eles foram tratados como sinônimos nas pesquisas. Pode-se observar que as pesquisas se concentraram principalmente em trabalhos envolvendo *Smart Cards* e microcontroladores. Também foi feita uma pesquisa apenas com os termos de IoT e segurança, com o objetivo de ver soluções gerais

de segurança para IoT, bem como avaliações de segurança e *surveys*. A maioria dos resultados encontrados foi relacionada a essa última pesquisa.

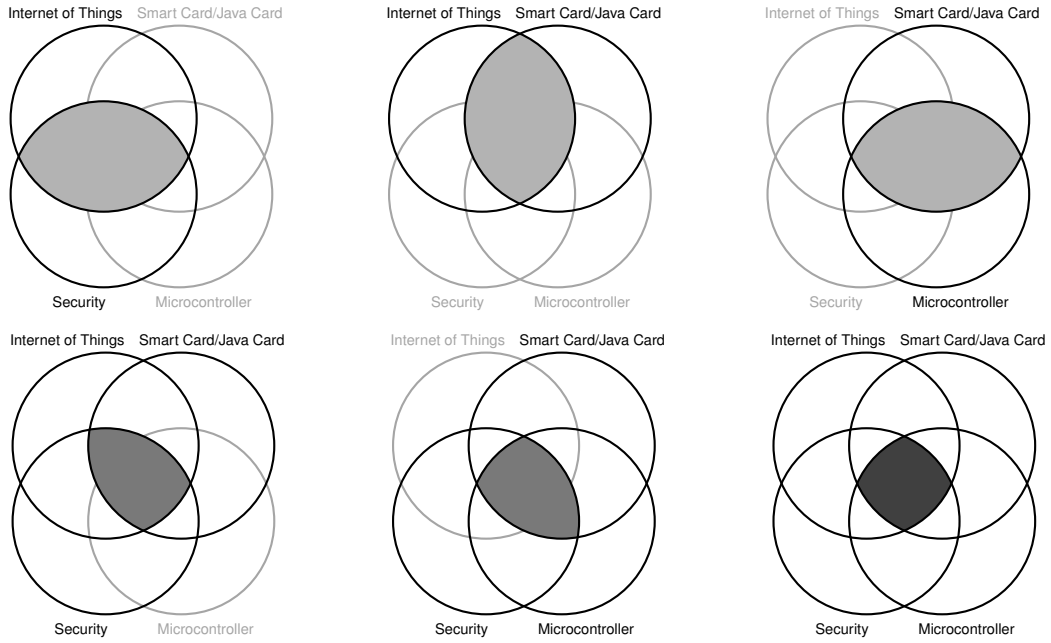


Figura 3.1: Diagrama das pesquisas de trabalhos relacionados realizadas

3.1 Aplicação de Smart Cards em Protocolos de Segurança

Badra e Urien (2008) apresentam uma proposta de sistema que utiliza *Smart Cards* para implementar o Protocolo TLS, aproveitando a segurança física que o cartão fornece. A arquitetura descrita no texto estabelece que tanto o cliente quanto o servidor possuem um SC, com os quais eles se comunicam. A aplicação não é específica para IoT, nem detalha o *hardware* dos clientes (é descrito um teste com telefones celulares). A principal diferença entre o *software* do texto e o que foi desenvolvido neste trabalho é que todos os parâmetros do *handshake* ficam armazenados no próprio cartão, e as entidades precisam enviar comandos ao mesmo para obter esses dados. Pode-se, inclusive, extrair do cartão as chaves negociadas após o *handshake*, algo que o sistema proposto neste trabalho não permite.

3.2 Conexão entre Microcontroladores e Smart Cards

A base da segurança do dispositivo IoT proposto neste trabalho consiste na integração entre seu microcontrolador e um *Smart Card* com capacidades criptográficas. Foram encontrados alguns trabalhos acadêmicos que também propõem essa integração.

Kim *et al.* (2008) descrevem a utilização de um *System-on-Chip* (SoC) com um processador RISC de 32 bits e um módulo criptográfico AES em um leitor de SCs para a verificação de dados biométricos contidos em cartões. Nessa aplicação, o cartão pertence a um usuário e

contém informações biométricas do mesmo. Embora esse trabalho possua uma implementação realizada, seus objetivos são distintos do que é proposto neste trabalho, pois não existe um foco em IoT e o cartão não é utilizado como um módulo criptográfico para o sistema, apenas como um objeto para identificação.

Outros trabalhos estão concentrados especificamente na conexão entre microcontroladores e *Smart Card*s, um dos pontos principais do desenvolvimento deste trabalho. Muji *et al.* (2008), por exemplo, apresentam uma simulação de comunicação entre um PIC (família específica de microcontroladores desenvolvida pela *Microchip Technology*) e um *Smart Card*, o qual é modelado por uma memória EEPROM, para aplicação em segurança de veículos. Nessa aplicação, o cartão possui uma informação de identificação armazenada previamente em sua memória por meio de uma interface gráfica em um computador. A informação é reconhecida pelo microcontrolador (que também está conectado ao carro) e permite ao usuário dar a partida no veículo. O texto não menciona a utilização das capacidades criptográficas do cartão, apenas do seu armazenamento seguro de informações, nem a utilização de redes de comunicação. Por fim, Dichou, Tourtchine e Rahmoune (2015) realizam uma simulação da troca de APDUs entre um *Smart Card*, o qual é modelado por um PIC e uma memória EEPROM, e um segundo PIC. A simulação implementa a recepção do ATR do cartão e o envio de três comandos definidos na ISO7816: `VERIFY`, `SELECT` e `UPDATE RECORD`. Não existe uma aplicação com o cartão nesse texto, nem a utilização de suas capacidades criptográficas. Em suma, os trabalhos encontrados que relacionam *Smart Card*s, microncontroladores e segurança não são direcionados para IoT, e mesmo aqueles que se propõem a integrar cartões e microcontroladores apenas realizam simulações simples.

3.3 Segurança na Internet das Coisas

3.3.1 Visão Geral

Uma grande parte dos textos encontrados nas pesquisas sobre segurança na IoT analisam o seu status atual, identificando problemas e desafios e propondo soluções. Kanuparthi, Karri e Addepalli (2013) identificam quatro desafios existentes na consolidação da IoT e discutem possíveis soluções de *hardware* e segurança embarcada. Os desafios são: garantia de origem e integridade dos dados; gerenciamento de identidades; gerenciamento de confiança; privacidade. Os três primeiros desafios, segundo os autores, podem ser vencidos com a utilização de *Physical Unclonable Functions* (PUFs), dispositivos que são o equivalente em *hardware* a uma função de sentido único (*one-way function*, do inglês): respondem a um estímulo de uma forma difícil de ser prevista ou replicada, devido à introdução de fatores aleatórios durante seu processo de fabricação. Herder *et al.* (2014) apresentam uma extensa análise sobre PUFs, suas implementações e aplicações. A questão da privacidade, por sua vez, pode ser resolvida com a utilização de algoritmos leves de cifragem.

Abomhara e Køien (2014) apresentam uma visão geral do estado da IoT, discutindo

pontos como arquiteturas existentes, domínios de aplicação, tecnologias, ameaças, desafios e requerimentos de segurança e privacidade. Os maiores desafios, segundo os autores, são a privacidade do usuário e proteção de informações; gerenciamento de identidade e autenticação; gerenciamento de confiança e integração de políticas; controle de acesso e autorização; segurança fim-a-fim; soluções de segurança resistentes a ataques. Bertino *et al.* (2016) identificam desafios já citados previamente, como controle de acesso, segurança de dados, autenticação de dispositivos e gerenciamento de identidades, e adicionam outros itens, tais como: segurança de *middleware*, gerenciamento de *patches*, descoberta de dispositivos e defesa de perímetro.

Foram publicadas também algumas *surveys* sobre segurança na IoT. Uma delas, de autoria de Pawar e Ghumbre (2016), mostra uma visão geral de aplicações e serviços da IoT voltados para assistência médica. Além disso, também identifica desafios de segurança, de acordo com os outros trabalhos citados, sumariza trabalhos publicados com propostas de solução para cada desafio, e sugere a utilização de algoritmos criptográficos em aplicações IoT, como AES e RSA (ambos aplicados neste trabalho). Outra *survey*, desenvolvida por Yang *et al.* (2017), mais extensa que a anterior, faz referências a trabalhos relacionados a diferentes características, entre elas: limitações dos dispositivos IoT, como tempo de bateria e restrições de processamento; classificação dos ataques à IoT; esquemas e arquiteturas para autenticação na IoT; segurança da IoT em suas diferentes camadas. Essa última análise considera a IoT dividida em quatro camadas distintas: Camada de Aplicação (e.g., casas inteligentes, sistemas de saúde); Camada de Transporte (e.g., TLS, DTLS); Camada de Rede (e.g., IP); Camada de Percepção (e.g., redes de sensores).

3.3.2 Propostas de Implementação e Verificação

Hummen *et al.* (2013) estudam a viabilidade de se utilizar certificados para autenticação na IoT, levando em conta as restrições existentes nos dispositivos que fazem parte da rede. O protocolo considerado na pesquisa é o DTLS, uma versão do TLS para o transporte de dados através de datagramas (pacotes utlizados em conexões onde entrega, hora de chegada, e a ordem das informações não são garantidas). Os autores concluem que a utilização de certificados pode introduzir uma sobrecarga considerável para dispositivos com muitas restrições. O projeto deste trabalho implementa a autenticação de clientes e servidores com certificados, mas o próprio servidor é a Autoridade de Certificação raiz (ou CA raiz, definida na Seção 2.1), de forma que não existem longas cadeias de certificados a serem verificados. Ainda assim, é necessária a realização de uma avaliação minuciosa de performance para determinar a viabilidade dessa aplicação. Liu *et al.* (2016) propõem uma solução para autenticação na IoT através da criação de uma CA privada. No exemplo descrito no texto, um hotel possui um servidor de CA que emite certificados para os hóspedes, os quais se autenticam com o sistema de controle automático dos quartos. A autenticação é realizada com o Protocolo TLS. Este trabalho, como mencionado anteriormente, propõe uma solução onde o servidor local é a própria CA.

Muitos artigos propõem novos algoritmos e esquemas de autenticação e/ou cifragem de

dados para IoT. Como o propósito deste trabalho não é propor um novo algoritmo, mas sim implementar um algoritmo existente e amplamente utilizado em aplicações de segurança, esses textos foram considerados fora de escopo e não são citados.

Tekeoglu e Tosun (2016) desenvolvem uma plataforma de testes para dispositivos conectados à IoT. Com a utilização de *hardwares* e *softwares* especializados para captura de pacotes (incluindo o *Wireshark*, também utilizado neste trabalho), os autores são capazes de analisar o tráfego de mensagens de cinco tipos diferentes de dispositivos: *dongles* HDMI para *streaming* de mídia, câmeras, *drones*, *smartbands* e *smartwatches*, os quais se comunicam via Wi-Fi e Bluetooth. Em posse dos dados de transmissão, são realizados experimentos para avaliar diferentes aspectos da comunicação, entre eles: testes de vulnerabilidades com *softwares* específicos; investigação das suítes de criptografia utilizadas no Protocolo TLS por cada dispositivo; observação de *updates* de *firmware* não criptografados; segurança das senhas, dos aplicativos para *smartphones*, da nuvem e dos dados transmitidos. Os autores concluem que os variados tipos de dispositivos possuem sérias vulnerabilidades e ressaltam a dificuldade de testar cada um deles.

3.3.3 Segurança para Dispositivos IoT de Baixo Custo

No que diz respeito à implementação de segurança na IoT especificamente com dispositivos de baixo custo (e.g. Arduino), foram encontrados alguns textos não acadêmicos (fora da pesquisa descrita no início do capítulo) com propostas de implementação. Ribeiro (2012), por exemplo, desconsidera a implementação da segurança no próprio dispositivo, por suas restrições de *hardware*, algo que pode ser resolvido com a proposta deste trabalho. A solução do autor envolve a comunicação segura entre dois servidores MQTT: um interno, em uma rede local em contato direto com cada dispositivo IoT e um externo, conectado à internet. É uma solução possível, porém não existe uma autenticação para cada dispositivo e a comunicação local ainda é insegura. Ardiri (2014) propõe algo semelhante ao que se pretende fazer neste trabalho, um acordo de chave simétrica AES entre cliente (um Arduino UNO) e servidor para comunicação criptografada utilizando o algoritmo RSA. No entanto, o protocolo proposto não envolve autenticação de nenhuma das partes.

3.4 Considerações Finais

A segurança da IoT é um tema de extrema importância no contexto atual. Como mencionado anteriormente, a pesquisa de publicações envolvendo os termos de IoT e segurança retornou a maior quantidade de resultados. Foi obtido um total de 832 textos, dos quais 452, mais da metade, foram publicados a partir do ano de 2016. Existem diversas análises do estado atual da IoT, bem como inúmeras propostas para adicionar segurança à IoT nas suas diversas camadas de operação. No entanto, a utilização de *Smart Cards* integrados em dispositivos IoT, da forma como é feita neste trabalho, é uma solução pouco explorada na literatura, com um total

de 7 resultados encontrados nas pesquisas relacionando os termos de IoT, segurança e *Smart Cards*. Embora existam propostas de integração entre SCs e microcontroladores e também um trabalho utilizando cartões em uma comunicação através do protocolo TLS, não foi encontrado um trabalho que una essas idéias em um contexto de Internet das Coisas.

4

PROPOSTA DE ARQUITETURA

A arquitetura proposta neste trabalho está ilustrada na Figura 4.1. Ela é composta por dispositivos IoT, os quais atuam como clientes, e um computador, que atua como servidor. Esses componentes se comunicam em uma conexão Wi-Fi (Protocolo 802.11) através de um *socket* TCP.

O computador roda uma aplicação escrita em Python, *TCPServer.py*, a qual realiza a configuração dos clientes previamente à sua utilização e, posteriormente, estabelece uma comunicação segura com os mesmos através dos protocolos de *Handshake* e Registro do TLS descritos nas subseções 2.4.1 e 2.4.2, respectivamente. Essa aplicação é baseada no módulo *open source* Cryptography, que implementa funções criptográficas utilizadas na aplicação e também dá suporte à utilização de certificados.

O dispositivo IoT, por sua vez, é composto por três partes principais: microcontrolador, *Smart Card* e sensores. O microcontrolador (MCU) é o "cérebro" do dispositivo. Nele está instalada a aplicação principal, *WifiClient.ino*, desenvolvida no Arduino IDE. Esta aplicação é responsável pela comunicação segura entre o cliente e o servidor, e também pela comunicação interna entre MCU, *Smart Card* e sensores. Ela é construída a partir de diferentes bibliotecas, as quais podem ser visualizadas na Figura 4.1. Com exceção da ArduinoSCLib (que apenas foi modificada e expandida) e da ArduinoDES, todas as bibliotecas foram criadas e desenvolvidas durante o trabalho. O *Smart Card* (SC) é responsável pela realização das operações criptográficas e cálculos do TLS no cliente, através da API Java Card (Subseção 2.3.3), e por armazenar dados, como chaves e certificados, de forma segura na sua memória, utilizando uma estrutura de arquivos PKCS#15 (Seção 2.5). Nele está instalada uma versão modificada e expandida do *applet open source* IsoApplet, que dá suporte às atividades citadas. A comunicação entre o MCU e o SC é feita através de uma interface serial, por onde são trocadas APDUs, descritas na Subseção 2.3.2. Por fim, os sensores realizam medidas e as enviam para o MCU, o qual as envia para o servidor em mensagens criptografadas e autenticadas. A conexão dos sensores pode ser feita através de GPIOs ou por um barramento SPI (um tipo de comunicação serial).

Os detalhes do desenvolvimento da plataforma de *hardware* dos dispositivos IoT e da arquitetura de *software* do sistema são especificados nas seções 4.1 e 4.2.

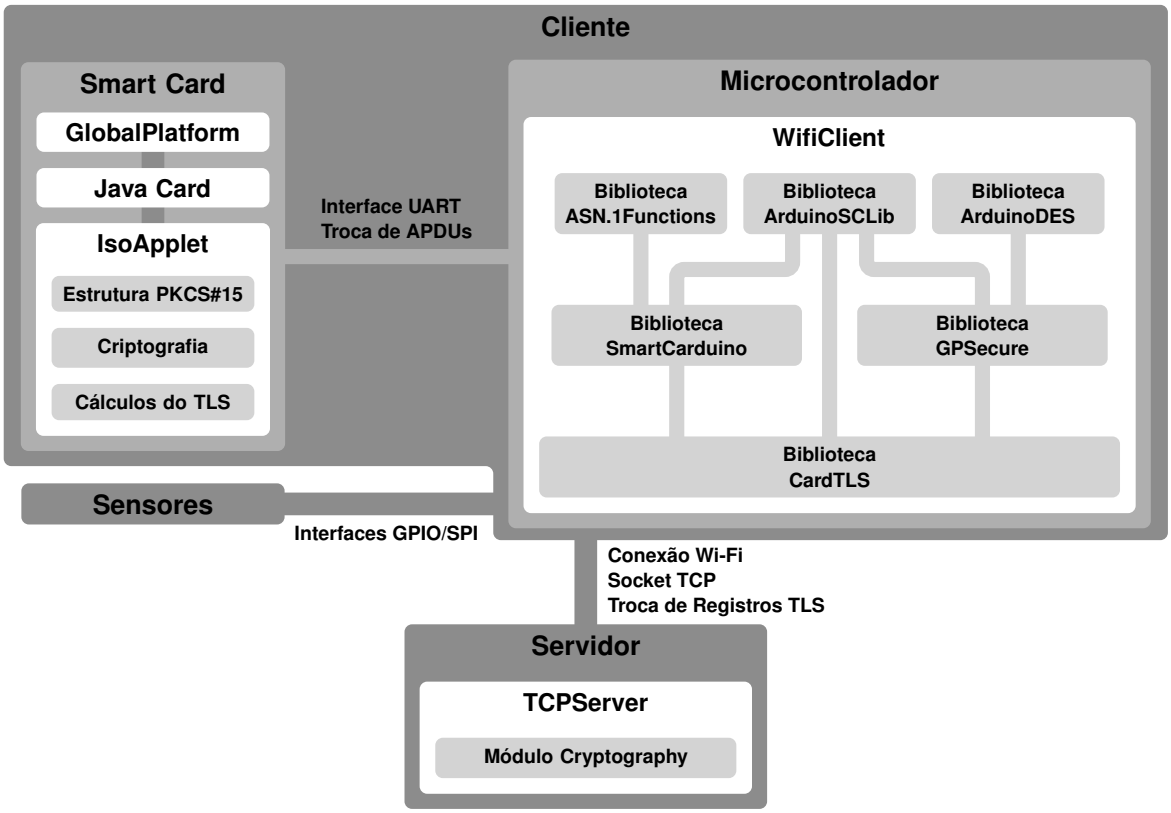


Figura 4.1: Arquitetura geral do sistema proposto neste trabalho

4.1 Plataforma de Hardware

O objetivo mais básico de *hardware* era estabelecer uma conexão entre o SC e um *Microcontroller Unit* (MCU). Testes iniciais foram realizados com Arduino UNO, uma das plataformas de desenvolvimento de baixo custo mais utilizadas para rápida prototipação de projetos eletrônicos (de acordo com pesquisa realizada pelo Hackster (2016)), que possui o MCU *ATmega328P*. Esta placa foi escolhida por sua abordagem *open source* (*hardware* e *software*), simplicidade de programação no Arduino IDE, possibilidade de extensão do código com bibliotecas escritas em C/C++ e vasta comunidade de usuários (aproximadamente 400.000 no fórum oficial), os quais disponibilizam recursos e informações. As características do SC utilizado estão sumarizadas na Tabela 4.1. Essas informações foram obtidas com o *software open source* GlobalPlatformPro (PALJAK, 2016).

4.1.1 Placas leitoras de Smart Cards utilizando Arduino UNO

A biblioteca *open source* ArduinoSCLib (BARGSTEDT, 2016), desenvolvida para Arduino e placas compatíveis com Arduino e que foi utilizada no projeto, implementa o envio e recebimento de informações para o SC, detalhados na Seção 4.2. Ela permite uma ligação direta entre a placa e o cartão, sem a necessidade de *chips* adicionais, o que pode ser visto na Figura 4.2. O VCC do cartão é ligado em um pino digital do Arduino, e não na saída de

ATR	3B FE 18 00 00 80 31 FE 45 80 31 80 66 40 90 91 06 2D 10 83 01 90 00 D3
Fabricante do CI	Infineon Technologies
GlobalPlatform	Versão 2.1.1
Java Card	Versão 2.2.2
Tamanho	ID-1
Classe	A

Tabela 4.1: Informações do *Smart Card* utilizado neste trabalho

alimentação de 5 V, uma vez que é necessário controlar quando o SC está ligado ou desligado, como descrito na Seção 4.2. Tanto o VCC quanto os outros terminais do cartão que precisam ser ligados em pinos digitais do Arduino (RST e I/O) podem ser conectados em qualquer pino digital, dependendo das necessidades do projeto. O terminal CLK, no entanto, precisa ser ligado no pino D9, pois o mesmo está atrelado ao módulo de *timer* do Arduino, o qual gera o sinal de relógio. A ISO7816-3 (ISO/IEC, 2006) determina que o relógio deve ter frequência entre 1 MHz e 5 MHz, com o valor máximo podendo ser menor de acordo com o que é suportado pelo cartão. O valor máximo de frequência suportado por cada SC é informado no ATR. No cartão utilizado no projeto, o valor máximo é de 5 MHz, mas a comunicação sem erros utilizando o Arduino UNO só foi possível com frequência de 1 MHz. Para valores maiores de frequência, as mensagens não foram recebidas corretamente (e.g., o ATR, que é um valor fixo, era diferente do esperado). Possíveis causas para esse problema podem ser: uma limitação de velocidade de leitura da porta digital da placa, associada à velocidade de processamento do próprio Arduino UNO (limitação de *hardware*); implementação de leitura de dados da biblioteca ArduinoSCLib não otimizada, fazendo com que sejam utilizados mais ciclos de processamento do que a quantidade suficiente para realizar a leitura dos bytes com uma frequência maior (limitação de *software*).

O acesso aos terminais do SC é feito com a utilização de um *slot* para leitor de cartão. O *slot* utilizado no projeto possui uma chave normalmente aberta entre os terminais C4 e C8 do SC, os quais não possuem função no sistema (NC significa *Não Conectado*). Quando o SC é inserido, a chave fecha e a presença do cartão pode ser detectada por um pino digital do Arduino (e.g., pino D5, na Figura 4.2). A Figura 4.3 mostra o formato do *slot* e como o cartão é inserido para leitura. É importante ressaltar que apesar de o *slot* ser feito para SCs de tamanho ID-000, é possível adaptá-lo para utilização com cartões maiores e isso foi feito nas Placas de Circuito Impresso (PCIs) iniciais do projeto.

As primeiras placas do projeto foram desenvolvidas no *software* EAGLE (*Easily Applicable Graphical Layout Editor*), desenvolvido pela CadSoft Computer, subsidiária da Autodesk desde 2016. O EAGLE é um programa de *Electronics Computer-aided Design* (ECAD), categoria de *softwares* que possuem um conjunto de ferramentas para elaboração de sistemas

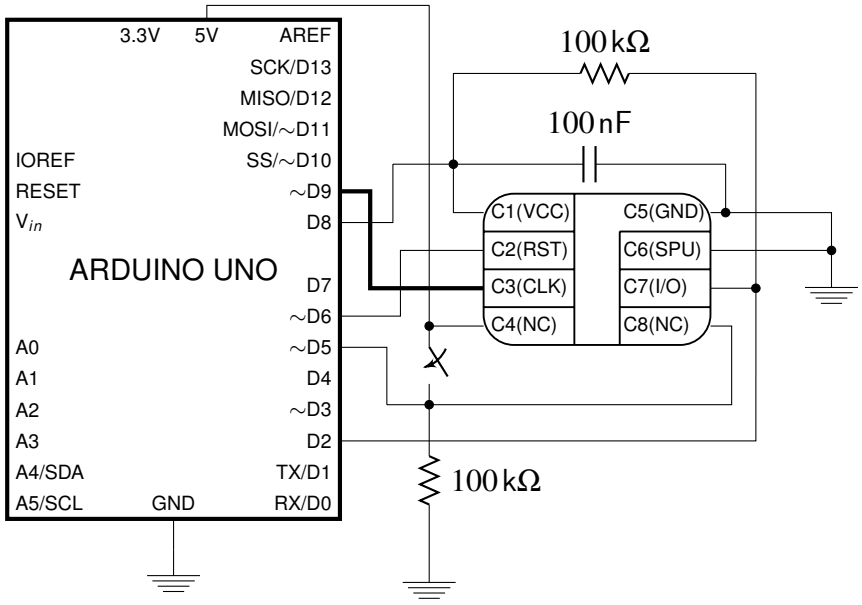
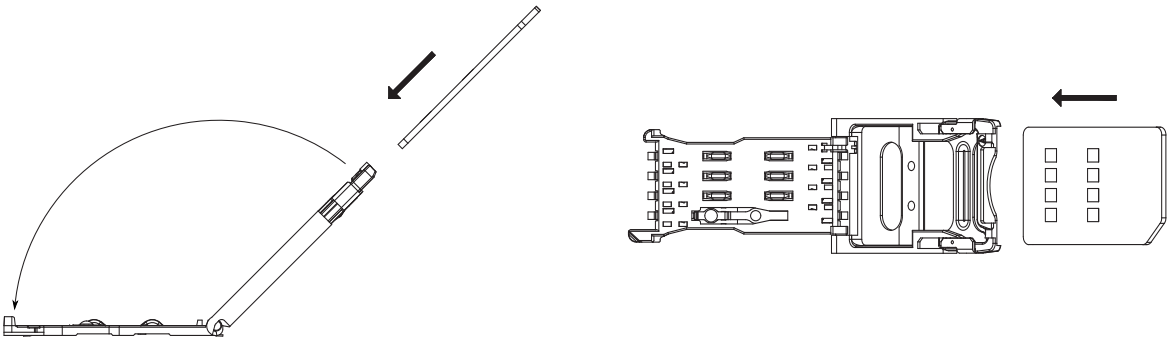


Figura 4.2: Conexão entre Arduino UNO e *Smart Card*, evidenciando a conexão fixa entre o terminal CLK e o pino D9

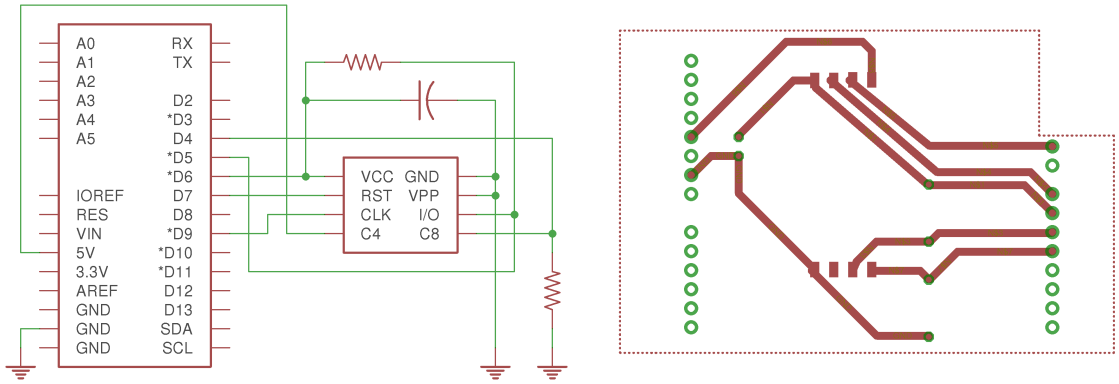


(a) **(b)**

Figura 4.3: Visão lateral (a) e superior (b) do *slot* para o *Smart Card*. *Fonte: ckswitches.com*

eletrônicos, como CIs e PCIs. Ele foi escolhido inicialmente por ser utilizado pela maioria das comunidades de projetos eletrônicos mais acessadas pelos usuários (Hackster, 2016). Todas fornecem os arquivos de projeto das suas placas no formato do EAGLE.

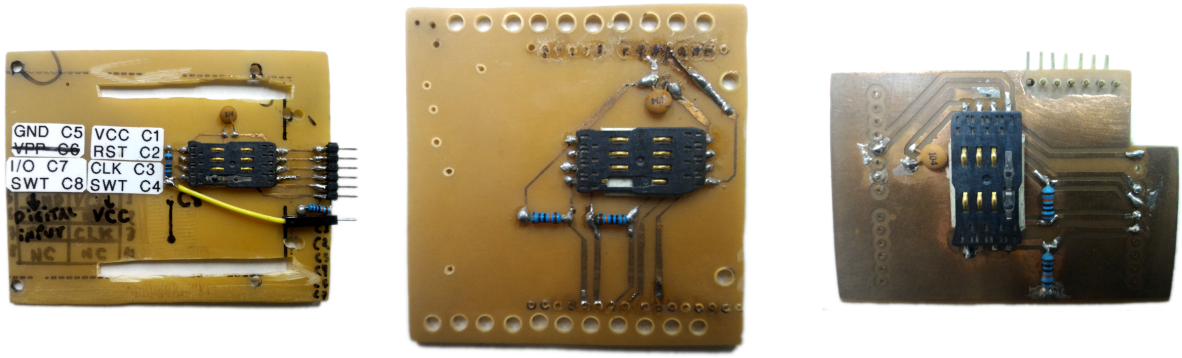
No programa, é possível construir esquemáticos de circuitos (representação abstrata, para mostrar o funcionamento) e desenhar *layouts* de Placas de Circuito Impresso. O aplicativo possui uma vasta coleção de bibliotecas de símbolos (representações gráficas dos componentes, posicionados no esquemático do circuito) e *footprints* (representações gráficas das regiões de cobre onde os componentes serão soldados, posicionados no *layout* da placa), que pode ser expandida com outras bibliotecas criadas por terceiros ou pelo próprio usuário. A Figura 4.4 mostra um exemplo de esquemático e *layout* para uma das primeiras placas construídas durante o desenvolvimento deste trabalho, onde podem ser vistos o símbolo e *footprint* para o *slot* de *Smart Card*, além de outros componentes.



(a) **(b)**

Figura 4.4: Esquemático (a) e *layout* (b) de uma das placas fabricadas, criados no EAGLE

Durante a elaboração das primeiras PCIs, o objetivo era criar um módulo leitor de cartão para Arduino semelhante a um leitor para computador, onde o cartão pudesse ser encaixado para leitura. A Figura 4.5 mostra as placas que foram fabricadas e desenvolvidas com a utilização do EAGLE.



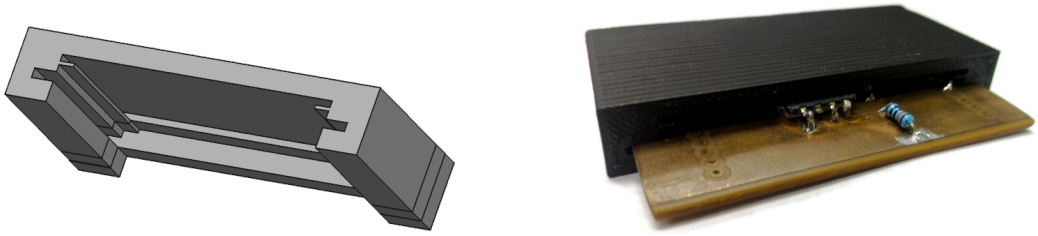
(a) **(b)** **(c)**

Figura 4.5: Primeiras PCIs fabricadas, em ordem cronológica da esquerda para a direita

A placa da Figura 4.5a foi feita para ser um módulo separado, conectado com fios ao Arduino, utilizando os pinos à direita da placa. O posicionamento do cartão no local correto do *slot* (para que os terminais do cartão fizessem contato com os terminais do leitor) foi feito com a instalação de parafusos em posições específicas da placa, marcadas no *software*. A abordagem da placa na Figura 4.5b foi um pouco diferente, pois ela foi fabricada para ser um *shield*, módulo que é encaixado em cima dos pinos do Arduino e que pode ser "empilhado" juntamente com outros módulos. Exemplos de *shields* incluem módulos de comunicação Wi-Fi, Ethernet e GSM (que poderiam ser usados em conjunto com o leitor, em uma aplicação de IoT), módulos de cartão SD, entre outros. O posicionamento do cartão nessa placa é feito de maneira semelhante à anterior.

A terceira placa, na Figura 4.5c, também foi criada para ser um *shield*, mas foi pos-

teriormente modificada para ser um módulo separado (pinos localizados acima da placa). As principais evoluções dessa placa com relação às anteriores foram a adição de um plano de terra (facilita o desenho do *layout*, diminui o tempo de corrosão do cobre, diminui o ruído e melhora a dissipação de calor), o corte mais preciso e regular da placa, e o posicionamento do cartão, que foi feito com impressão 3D. Com a utilização do FreeCAD, um *software* modelador 3D *open source*, foi desenhado um modelo de peça para encaixar na PCI. Nela, o cartão seria inserido de maneira alinhada com os terminais do leitor de uma forma muito mais precisa, tal qual um leitor para computador. A Figura 4.6 mostra o desenho do modelo e a peça impressa, encaixada na placa.



(a) **(b)**

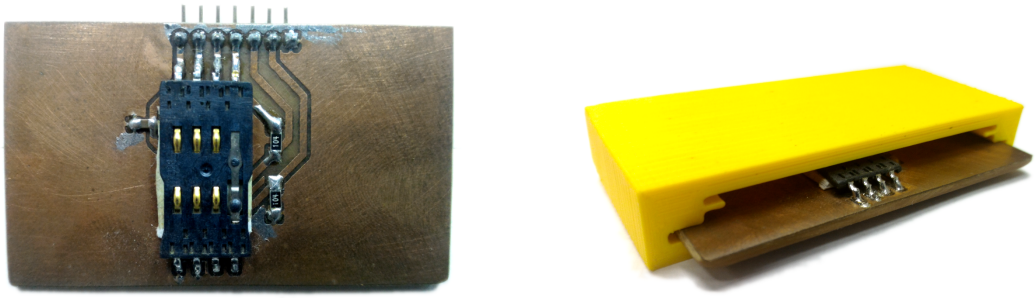
Figura 4.6: Modelo da peça de plástico, desenhado no FreeCAD (a) e placa da Figura 4.5c com a peça de plástico encaixada (b)

Após a fabricação das PCIs citadas, os desenhos das placas seguintes passaram a ser feitos no KiCAD, *software* de ECAD alternativo ao EAGLE. Apesar de ser amplamente adotado na comunidade de projetos eletrônicos, o EAGLE é um programa comercial com limitações na sua versão grátis, a qual estava sendo utilizada no projeto. O KiCAD, por sua vez, possui funções semelhantes ao EAGLE, é livre e *open source*, sem nenhuma limitação quanto às suas funcionalidades.

Foi fabricado, então, um segundo módulo de leitor com encaixe de plástico impresso, com desenho feito no KiCAD. Era interessante ter dois módulos funcionais para a realização de testes. A PCI, mostrada na Figura 4.7a, foi criada desde o início para ser um módulo de leitor de cartão avulso e utiliza *Surface-mount Technology* (SMT): componentes menores, também chamados de *Surface-mount Devices* (SMDs), que são soldados diretamente na superfície da placa, eliminando a necessidade de se fazer furos. O resultado foi uma placa mais compacta que a da Figura 4.5c.

4.1.2 Adição de comunicação sem fio e troca de Arduino por ESP8266

Para que o sistema fosse adequado para aplicações em IoT era necessário adicionar ao mesmo a capacidade de comunicação sem fio, o que possibilitaria ao dispositivo se comunicar remotamente com um servidor ou *gateway* e trocar informações. Foram consideradas duas possibilidades: os módulos nRF24L01+ (Nordic Semiconductor, 2008) e ESP8266 (Espressif, 2017), mostrados na Figura 4.8.



(a) **(b)**

Figura 4.7: Quarta placa fabricada, projetada no KiCAD (a) e leitor de cartão composto pela placa e a peça de plástico (b)

(a) **(b)**

Figura 4.8: Módulos nRF24L01+ (a) e ESP8266, modelo ESP-01 (b). *Fontes: dx.com e instructables.com*

O nRF24L01+ é um transceptor (i.e., transmissor e receptor em um único *chip*) de Rádiofrequência (RF) e de baixa potência que opera na banda ISM (*Industrial, Scientific and Medical*) de 2,4 GHz, espectro de frequência para desenvolvimento livre, sem a necessidade de licenciamento. O *chip* se conecta a um MCU (e.g., Arduino) através do barramento *Serial Peripheral Interface* (SPI) para configuração e operação. Este dispositivo é interessante para aplicação em projetos eletrônicos por sua simplicidade, preço e disponibilidade de recursos (como bibliotecas). No entanto, esses módulos não possuem protocolos de rede nem acessam a Internet, eles apenas criam um canal de conexão RF entre os rádios. A inclusão do nRF24L01+ no sistema proposto neste trabalho exigiria dois esforços adicionais: a implementação de protocolos de rede para os dispositivos e o desenvolvimento de um *gateway* conectado à Internet, o qual receberia as informações de todos rádios e as tornaria acessíveis a um dispositivo remoto (e.g., celular, *tablet*, computador). Portanto, a utilização do nRF24L01+ foi descartada.

O ESP8266 é um *System-on-Chip* (SoC) que integra Wi-Fi, um MCU de 32 *bits* com *clock* de 80 MHz e uma memória SRAM (*Static random-access memory*). Ele pode funcionar como adaptador Wi-Fi para um MCU rodando uma aplicação (e.g. Arduino) ou, alternativamente, funcionar como um dispositivo Wi-Fi *standalone* rodando sua própria aplicação. O CI não possui uma memória programável integrada, o que faz com que seja necessária uma memória *flash* externa (já presente em todos os módulos que são fabricados com esse *chip)*, de até 16 MB. O ESP8266 fornece ainda a capacidade de atualização *Over-the-air* (OTA) do *firmware* (i.e., gravar

uma nova aplicação na memória através da rede *wireless*) e três modos de economia de energia (*modem-sleep*, *light-sleep* e *deep-sleep*), úteis para prolongar o funcionamento de dispositivos alimentados por baterias. Diferentemente do nRF24L01+, o ESP8266 já possui protocolos de rede bem definidos (Padrão IEEE 802.11 e comunicação TCP/IP) e pode acessar a Internet, facilitando o desenvolvimento de aplicações para *Internet of Things* (IoT).

Inicialmente, o módulo ESP8266 foi escolhido para atuar apenas como um adaptador Wi-Fi para o Arduino, como sugerido no parágrafo anterior. De acordo com a Figura 4.9, o Arduino funcionaria como o MCU principal do sistema, recebendo as leituras de um ou mais sensores, cifrando-as com o SC e enviando os dados criptografados para o ESP8266, o qual enviaria os dados para um servidor remoto através da Internet.

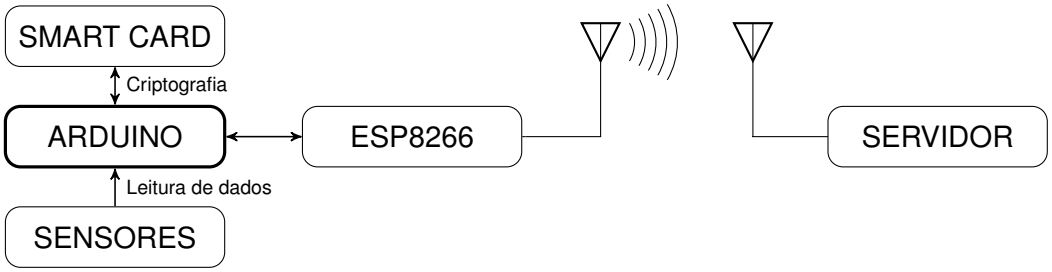


Figura 4.9: Diagrama inicial do sistema proposto, com módulo ESP8266 atuando apenas como adaptador Wi-Fi

Verificou-se, no entanto, que a utilização do módulo ESP8266 *standalone* ao invés do Arduino UNO traria algumas vantagens, entre elas uma maior capacidade de memória e processamento em um formato muito mais compacto e mais compatível com IoT. Alguns módulos ESP8266 diferentes daquele da Figura 4.8b, como os modelos ESP-201 e ESP-12E (Figura 4.10), dão ao usuário acesso a uma quantidade muito maior de pinos de entrada/saída, também chamados de *General-purpose Inputs/Outputs* (GPIOs), e interfaces de comunicação, os quais poderiam ser utilizados para conexão com o cartão, sensores e outros dispositivos. A Tabela 4.2 faz uma comparação de memória, velocidade de *clock*, quantidade de GPIOs e tamanho, entre o Arduino UNO e os módulos ESP8266 mencionados no texto. O Arduino UNO supera os módulos ESP8266 apenas na quantidade de GPIOs disponíveis para uso e fica bem atrás em todos os outros quesitos. Apesar disso, a quantidade de GPIOs presentes nos módulos da Figura 4.10 seria mais do que suficiente para o funcionamento do sistema.

Há ainda outro ponto importante, que tornaria mais simples a transição do Arduino UNO para o módulo ESP8266: a programação do módulo poderia ser feita no próprio Arduino IDE e com a mesma linguagem, ou seja, o que já tinha sido feito até então para o Arduino UNO poderia ser aproveitado com pouca ou nenhuma alteração. A diferença é que o Arduino já vem pronto para ser programado e possui uma porta USB para conexão com o computador, enquanto que os módulos ESP8266 precisam de uma placa auxiliar, com um botão de *reset*, um *switch* para configurar o módulo no modo de gravação ou de execução do programa e uma porta USB ou adaptador USB/Serial para permitir a conexão com o computador.

MÓDULO	GPIOs	MEMÓRIA (kB)			CLOCK (MHz)	ÁREA (cm^3)
		FLASH	EEPROM	RAM		
Arduino UNO	20	32	1	2	16	3,66
ESP-01	4	1024	4	80	80	0,35
ESP-201	17	1024	4	80	80	0,87
ESP-12E	17	4096	4	80	80	0,38

Tabela 4.2: Comparação entre Arduino UNO e módulos ESP8266

(a) **(b)**

Figura 4.10: Módulos ESP8266, modelo ESP-201 (a) e modelo ESP-12E (b). *Fontes: dx.com e alibaba.com*

O módulo ESP-201 (Figura 4.10a) foi o primeiro escolhido para tomar o lugar do Arduino UNO no sistema. A placa auxiliar fabricada para possibilitar a gravação do módulo é mostrada na Figura 4.11. Com isso, o diagrama do sistema foi atualizado, pois o Arduino UNO não faz mais parte dele e o ESP8266 atua agora como controlador principal, trocando informações diretamente com o SC e os sensores, como pode ser visto na Figura 4.12.

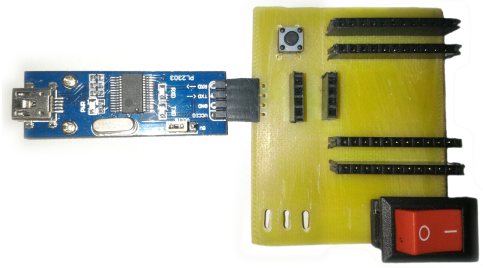


Figura 4.11: Placa utilizada para gravação do módulo ESP-201, com o adaptador USB/Serial conectado (à esquerda)

Para utilizar o módulo ESP8266 no lugar do Arduino foi necessário realizar algumas modificações no sistema. Como mencionado anteriormente, o sinal de relógio utilizado na comunicação com o SC era gerado pelo próprio Arduino, utilizando o *timer*. Para o módulo ESP8266, não foram encontradas informações sobre como utilizar o *timer* (ou mesmo se ele existe), tornando impossível a geração do sinal de relógio no próprio módulo. A solução encontrada foi a criação de um circuito de relógio externo, controlado por uma GPIO do ESP-

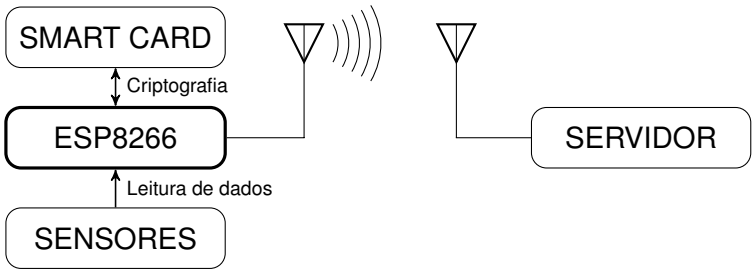


Figura 4.12: Diagrama atualizado do sistema proposto, com módulo ESP8266 atuando como microcontrolador principal do sistema

201. Foi utilizado, a princípio, o Oscilador Pierce, mostrado na Figura 4.13, por ser um circuito simples (apenas seis componentes) e de baixo custo. Com isso, foi possível utilizar um sinal de relógio de 4 MHz, quatro vezes mais rápido do que era possível com o Arduino, e o sistema se tornou mais flexível por não exigir uma conexão com um pino específico para o CLK do cartão.

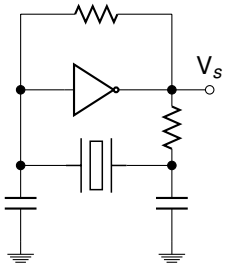


Figura 4.13: Oscilador Pierce

Uma segunda alteração necessária para o funcionamento do sistema com o módulo ESP8266 está associada às tensões de funcionamento do módulo e do SC. O ESP8266 precisa ser alimentado com 3,3 V e essa é a tensão de entrada/saída em qualquer um dos seus pinos. O cartão, por sua vez, funciona com 5 V (como visto na Tabela 4.1), o que impede que ele seja ligado diretamente ao ESP8266 da mesma forma que o Arduino. Semelhantemente, quaisquer outros dispositivos conectados ao módulo ESP8266 (e.g. sensores) também devem funcionar com 3,3 V, mas nem sempre é possível assegurar essa compatibilidade. Para garantir que qualquer cartão e dispositivo pudesse ser utilizado no sistema, foram adicionados dois *chips* conversores de nível de tensão: um para a conexão com o cartão e outro para quatro GPIOs livres do módulo, que poderiam ser usadas individualmente ou como interface SPI. A Figura 4.14 e a Figura 4.15 ilustram o funcionamento dos *chips* mencionados, mostrando os níveis de tensão que podem ser aplicados em cada lado. A adição dos *chips* aumentou a versatilidade do sistema, que agora poderia ser utilizado com qualquer classe de SC e com sensores de 3,3 V ou 5 V.

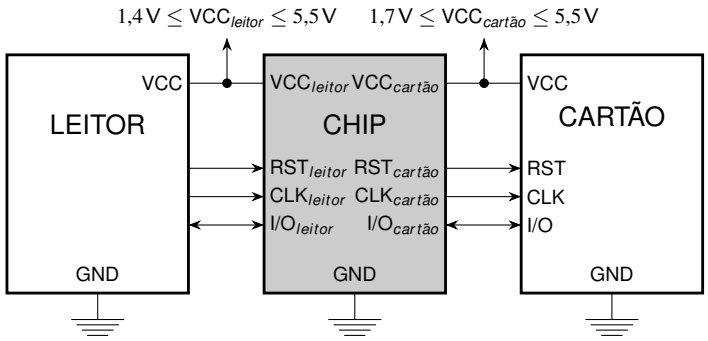


Figura 4.14: Funcionamento do *chip* conversor de tensão para o cartão

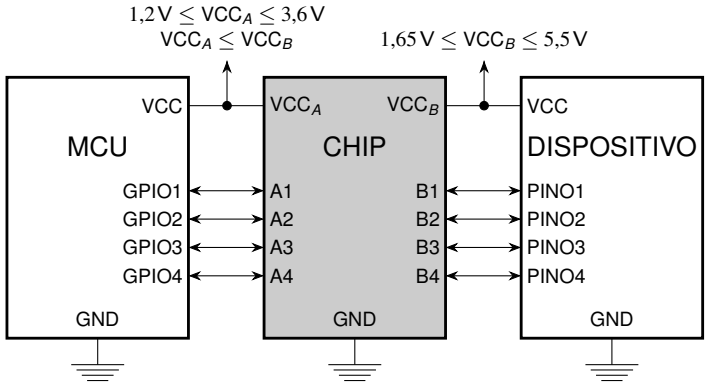


Figura 4.15: Funcionamento do *chip* conversor de tensão para as GPIOs

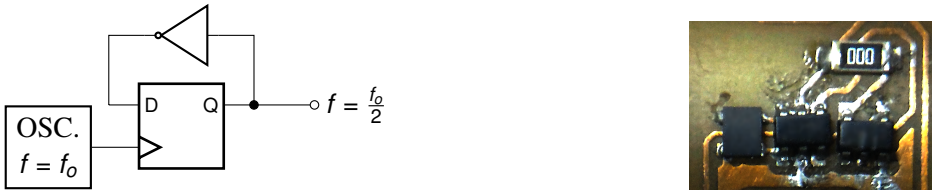
Os testes iniciais sem o Arduino foram feitos com as placas das Figuras 4.7b e 4.11 conectadas com fios às placas avulsas dos *chips* conversores (Figura 4.16a) e ao circuito de relógio externo, montado em uma *protoboard* (Figura 4.16b, onde a placa é o inversor e o encapsulamento metálico é o cristal piezoelétrico de 4 MHz).

(a) **(b)**

Figura 4.16: Placa individual de um dos *chips* conversores de tensão (a) e circuito de relógio externo, montado na *protoboard* (b)

Com o funcionamento do novo sistema verificado, o próximo passo no projeto de *hardware* era fazer uma PCI que integrasse todos os módulos em um único lugar, criando um dispositivo *wireless* portátil, com leitor de SC integrado para comunicação segura, sendo, portanto, adequado para aplicações em IoT. A primeira placa do projeto com ESP8266 foi

desenhada no KiCAD e apresentou algumas modificações com relação à montagem de testes. A primeira delas foi no leitor de cartões. Foi utilizado o *slot* da Figura 4.3 sem nenhuma adaptação e o cartão foi cortado com o tamanho ID-000 para encaixar no *slot*, com o objetivo de manter o dispositivo compacto. Outra modificação foi realizada no circuito de relógio externo. Como pode ser visto na Figura 4.16b, o cristal piezoelétrico é um dispositivo que ocupa muito espaço, além de ser *through-hole* (i.e., a placa precisa ser furada para soldar o componente). Para contornar esse problema, foi utilizado um CI oscilador SMD de 8 MHz e 2,5 mm x 2,0 mm em conjunto com um circuito divisor de frequência digital, mostrado na Figura 4.17.



(a) **(b)**

Figura 4.17: Circuito divisor de frequência digital (a) e detalhe do relógio externo na placa (b), onde podem ser vistos os chips: oscilador, *flip-flop* e inversor, da esquerda para a direita

A placa possui as conexões e *switches* necessários para gravar a aplicação no módulo ESP-201 e é ligada através da conexão com um adaptador USB/Serial, da mesma forma que a placa da Figura 4.11. Ela pode ser alimentada separadamente com 5 V e 3,3 V, ou utilizar apenas os 5 V do adaptador e obter os 3,3 V de um regulador de tensão. A Figura 4.18 mostra a frente e o verso da primeira placa fabricada com o módulo ESP8266. É possível ver na Figura 4.18a o módulo encaixado, o *slot* de SC (sem cartão), os pinos para as GPIOs livres (acima, do lado direito) e a entrada para o adaptador USB/Serial (abaixo, do lado direito). Já na Figura 4.18b estão as indicações de cada terminal e os *switches* necessários para gravação do módulo. Pode-se observar que a placa só possui uma face de cobre na qual os componentes estão distribuídos, o que acaba aumentando as dimensões do dispositivo, que são de aproximadamente 7,8 cm x 6,2 cm.

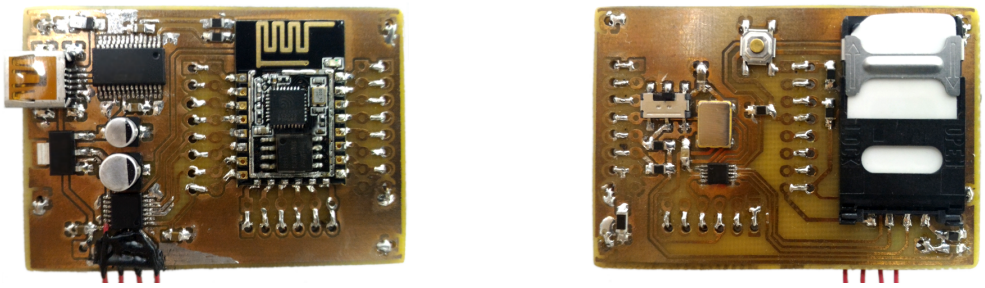
(a) **(b)**

Figura 4.18: Primeira placa com o módulo ESP8266, frente (a) e verso (b)

4.1.3 Placa Final

Foi desenvolvida uma segunda placa com o módulo ESP8266, mostrada na Figura 4.19, tendo em vista as alterações necessárias para tornar o dispositivo o mais adequado possível para aplicações em IoT. Esta foi também a placa final para este trabalho. Ela possui várias melhorias com relação à placa anterior:

- O módulo ESP8266 utilizado foi o ESP-12E (Figura 4.10b), que é menor, possui a mesma quantidade de GPIOs e é SMD;
- Todos os componentes utilizados foram também SMD, incluindo os *switches* para programação do módulo;
- O circuito de relógio foi substituído por um único CI oscilador de 4 MHz;
- A placa possui ambas as faces com cobre (também chamada de placa dupla-face), o que permite que os componentes possam ser distribuídos na frente e no verso, diminuindo as dimensões da placa para aproximadamente 5,6 cm x 3,9 cm (45% da área da placa anterior);
- Adição de uma porta mini USB à placa, dispensando a utilização de adaptadores externos para energizá-la e conectá-la com o computador.



(a) **(b)**

Figura 4.19: Segunda placa com o módulo ESP8266 e placa final do trabalho, frente (a) e verso (b)

Após todas as modificações, o diagrama final do *hardware* do sistema ficou como mostrado na Figura 4.20.

4.2 Arquitetura de Software

O *software* do sistema é composto por três programas distintos: um instalado no SC, um no MCU do dispositivo e um no computador, que atua como servidor do sistema. O desenvolvimento de cada programa é detalhado nos parágrafos seguintes.

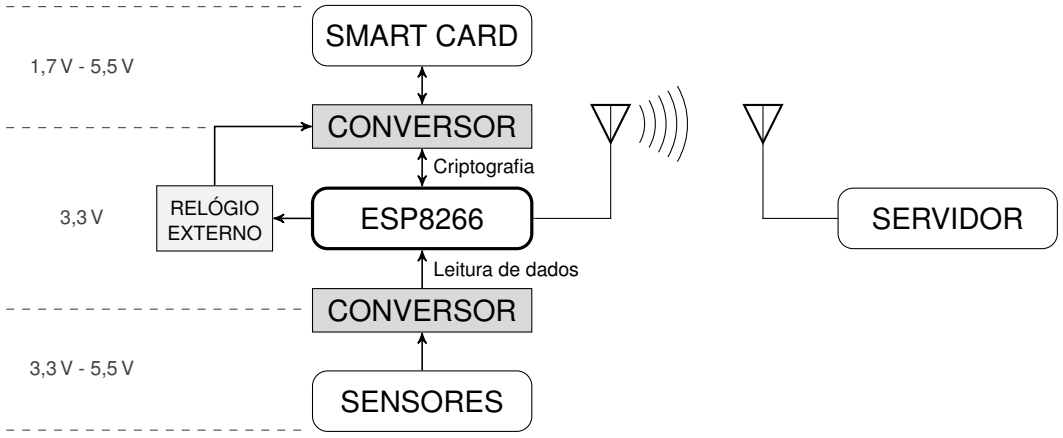


Figura 4.20: Diagrama final do *hardware* do sistema proposto

4.2.1 Protocolo de Comunicação entre Smart Card e Microcontrolador

O passo inicial no desenvolvimento do *software* do sistema era estabelecer a troca de informações entre SC e MCU. A biblioteca *open source* ArduinoSCLib (BARGSTEDT, 2016), desenvolvida para Arduino e placas compatíveis com Arduino, foi utilizada como a base do projeto de *software* do MCU. Ela implementa os procedimentos de operação do SC descritos na Subseção 2.3.2, incluindo a troca de APDUs, sendo portanto a camada mais básica da comunicação entre o microcontrolador e o cartão. No entanto, apesar de possibilitar a troca de mensagens entre o cartão e o MCU, testes iniciais de comunicação utilizando a ArduinoSCLib e o Arduino, monitorados com o auxílio de um analisador lógico, mostraram uma falha na implementação. O terceiro caso da Tabela 2.1 não era tratado, de forma que muitos comandos não eram executados corretamente, como é ilustrado na Figura 4.21.

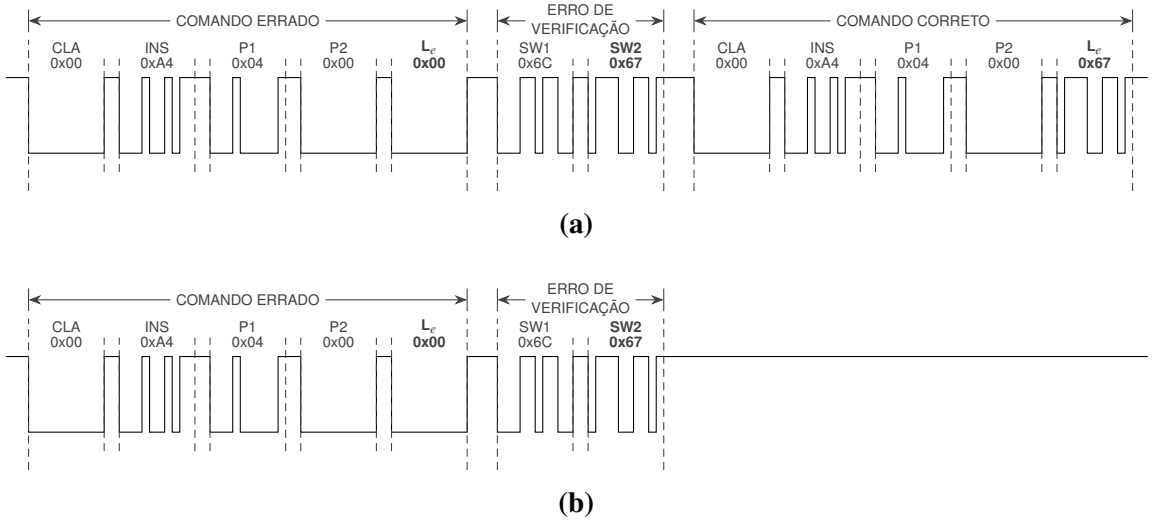


Figura 4.21: Funcionamento esperado (a) e implementação da biblioteca ArduinoSCLib (b) durante envio de comandos para o *Smart Card*

Uma inspeção posterior do código-fonte também revelou a necessidade de correções na rotina de envio e recebimento de APDUs, para garantir uma maior conformidade com o

protocolo T=0 especificado na ISO7816-3 e convenientemente representado em uma máquina de estados no *Smart Card Handbook* (RANKL; EFFING, 2010).

A migração do Arduino UNO para o ESP8266, descrita na Seção 4.1, também exigiu algumas alterações no código. Após a substituição das placas, a execução do *software* no ESP8266 era frequentemente interrompida por falhas que causavam a reinicialização do sistema. Após pesquisa, descobriu-se que o *Watchdog Timer* (WDT) era o causador dessas falhas. Ele é um módulo presente na maioria dos MCUs, responsável por garantir que o sistema funcione corretamente. Geralmente, ele possui um *timer* que é constantemente reiniciado pelo sistema, para mostrar que a execução está acontecendo de forma correta. Quando, por exemplo, a execução do programa permanece durante muito tempo dentro de um laço da programação (e.g., *while*, *for*) e esse *timer* não é reiniciado, o WDT é ativado e reinicia o sistema. Como o ESP8266 executa constantemente tarefas secundárias relativas ao protocolo de comunicação Wi-Fi, uma interrupção longa dessas tarefas por causa de um laço é interpretada pelo WDT como mal funcionamento do sistema. A solução para esse problema foi a adição de *delays* ao longo do código, para permitir ao MCU a realização das tarefas em segundo plano mesmo dentro de um laço.

4.2.2 Algoritmos Suportados pelo Smart Card

Estabelecida a troca de mensagens entre o SC e o MCU, era necessário encontrar uma forma de utilizar as capacidades criptográficas do cartão para, em conjunto com o microcontrolador, empregá-las na criação de um sistema de comunicação segura para IoT. O cartão usado no projeto possui a tecnologia Java Card descrita na Subseção 2.3.3, o que significa que nele podem ser instalados programas desenvolvidos em Java.

A presença de uma determinada funcionalidade no cartão depende da versão do Java Card presente nele (e.g., o algoritmo de *hash* SHA256 só foi introduzido na versão 2.2.2) e das próprias limitações de *hardware* e *software* do SC, o que significa que nem toda funcionalidade de uma dada versão da *Application Programming Interface* (API) Java Card estará contida no cartão. As funcionalidades do cartão mais interessantes para este trabalho são os algoritmos criptográficos que ele é capaz de executar. A maneira utilizada no projeto para descobrir quais algoritmos de segurança eram suportados pelo cartão foi através do *software* JCAlgTest (ŠVENDA, 2016). Composto por um *applet* que é instalado no cartão e dois programas que são executados no computador (um para comunicação com o SC e outro para tratamento dos resultados), o JCAlgTest realiza testes automáticos de algoritmos suportados e testes de performance de execução para cada algoritmo, gerando tabelas e gráficos para melhor visualização dos resultados.

A Tabela 4.3 mostra uma parte dos resultados do JCAlgTest, especificamente para algoritmos de *hash*, onde é possível observar que, mesmo possuindo o Java Card na versão 2.2.2, o cartão não dá suporte a algoritmos introduzidos nessa mesma versão (SHA384 e SHA512). As tabelas com todos os algoritmos suportados pelo cartão encontram-se no Apêndice A. A Figura 4.22, por sua vez, mostra um dos gráficos obtidos com os testes de performance na

execução do algoritmo de *hash* SHA256, com diferentes quantidades de dados de entrada. Os testes de performance são úteis na comparação entre diferentes SCs e também na visualização do comportamento dos algoritmos com a variação da quantidade de dados.

ALGORITMO	JAVA CARD	SUPORTADO
SHA	$\leq$ 2.1	SIM
MD5	$\leq$ 2.1	SIM
RIPEMD160	$\leq$ 2.1	SIM
SHA256	2.2.2	SIM
SHA384	2.2.2	NÃO
SHA512	2.2.2	NÃO
SHA224	3.0.1	NÃO

Tabela 4.3: Tabela de suporte a algoritmos gerada pelo JCAlgTest, para algoritmos de *hash*

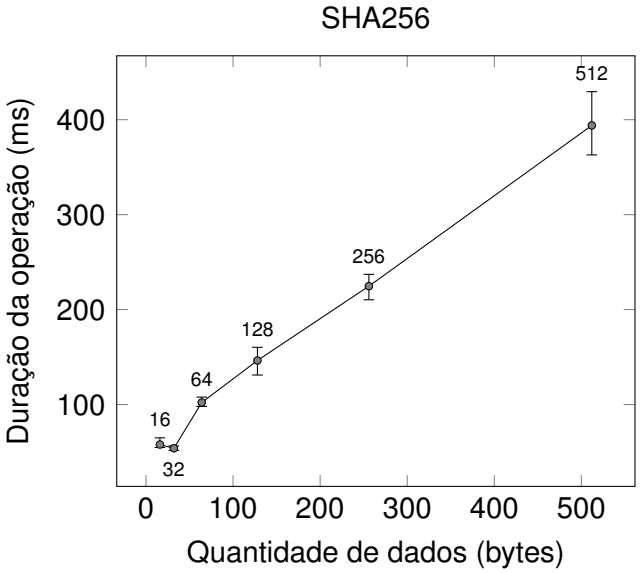


Figura 4.22: Gráfico de performance para execução do algoritmo de *hash* SHA256 com o *Smart Card*

4.2.3 Applets para Smart Cards

Existem diversos *applets open source* em desenvolvimento para SCs. Vários deles estão concentrados no repositório *Applet Playground* (PALJAK, 2017). Alguns deles são específicos para aplicações financeiras (e.g., OpenEMV, SatoChipApplet, Ledger Unplugged), outros para aplicações em documentos de identificação (e.g., eID Applet, GIDS Applet, PLAID, Passport Applet), outros implementam transmissão de dados para NFC (protocolo NDEF) e o restante apresenta soluções voltadas para segurança da informação em geral. A Tabela 4.4 lista os *applets*

tidos como mais relevantes para o desenvolvimento do projeto, as datas em que foram atualizados e em qual padrão/especificação eles são baseados. Esses *applets* foram considerados para serem a base do *software* que seria instalado no SC.

Tendo em vista que o objetivo do sistema é realizar uma comunicação de forma segura e autenticada entre o dispositivo IoT e o servidor, foi criada a Tabela 4.5, a qual lista as capacidades de criptografia do SC consideradas mais importantes para o alcance do objetivo citado. A tabela também associa um código abreviado a cada capacidade. Após inspeção dos códigos-fonte de cada *applet* listado na Tabela 4.4, foi possível determinar quais capacidades da Tabela 4.5 foram implementadas em cada um deles. A Tabela 4.6 faz uma comparação entre os *applets* e suas capacidades.

NOME	**DATA**	**ESPECIFICAÇÃO**
MUSCLE Applet	mar/2012	MUSCLE Cryptographic Card Edge Definition
CoolKey Applet	jan/2017	MUSCLE Cryptographic Card Edge Definition
SatoChipApplet	dez/2015	MUSCLE Cryptographic Card Edge Definition
IsoApplet	feb/2017	ISO7816
JC PKI Applet	jan/2011	ISO7816
YKNEO OpenPGP	jan/2017	OpenPGP application on ISO Smart Card OSs
OpenPGP-Card	abr/2015	OpenPGP application on ISO Smart Card OSs

Tabela 4.4: *Applets* de segurança para *Smart Cards*

CAPACIDADES CRIPTOGRÁFICAS	CÓDIGO
Proteção com PIN ou senha	PIN
Geração de chaves assimétricas	GER/A
Cifragem com criptografia assimétrica	CIF/A
Decifragem com criptografia assimétrica	DEC/A
Geração de chaves simétricas	GER/S
Cifragem com criptografia simétrica	CIF/S
Decifragem com criptografia simétrica	DEC/S
Importação de chaves	IMP
Assinatura digital	SIG
Verificação de assinatura digital	VER
Geração de números aleatórios	RND
Armazenamento de objetos	OBJ
Hash de mensagens	HASH
Código de autenticação de mensagem	MAC
Troca de chaves	KEX
Criptografia de Curvas Elípticas	CCE

Tabela 4.5: Capacidades criptográficas importantes na implementação do sistema

APPLET	CAPACIDADES CRIPTOGRÁFICAS															
	PIN	GER/A	CIF/A	DEC/A	GER/S	CIF/S	DEC/S	IMP	SIG	VER	RND	OBJ	HASH	MAC	KEX	CCE
MUSCLE Applet	●	●	●	●		●	●	●	●	●	●	●				
CoolKey Applet	●	●	●	●				●			●	●				
SatoChipApplet	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
IsoApplet	●	●		●				●	●		●	●				●
JC PKI Applet	●	●		●				●	●		●	●				
YKNEO OpenPGP	●	●		●				●	●		●	●				
OpenPGP-Card	●	●		●				●	●		●	●				

Tabela 4.6: Comparação entre *applets* de segurança

É possível observar que todos os *applets* dão suporte a proteção com PIN, geração de chaves assimétricas (a maioria apenas suporta RSA, somente dois também dão suporte a curvas elípticas), decifragem com criptografia assimétrica e importação de chaves externas para o cartão, geração de números aleatórios e armazenamento de objetos na memória do SC. A maioria ainda suporta assinatura digital. No entanto, poucos *applets* implementam capacidades fundamentais, como: criptografia de chave simétrica (geração de chave, cifragem e decifragem), muito importante na comunicação sigilosa por ser mais rápida; verificação de assinaturas, essencial para autenticação com certificados; códigos de autenticação de mensagem, que garantem a integridade dos dados transmitidos; algoritmos de troca de chaves, para estabelecer uma chave secreta entre duas partes; algoritmos de criptografia com curvas elípticas, que oferecem segurança comparável ao RSA com chaves muito menores.

O SatoChipApplet é o *applet* com mais capacidades implementadas na lista, sendo portanto uma aparente escolha óbvia para o projeto. No entanto, este *applet* é focado em aplicações com *bitcoins* e possui várias funções que fogem do escopo deste trabalho. Além disso, e o mais importante, ele é baseado na especificação MUSCLE Card Edge (CORCORAN; CUCINOTTA, 2001)(CUCINOTTA; NATALE; CORCORAN, 2003), assim como o MUSCLE Applet (que é o segundo *applet* com mais capacidades na tabela) e o CoolKey Applet. Essa especificação foi criada independentemente da ISO7816, portanto não possui nenhuma compatibilidade com o padrão adotado pelo mercado. Além do mais, por ser uma especificação antiga, ela não leva em consideração algoritmos mais recentes de criptografia. Por este motivo, os três *applets* citados foram desconsiderados.

Existem dois *applets* baseados na especificação do OpenPGP para *Smart Cards* (PIETIG, 2015): o YKNEO Applet (utilizado comercialmente em dispositivos da empresa Yubico) e o OpenPGP Card. Essa especificação é construída sobre a ISO7816, porém ela limita o que pode ser feito com o cartão, como a quantidade de chaves que ele possui e os tipos de operação criptográficas que pode realizar, diminuindo a versatilidade das aplicações que podem ser desenvolvidas a partir dela. Sendo assim, estes *applets* também foram descartados, restando apenas o IsoApplet e o JC PKI Applet. Este último, apesar de ter praticamente as mesmas capacidades que o IsoApplet, não implementa criptografia de curvas elípticas e não é atualizado há alguns anos. Por isso, foi escolhido o IsoApplet para ser instalado no cartão.

4.2.4 Modificações no IsoApplet

Como visto na Tabela 4.6, o IsoApplet não implementa todas as funcionalidades consideradas relevantes para o projeto. A Tabela 4.7 especifica os algoritmos que estão implementados no IsoApplet original.

Durante o desenvolvimento do *software*, foram realizadas modificações no *applet*, para adicionar novas funções necessárias para o sistema. Ao final do trabalho, as funcionalidades do IsoApplet haviam sido expandidas para aquelas que estão na Tabela 4.8. Todas essas funções foram implementadas utilizando a API Java Card com exceção das funções de MAC, as quais

CAPACIDADES	ALGORITMOS		
GER/A	NOME	BITS	
	RSA	2048	
	Curvas Elípticas	192/224/256	
DEC/A	NOME	PADDING	
	RSA	PKCS#1	
IMP	NOME	BITS	
	RSA	2048	
	Curvas Elípticas	192/224/256	
SIG	NOME	PADDING	HASH
	RSA	PKCS#1	N/A
	ECDSA	N/A	SHA

Tabela 4.7: Algoritmos implementados no IsoApplet original

foram implementadas manualmente a partir da especificação do HMAC encontrada no RFC 2104 (KRAWCZYK; BELLARE; CANETTI, 1997). Como pode ser visto na Tabela A.1, a API implementada no cartão só suporta algoritmos de MAC baseados em algoritmos de cifragem (CMACs), embora algoritmos de HMAC tenham sido introduzidos na versão 2.2.2 do Java Card (Sun Microsystems, 2006a). Como o protocolo TLS utiliza HMACs, foi necessário fazer uma implementação manual.

Outras funções relacionadas ao TLS não listadas na Tabela 4.8 também foram implementadas manualmente no IsoApplet. Visto que vários cálculos realizados durante o *handshake* (Subseção 2.4.2) dependem da função PRF, ela foi a primeira a ser implementada, a partir da função de HMAC citada. Posteriormente, foram implementados também no IsoApplet comandos para o cálculo dos seguintes valores utilizados no TLS: `pre_master_secret`, `master_secret`, `key_block` e `verify_data`. Com exeção do último valor, que precisa ser enviado para o servidor ao final do *handshake*, todos os outros valores são calculados dentro do *Smart Card* e lá permanecem armazenados de forma segura na memória não volátil do cartão, de tal maneira que nem o próprio dispositivo IoT tem acesso à informação. Como esses valores estão intimamente relacionados aos mecanismos de autenticação do protocolo de *handshake* e também à confidencialidade, autenticação e integridade das mensagens que serão trocadas após o *handshake*, é imprescindível que eles não sejam expostos em um ambiente inseguro. As chaves simétricas derivadas do `key_block` também são automaticamente criadas dentro do cartão. O dispositivo IoT envia ao SC os comandos de cifragem, decifragem e criação de MACs que utilizam essas chaves, mas nunca tem acesso aos seus valores.

CAPACIDADES	ALGORITMOS		
GER/A	NOME	BITS	
	RSA	2048	
	Curvas Elípticas	192/224/256	
CIF/A	NOME	PADDING	
DEC/A	RSA	PKCS#1	
GER/S	NOME	BITS	
	AES	128	
	3DES	192	
CIF/S	NOME	MODO	PADDING
DEC/S	AES	CBC	N/A
	3DES	CBC	N/A
IMP	NOME	BITS	
	RSA	2048	
	Curvas Elípticas	192/224/256	
	AES	128	
	3DES	192	
SIG	NOME	PADDING	HASH
VER	RSA	PKCS#1	N/A
	RSA	PKCS#1	SHA
	ECDSA	N/A	SHA
HASH	NOME		
	SHA		
	SHA256		
MAC	NOME		
	HMAC-SHA		
	HMAC-SHA256		
KEX	NOME		
	DH com Curvas Elípticas		

Tabela 4.8: Algoritmos implementados no IsoApplet após modificações

4.2.5 Bibliotecas desenvolvidas para o Microcontrolador

A elaboração do *software* que seria instalado no MCU, *WifiClient*, exigiu a criação de bibliotecas de código específicas, onde estariam compiladas as funções e constantes necessárias para construir a aplicação principal. Foram criadas quatro bibliotecas: uma relacionada à manipulação de dados codificados em ASN.1, outra relativa à especificação GlobalPlatform, outra para comunicação com o *Smart Card* e outra para implementar o protocolo TLS. As bibliotecas e suas principais características são detalhadas nas seções subsequentes.

Biblioteca ASN.1Functions. Esta biblioteca possui funções auxiliares necessárias para a manipulação de dados codificados de acordo com as regras da ASN.1, descritas na Seção 2.6. Ela é necessária, pois informações com esse tipo de notação são extensivamente usadas nos padrões nos quais este trabalho é baseado. A biblioteca contém poucas funções, com o que é necessário para se "navegar" entre os diferentes objetos em uma estrutura ASN.1. Por exemplo:

- Busca de *tags* específicas em um conjunto de dados. Muitos valores possuem tags padronizadas e uma função que possa encontrá-las diretamente é útil;
- Decodificação de *tags* e campos de tamanho. Quando se está percorrendo um conjunto de dados ASN.1, é importante determinar o comprimento dos próprios campos de *tag* e tamanho e também o comprimento dos dados. Isso é feito através da decodificação;
- Funções para pular de um objeto para outro, ou de um objeto externo para um objeto interno, quando existem sequências de objetos.

Além das funções, também são declaradas algumas constantes com valores de *tags* mais comuns, como booleanos, inteiros, cadeias de caracteres e sequências.

Biblioteca GPSecure. Esta biblioteca implementa as funções necessárias para criar uma comunicação segura entre o cartão e o MCU através do *Secure Channel Protocol* e para realizar a instalação e desinstalação de *applets* no cartão, de acordo com os procedimentos e comandos determinados na especificação GlobalPlatform (GlobalPlatform, 2003). É uma biblioteca importante, pois permite ao servidor atualizar a aplicação no cartão através do envio de comandos para o MCU. A execução do SCP requer a realização de determinadas tarefas implementadas na biblioteca, como:

- Geração das *chaves de sessão* utilizadas na cifragem e autenticação das mensagens, através da cifragem de um conjunto de dados específico com o algoritmo *Triple DES* (3DES) em modo CBC;
- Criação de *criptogramas de autenticação*, valores calculados a partir da chave base instalada no cartão e de números aleatórios com o algoritmo de CMAC *Full Triple DES* (definido na ISO9797-1), os quais são utilizados pelo MCU e pelo cartão para se autenticarem mutualmente;

- Cálculo de um código MAC para cada APDU enviada durante a comunicação segura, com a utilização do algoritmo *Retail MAC* (também definido na ISO9797-1).

Todos os algoritmos citados acima utilizam como base o algoritmo de cifragem DES. Como a utilização do SCP é um passo necessário para instalar o *applet* no cartão, esses algoritmos devem ser executados pelo próprio MCU. Para isso, foi utilizada a biblioteca *open source* ArduinoDES (RIEMANN, 2015), a qual implementa o DES para Arduino e placas compatíveis. É importante ressaltar que esta é a única situação no sistema na qual algoritmos de criptografia *não* são executados pelo *Smart Card*.

A instalação de *applets* no cartão, por sua vez, requer a abertura de um canal seguro com o SCP e o envio de APDUs de comando para carregamento do código do *applet* e subsequente instalação. A biblioteca GPSecure implementa a instalação do *applet* de duas maneiras distintas:

1. Em uma única função, a qual pode ser utilizada quando toda a informação do *applet* a ser instalado está disponível para o MCU. Por exemplo, quando o código que é instalado no MCU já possui os bytes de instalação do *applet* em uma variável;
2. Em três funções diferentes: uma para iniciar a instalação, outra para enviar os blocos de bytes de instalação do *applet* a serem carregados no cartão e outra para finalizar a instalação. Essa função é útil quando o MCU recebe os bytes de instalação do *applet* aos poucos, através da porta serial ou da rede sem fio (enviados pelo servidor, por exemplo).

Os bytes de instalação do *applet* são obtidos a partir do arquivo CAP, gerado na compilação e conversão do código Java (Subseção 2.3.3) e definido na especificação da JCVM (Sun Microsystems, 2006c). Este arquivo é, na verdade, um contêiner para componentes que também são arquivos CAP, com diferentes informações sobre o *applet* a ser instalado. No total, são onze componentes: Header.cap, Directory.cap, Applet.cap, Import.cap, ConstantPool.cap, Class.cap, Method.cap, StaticField.cap, ReferenceLocation.cap, Export.cap e Descriptor.cap. O conteúdo de cada componente está fora do escopo deste trabalho. Durante a instalação, esses componentes devem ser extraídos do arquivo CAP principal e enviados para o cartão.

A descoberta dos APDUs de comando que devem ser enviados durante a instalação, a forma de extrair os componentes do arquivo CAP principal e a sua ordem de envio foi feita de uma maneira mais prática, a partir da observação do funcionamento do *software* GlobalPlatformPro (PALJAK, 2016). Ele é desenvolvido em Java e possui uma série de comandos que permitem a interação entre um computador e um *Smart Card* compatível com a especificação GlobalPlatform. Entre esses comandos, estão: envio de APDUs (com ou sem SCP), instalação e desinstalação de *applets*, listagem de *applets* no cartão, alteração da chave base do cartão. Todos esses comandos podem ser executados com as opções *verbose* e *debug*. A primeira aumenta a quantidade de mensagens impressas pelo programa sobre o que está sendo feito e a segunda habilita a impressão dos APDUs que estão sendo trocados entre o computador e o cartão. Com a observação dessas

informações foi possível implementar as funções da biblioteca GPSecure e um *script* em Python, *CapFileParser*, o qual extrai os bytes necessários do arquivo CAP e os exporta para arquivos que podem ser usados tanto pelo dispositivo IoT quanto pelo servidor para fazer uma atualização no *software* do cartão.

Biblioteca SmartCarduino. Está biblioteca é o núcleo do projeto de *software* do MCU. Nela estão implementadas as funções e constantes relativas à criação e administração de uma estrutura de arquivos PKCS#15 no cartão e também à requisição de operações criptográficas do cartão.

A estrutura de arquivos PKCS#15 (Seção 2.5) é o que dá ao *Smart Card* a capacidade de armazenamento sistematizado de objetos como chaves e certificados. O IsoApplet possibilita a criação dessa estrutura no cartão a partir dos comandos enviados pelo MCU. As funções da biblioteca SmartCarduino relacionadas ao padrão PKCS#15 incluem:

- Criação da própria estrutura PKCS#15 no cartão, que envolve a criação dos DFs e EFs obrigatórios e de um PIN para controle de acesso;
- Criação das *File Control Informations* (FCIs), propriedades de um determinado arquivo que incluem seu número de identificação, tamanho e condições de acesso. Todo arquivo que é criado na estrutura PKCS#15 possui um FCI;
- Criação dos *Control Reference Templates* (CRTs), conjuntos de informações enviados ao cartão sempre que uma operação criptográfica é solicitada, através do comando `MANAGE SECURITY ENVIRONMENT`. Contém informações sobre o algoritmo a ser utilizado e, se necessário, uma referência de uma chave (pública, privada ou secreta) para ser usada com o algoritmo em questão;
- Criação dos diferentes objetos que são armazenados nos EFs.

As funções restantes da biblioteca SmartCarduino são associadas às operações criptográficas que o cartão pode executar. Existem funções para a requisição de cada capacidade listada na Tabela 4.8, onde os algoritmos e chaves são especificados nos CRTs. Em geral, essas requisições têm uma construção semelhante: primeiramente, é selecionado o IsoApplet instalado no cartão com o comando `SELECT` (seria o equivalente a rodar a aplicação); depois, é enviado um comando de verificação do PIN (`VERIFY`) para que o cartão permita que a operação seja executada; é enviado, então, o comando `MANAGE SECURITY ENVIRONMENT` com o CRT relativo à operação criptográfica; por fim, é enviado o comando `PERFORM SECURITY OPERATION` com os parâmetros que indicam ao cartão qual operação será executada. A depender da operação, o cartão pode responder com um bloco de dados (e.g., cifragem, decifragem, *hash*, assinatura digital) ou apenas os bytes SW1 e SW2 informando se a operação foi realizada corretamente ou não (e.g., verificação de assinatura).

Uma exceção à essa estrutura de comandos é a operação de geração de chaves assimétricas, na qual é enviado um comando específico, `GENERATE ASYMMETRIC KEYPAIR`, que é respondido pelo cartão com a chave pública que foi gerada. A chave privada, no entanto, nunca

é revelada para o MCU por motivos de segurança. As operações que, como essa, envolvem a utilização de chaves possuem alguns comandos a mais onde são verificados os EFs de chaves (SKDF, PrKDF, PuKDF) em busca das chaves com a mesma referência presente no CRT. Essa verificação garante que não serão criadas chaves novas com a mesma referência de chaves existentes ou que não serão realizadas operações com chaves não existentes no *Smart Card*. No caso da geração de chaves, os EFs citados ainda são atualizados com os objetos das novas chaves criadas, através do comando `UPDATE BINARY`.

O desenvolvimento dessa biblioteca, assim como no caso da GPSecure, foi feito a partir da observação do funcionamento de um *software* para computador: o *OpenSC* (OpenSC Team, 2015). Ele consiste em um conjunto de ferramentas *open source* para se trabalhar com as capacidades criptográficas dos *Smart Cards*. A partir da sua versão 0.15.0, o OpenSC passou a dar suporte a cartões com o IsoApplet instalado, o que possibilitou a realização de testes. As ferramentas do OpenSC utilizadas foram as seguintes:

- `opensc-explorer`: Permite a exploração dos arquivos na estrutura PKCS#15 do *Smart Card* de uma forma semelhante à exploração de arquivos no terminal do Linux ou no Prompt de Comando do Windows;
- `pkcs15-init`: Possui funções para criação de uma estrutura PKCS#15 no cartão, armazenamento de objetos como PINs, chaves e certificados e geração de chaves assimétricas;
- `pkcs15-tool`: Contém funções para leitura e listagem dos diferentes objetos armazenados no cartão;
- `pkcs15-crypt`: Possui funções para requisição de operações criptográficas do cartão. Especificamente, apenas suporta as operações de assinatura digital e decifragem.

Ao contrário do GlobalPlatformPro, o OpenSC não fornece opções que permitam a visualização dos APDUs trocados entre o computador e o cartão. Dessa forma, para conseguir observar essas informações, foi utilizado o *Wireshark*. Este *software* é um analisador de pacotes, geralmente usado para análise e manutenção de redes. Sua principal característica é a capacidade de capturar e registrar o tráfego de mensagens em uma interface do computador, como a porta Ethernet ou as portas USB.

A captura foi feita na porta USB na qual estava ligado um leitor de cartões. Apesar de o computador possuir várias portas USB, todas elas são administradas por uma mesma interface, a qual é monitorada pelo Wireshark. Por causa disso, todas as mensagens de todos os dispositivos que estão conectados a essas portas acabam sendo capturadas, o que gera mais informação que o necessário. Para resolver esse problema, o Wireshark permite que as mensagens sejam filtradas por um par de características da interface USB que é único para cada dispositivo conectado: *Endereço de Dipositivo* e *ID de Barramento*, os quais podem ser descobertos facilmente com um comando no terminal. Com isso, é possível visualizar apenas as mensagens relativas ao leitor de cartão. Essas mensagens são salvas em um arquivo *pcap*, próprio do Wireshark, que contém

todas as informações sobre o tráfego. Com a utilização do *Tshark*, uma versão do Wireshark para linha de comando, é possível criar um arquivo de texto mais simples, apenas com o conteúdo das mensagens e a direção de envio (transmitida ou recebida), o que facilita a análise. O resultado da captura pode ser visto no fragmento a seguir:

```
0 65:00:00:00:00:00:03:00:00:00
1 81:00:00:00:00:00:03:01:00:01
0 62:00:00:00:00:00:04:00:00:00
1 80:18:00:00:00:00:04:00:00:00:3B:FE:18:00:00:80:31:FE:45:80:31:
  80:66:40:90:91:06:2D:10:83:01:90:00:D3
0 6F:11:00:00:00:00:0B:00:00:00:00:A4:04:00:0C:F2:76:A2:88:BC:FB:
  A6:9D:34:F3:10:01
1 80:05:00:00:00:00:0B:00:00:00:00:06:06:90:00
0 6F:08:00:00:00:00:0D:00:00:00:00:A4:08:00:02:2F:00:00
1 80:19:00:00:00:00:0D:00:00:00:6F:15:81:02:00:80:82:01:01:83:02:
  2F:00:86:08:FF:00:00:00:00:00:00:00:90:00
```

Nesse fragmento, as mensagens que iniciam com “0” foram enviadas pelo computador e as que iniciam com “1” foram recebidas pelo computador, portanto enviadas pelo cartão. As partes das mensagens que estão na cor cinza fazem parte do protocolo USB, o qual está fora do escopo deste trabalho. A mensagem marrom, por sua vez, é o ATR enviado pelo cartão após sua ativação e *reset*, marcando o início da comunicação. As partes azuis são os cabeçalhos dos APDUs de comando e as partes laranjas são os rodapés dos APDUs de resposta. As partes roxas representam os corpos de ambos os tipos de APDU. A análise dessas capturas para diferentes comandos das ferramentas do OpenSC, juntamente com consultas ao código-fonte do IsoApplet e aos padrões ISO7816 possibilitaram o desenvolvimento da biblioteca SmartCarduino.

Biblioteca CardTLS. Esta biblioteca possui o que é necessário para implementar os Protocolos de Registro e *Handshake* do TLS (descritos na Seção 2.4) e, consequentemente, a comunicação segura e autenticada entre o dispositivo IoT e o servidor. Suas principais funções são:

- Criação de cada uma das mensagens enviadas pelo cliente no protocolo de *handshake*, o que envolve diversas requisições de operações criptográficas ao *Smart Card*, incluindo os cálculos relativos ao TLS mencionados na Subseção 4.2.4;
- Cifragem e cálculo de MAC de mensagens enviadas após o *handshake*, com as chaves que estão armazenadas no cartão (Protocolo de Registro);
- Funções utilizadas na pré-configuração do dispositivo IoT, como o recebimento dos bytes para gravação remota do cartão, a limpeza da memória EEPROM do MCU onde serão guardados os parâmetros da sessão, a geração e o envio das chaves assimétricas principais do dispositivo para o servidor e o subsequente recebimento dos certificados de cada chave assinados pelo servidor.

Além dessas funções, a biblioteca contém grande parte das constantes definidas para o TLS no RFC 5246 (ALLEN *et al.*, 2008). Também são definidos alguns valores constantes próprios da aplicação principal, como as referências, rótulos e identificadores das chaves e dos certificados do dispositivo e do servidor, os quais são fixos.

As principais referências no desenvolvimento dessa biblioteca foram os RFCs: 5246, que especifica o TLS; 4492 (BLAKE-WILSON *et al.*, 2006), que estabelece suites de criptografia para o TLS com uso de Curvas Elípticas; 5280 (COOPER *et al.*, 2008), que trata dos certificados X.509 utilizados na aplicação; 7366 (GUTMANN, 2014), que define uma extensão para o TLS onde o Protocolo de Registro é feito com a construção *encrypt-then-MAC*.

4.2.6 Código do Servidor e Funcionamento Geral do Sistema

O servidor do sistema é feito com um código em Python, *TCPServer*. Da mesma forma que a biblioteca CardTLS, ele também declara diversas constantes relativas ao TLS e define a função PRF, fundamental para a execução do algoritmo. A base deste código é o pacote *Cryptography* (Python Cryptographic Authority, 2017), que implementa todos os algoritmos de criptografia necessários. A comunicação entre cliente e servidor é feita através de um *socket* com o protocolo TCP, em uma rede Wi-Fi.

No início do programa, quando o cliente (dispositivo IoT) e o servidor iniciam a troca de mensagens, eles podem escolher entre continuar a sessão anterior, quando já foi realizado o *handshake*, ou iniciar uma nova sessão. No primeiro caso, o servidor lê os parâmetros de segurança de um arquivo salvo no computador e o cliente lê os parâmetros da sua memória EEPROM. No segundo caso, se o cliente está sendo ligado pela primeira vez, ele precisa ser pré-configurado. Por outro lado, se o cliente já foi pré-configurado, é realizado o protocolo de *handshake* e depois acontece a troca de mensagens da aplicação. Durante a pré-configuração do cliente, são realizados três passos:

1. Reinstalação do IsoApplet no *Smart Card* e criação de uma nova estrutura PKCS#15. O cartão envia os blocos de bytes extraídos do arquivo CAP para o MCU, o qual efetua o processo de instalação. Dessa forma, o *software* do cartão pode ser atualizado sempre que necessário;
2. O cliente gera suas chaves assimétricas principais, uma RSA e uma de Curvas Elípticas, para aumentar a compatibilidade e flexibilidade do sistema. Essas chaves serão utilizadas para a criação de assinaturas digitais durante o *handshake*. O cliente, então, as envia para o servidor;
3. O servidor recebe as chaves públicas do cliente e gera certificados para elas, assinados com as suas chaves privadas principais e diferentes algoritmos de *hash*. O servidor, então, envia os certificados para o cliente, que os armazena no *Smart Card*.

Durante essa fase de pré-configuração, o servidor atua como uma CA, emitindo certificados para os clientes. Esses certificados são utilizados por eles, posteriormente, para autenticação

com o servidor. O servidor, por sua vez, possui chaves assimétricas principais que são utilizadas para assinar os certificados do cliente e seus próprios certificados. Assim como as do cliente, uma delas é RSA e a outra é de Curvas Elípticas. A depender dos algoritmos de assinatura que o cliente pode verificar (que são informados durante o *handshake*), o servidor decide enviar os certificados assinados com uma ou outra chave. Além de verificar a assinatura do certificado do servidor, o cliente também verifica a sua identidade que está no certificado e contém alguns campos com diferentes informações. Nos testes, foi utilizada a identidade representada na Tabela 4.9.

CAMPO	**VALOR**
ESTADO	PE
LOCALIDADE	Recife
ORGANIZAÇÃO	GPRT
DEPARTAMENTO	IoTeam
NOME	Servidor

Tabela 4.9: Identidade do servidor utilizada nos testes de comunicação entre dispositivo IoT e servidor

A fase de *handshake* é realizada de acordo com o procedimento descrito na Seção 2.4. As operações criptográficas do cliente são realizadas pelo *Smart Card* e as do servidor são implementadas no pacote *Cryptography*. Os parâmetros de segurança que são determinados após o *handshake* são aqueles especificados no RFC 5246:

- Entidade: Servidor ou cliente;
- Algoritmo de PRF: O documento só especifica um, descrito na Seção 2.4;
- Tipo de cifra: De bloco, de fluxo ou AEAD (cifragem e autenticação em um único protocolo). O *Smart Card* só dá suporte a algoritmos de bloco (Tabela A.3);
- Tamanho da chave de cifragem (*write key*): Depende do algoritmo de cifragem;
- Tamanho do bloco cifrado: Depende do algoritmo de cifragem;
- Tamanho do vetor de inicialização: Depende do algoritmo de cifragem. Para as cifras de bloco, é igual ao tamanho do bloco cifrado;
- Algoritmo de MAC: Algoritmos de HMAC, com diferentes funções de *hash*;
- Tamanho do MAC: Depende do algoritmo de *hash* escolhido para o HMAC;
- Tamanho da chave de MAC: Depende do algoritmo de *hash* escolhido para o HMAC. Possui o mesmo valor do tamanho do MAC;
- *master_secret*: Valor pseudoaleatório de tamanho fixo (48 B) calculado durante o *handshake*. No dispositivo, ele é gerado e armazenado dentro do *Smart Card*;
- *client_random* e *server_random*: Valores aleatórios de tamanho fixo (32 B).

O fluxo das mensagens durante o *handshake* é controlado com o auxílio de uma variável de estado, existente tanto no cliente quanto no servidor, a qual informa a ambas as entidades se elas devem receber ou enviar uma mensagem, e qual mensagem deve ser enviada. A Tabela 4.10 sumariza os valores da variável de estado e o que deve ser feito por cada entidade.

ESTADO	**AÇÃO**		**MENSAGEM**
	CLIENTE	SERVIDOR	
0	Enviar	Receber	ClientHello
1	Receber	Enviar	ServerHello
2	Receber	Enviar	ServerCertificate
3	Receber	Enviar	ServerKeyExchange
4	Receber	Enviar	CertificateRequest
5	Receber	Enviar	ServerHelloDone
6	Enviar	Receber	ClientCertificate
7	Enviar	Receber	ClientKeyExchange
8	Enviar	Receber	CertificateVerify
9	Enviar	Receber	ChangeCipherSpec
10	Enviar	Receber	Finished
11	Receber	Enviar	ChangeCipherSpec
12	Receber	Enviar	Finished

Tabela 4.10: Estados do *handshake* e ações de cada entidade

A transição entre estados é condicional, dependendo da suíte de criptografia escolhida e do envio de determinadas mensagens, o que significa que nem todos os estados estarão presentes em todos os *handshakes*. Especificamente:

- O estado 3 só é executado se o algoritmo da troca de chaves for o DH com Chaves Efêmeras, pois a chave efêmera é gerada e enviada pelo servidor nesse momento. Nos outros casos, a chave necessária já está no certificado enviado no estado 2;
- O estado 4 é opcional, o servidor decide se vai ou não requisitar um certificado do cliente para autenticá-lo;
- O estado 6 só é executado se o estado 4 também for executado, ou seja, o cliente só envia um certificado para o servidor se isso lhe for requisitado;
- O estado 8 só é executado se: o cliente tiver enviado um certificado, ou seja, se o estado 6 for executado, e se o algoritmo de troca de chaves não for o DH de Chave Fixa (quando a chave não é gerada na hora para aquela sessão, e sim já existe em um certificado). Isso acontece pois a chave desse certificado não pode ser usada para assinaturas, apenas para o algoritmo de troca de chaves.

A máquina de estados da Figura 4.23 ilustra a transição entre os estados e o fluxograma na Figura 4.24 sumariza o funcionamento geral do sistema.

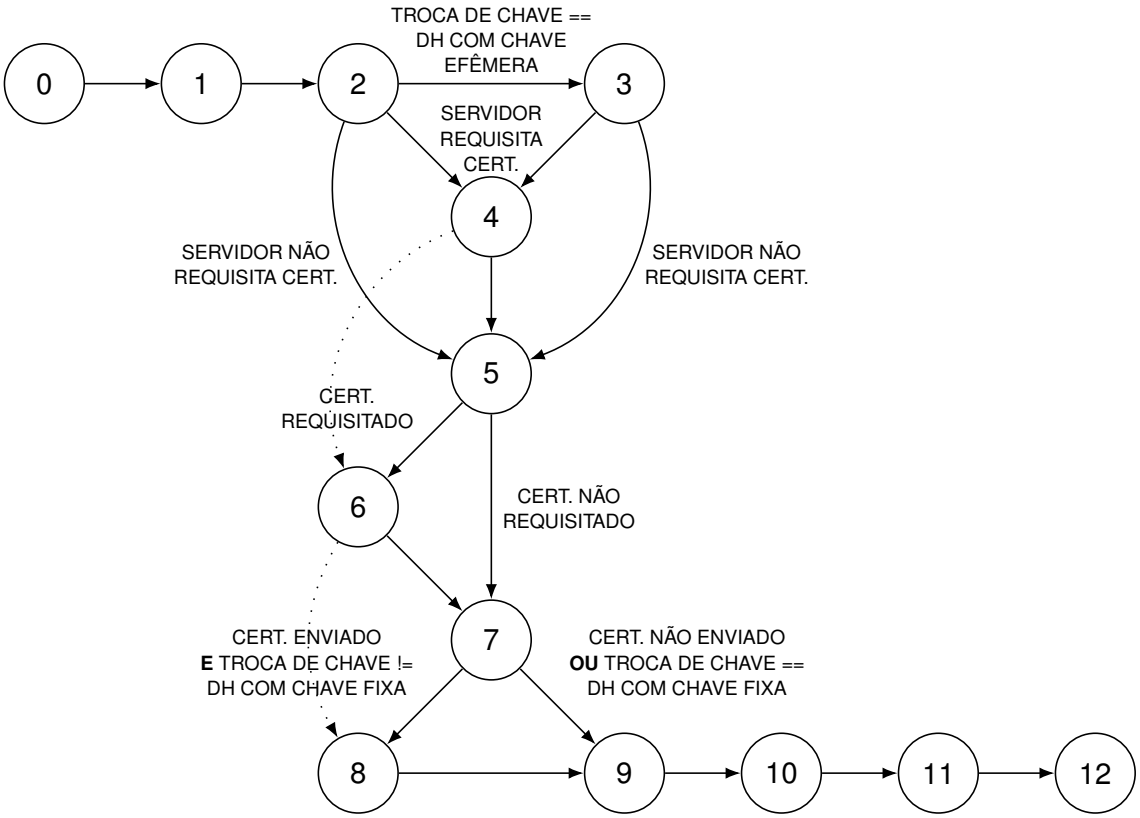


Figura 4.23: Máquina de estados do Protocolo de *Handshake* do TLS

4.3 Considerações Finais

A união entre os projetos que foram desenvolvido nas seções 4.1 e 4.2 compõem o sistema de comunicação segura para IoT proposto neste trabalho. Foram feitos testes de comunicação entre o dispositivo fabricado e o servidor, compostos pela realização da pré-configuração seguida da execução do protocolo de *handshake* (utilizando várias suítes de criptografia definidas no TLS) e, por fim, o envio de mensagens cifradas e autenticadas do cliente para o servidor. Isso mostra que é possível construir dispositivos IoT assegurados por *Smart Cards* e que o objetivo principal do trabalho foi alcançado.

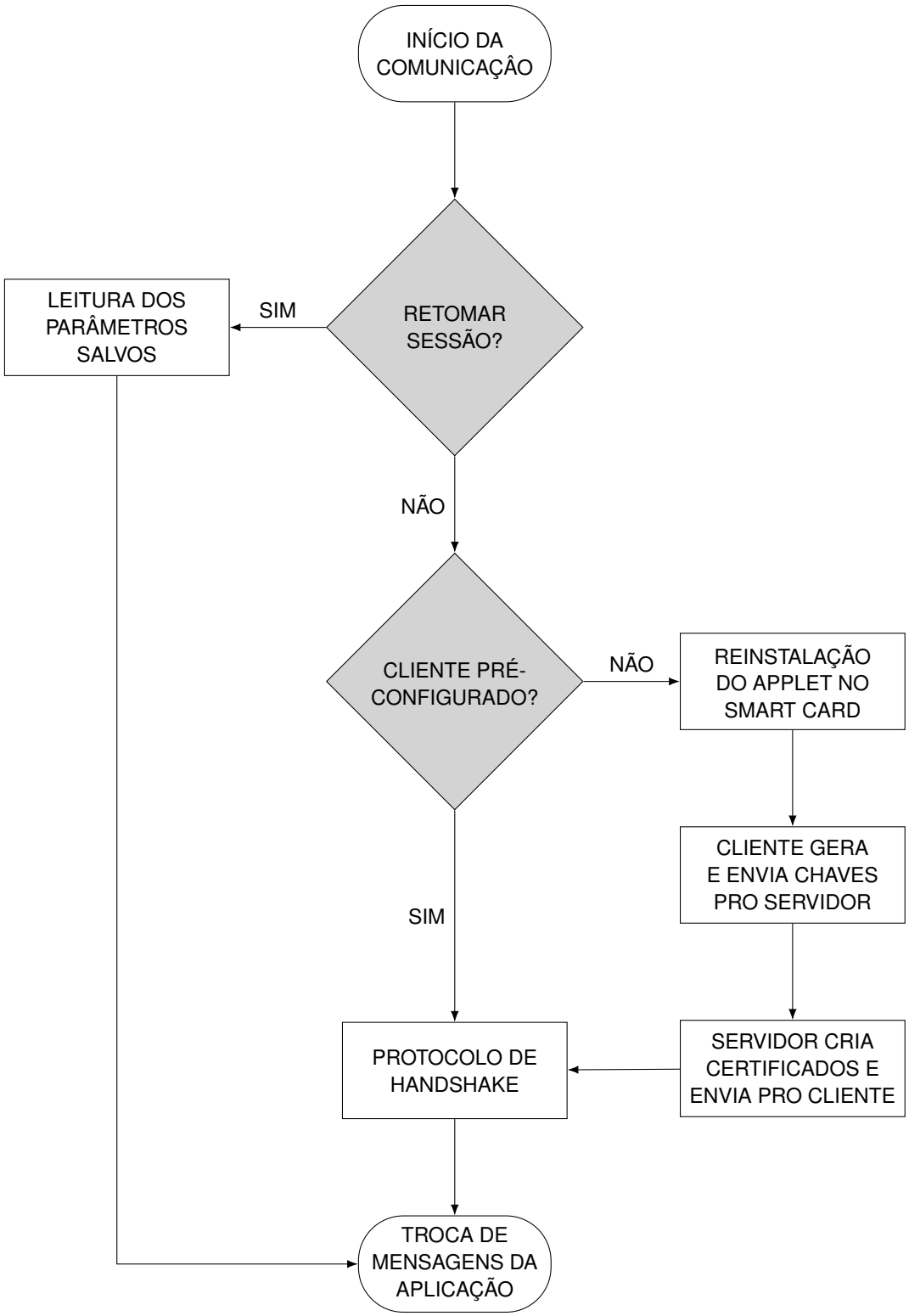


Figura 4.24: Fluxograma de funcionamento do sistema proposto neste trabalho

5

RESULTADOS

Esta capítulo apresenta avaliações e análises realizadas sobre o projeto após a sua finalização.

5.1 Avaliação de Custo

Um dos objetivos deste trabalho é a criação de um dispositivo IoT que seja economicamente acessível. O KiCAD permite a exportação da *Bill of Materials* (BOM) do projeto: uma lista com todos os componentes que são utilizados na sua fabricação. Com essa lista, é possível pesquisar os preços de cada componente para obter uma estimativa do custo do dispositivo.

O site *Octopart* é um mecanismo de busca de componentes eletrônicos que fornece informações como *datasheets* e preços, baseados em pesquisas realizadas em centenas de distribuidores e milhares de fabricantes, segundo a própria página. O Octopart possui também uma ferramenta para importação de BOMs, a qual realiza uma busca de todos os itens nela presentes e fornece o preço baseado na quantidade que se deseja produzir. É possível, inclusive, fazer uma estimativa com os fornecedores que possuem os preços mais baixos. Como a BOM é gerada a partir dos componentes que são soldados à placa, ficam excluídos dessa pesquisa automática de preços o *Smart Card* e a própria placa de cobre dupla face.

O repositório do *software* GlobalPlatformPro (PALJAK, 2016) possui um guia de compras para Java Cards no qual são mencionadas algumas lojas de SCs, entre elas a *SmartCardSource*. Seu Java Card mais barato, o modelo J2A040 da NXP com EEPROM de 40 kB, custa US$4,99 para uma unidade. Esse preço diminui gradativamente a cada centena de unidades até chegar em US$3,49 para mil ou mais unidades.

Agregando os preços dos componentes eletrônicos, obtidos no Octopart, com os preços de *Smart Card*s, são obtidos os valores da Tabela 5.1. Pode-se observar uma queda considerável do custo que se tem para fabricar apenas uma unidade para o custo unitário de mil unidades, de aproximadamente 30%. É importante destacar que, se esse dispositivo fosse fabricado a um custo de US$18,00 e vendido com um lucro de 100% por US$36, ele ainda estaria na mesma faixa de preço de várias placas da família Arduino, por exemplo, com a vantagem adicional da segurança fornecida pelo *Smart Card*.

	PREÇO/UNIDADE		
	1 Unidade	100 Unidades	1000 Unidades
Componentes Eletrônicos	US$21,74	US$15,72	US$14,51
Smart Cards	US$4,99	US$4,75	US$3,49
Total	US$26,73	US$20,47	US$18,00

Tabela 5.1: Custo unitário do dispositivo IoT desenvolvido neste trabalho, para diferentes quantidades produzidas

5.2 Avaliação de Consumo de Energia

A análise do consumo de energia do dispositivo IoT é muito importante, pois determinadas aplicações podem exigir um funcionamento independente da rede elétrica, com a utilização de baterias. A depender do propósito da aplicação, um dos requerimentos pode ser o funcionamento durante um tempo considerável com uma única carga, quando não é conveniente recarregar a bateria frequentemente. Atualmente, o dispositivo proposto é alimentado pela porta USB, mas uma versão com bateria é uma possibilidade que pode ser realizada.

O dispositivo proposto possui alguns CIs responsáveis pelo consumo de energia da placa. Em seus respectivos *datasheets* é possível encontrar valores típicos para correntes de funcionamento e estimar quanto tempo o dispositivo poderia funcionar com bateria. A Tabela 5.2 lista os *chips* e suas respectivas correntes de funcionamento. A corrente para o ESP-12E, de acordo com *datasheet* (Espressif, 2017), varia de acordo com vários parâmetros como: modo de comunicação (recebimento ou transmissão), protocolo Wi-Fi (802.11b/g/n) e potência de transmissão. O texto informa, então, um valor médio de 80 mA, o qual foi utilizado na tabela. A corrente do *Smart Card* foi obtida do padrão ISO7816-3, que informa a corrente máxima de funcionamento permitida pra um cartão classe A.

A carga nominal de uma bateria geralmente é dada em miliampères-hora (mAh). Embora esse valor dependa de fatores como temperatura e taxa de descarga, ele pode ser utilizado para fazer estimativas aproximadas de quanto tempo dura a bateria com um determinado padrão de uso. Uma breve pesquisa em lojas de componentes eletrônicos como Adafruit e Sparkfun mostra alguns valores típicos de carga para baterias de lítio utilizadas em projetos eletrônicos. Esses valores variam de 1000 mAh a 6600 mAh.

Considerando os valores de 1000, 2200, 4400 e 6600 mAh, caso o dispositivo IoT funcionasse continuamente com o valor total de corrente da Tabela 5.2, o tempo de funcionamento poderia ser calculado dividindo a carga nominal pela corrente do dispositivo. Dessa forma, ele seria de aproximadamente 5, 12, 25 e 38 horas, respectivamente. Para aplicações que requerem um tempo de funcionamento de dias até que seja realizada uma manutenção como, por exemplo, medição de consumo de energia elétrica ou água em residências, essas durações não são aceitáveis.

COMPONENTE	CORRENTE (mA)
Oscilador 4 MHz	7
Conversor para o Cartão	0,805
Conversor para as GPIOs	0,01
Conversor para Porta USB	15
Regulador de Tensão	11
Módulo ESP-12E	80
Smart Card	60
Total	173,815

Tabela 5.2: Correntes típicas de funcionamento para os *chips* do dispositivo IoT desenvolvido neste trabalho

Uma possível solução seria a utilização de modos de baixo consumo de energia, geralmente presentes em microcontroladores utilizados em projetos eletrônicos. O ESP8266 possui três modos de economia de energia:

- *Modem-sleep*: Apenas o modem Wi-Fi é desligado, mas a conexão é mantida. Corrente de funcionamento de 15 mA;
- *Light-sleep*: O modem Wi-Fi, o relógio do sistema e a CPU são desligados, mas a conexão é mantida. Corrente de funcionamento de 0,4 mA;
- *Deep-sleep*: Todos os módulos do *chip* são completamente desligados (a conexão não é mantida) e apenas o Relógio de Tempo Real continua funcionando, para acordar o microcontrolador em um tempo pré-determinado. Corrente de funcionamento de 0,02 mA.

Para realização das estimativas, pode ser considerada uma aplicação onde o dispositivo faz medições periódicas em intervalos de x minutos, onde ele passa 1 minuto ligado realizando diferentes tarefas e utilizando toda sua capacidade, e nos $x-1$ minutos restantes ele se encontra no estado de *deep-sleep*. Nesse caso, durante o estado de espera, todos os *chips* relacionados ao *Smart Card* também estarão desligados, bem como o conversor da porta USB (o qual só é ligado quando a porta está em uso). Isso reduz o total de corrente para $0,01+11+0,02 = 11,03$ mA. A carga em mAh utilizada pelo dispositivo a cada ciclo nessa situação pode ser, então, calculada a partir da seguinte expressão:

$$\frac{11,03(x-1)+158,815}{60} \approx 2,46+0,18x$$

Multiplicando esse valor pela quantidade de ciclos de x minutos presentes em uma hora (ou seja, $60/x$), é possível descobrir a carga utilizada por hora. Dividindo, então, a carga nominal da bateria por esse resultado, pode-se encontrar quantas horas a bateria vai durar,

aproximadamente. A expressão para as horas fica dessa forma:

$$\frac{C_{nominal}}{(2,46+0,18x)\left(\frac{60}{x}\right)} = \frac{x \cdot C_{nominal}}{147,60+10,80x}$$

Considerando diferentes valores para x, a Tabela 5.3 mostra as durações (em horas) da bateria para os quatro valores de carga nominal citados anteriormente e a Figura 5.1 mostra o gráfico obtido a partir da tabela.

	PERÍODO DE MEDIÇÃO (min)					
CARGA (mAh)	15	30	45	60	75	90
1000	48,45 h	63,61 h	71,02 h	75,41 h	78,32 h	79,99 h
2200	106,59 h	139,95 h	156,25 h	165,91 h	172,31 h	175,97 h
4400	213,18 h	279,90 h	312,50 h	331,83 h	344,61 h	351,94 h
6600	319,77 h	419,85 h	468,75 h	497,74 h	516,92 h	527,91 h

Tabela 5.3: Duração da bateria em horas para os diferentes valores de carga nominal e período de medição

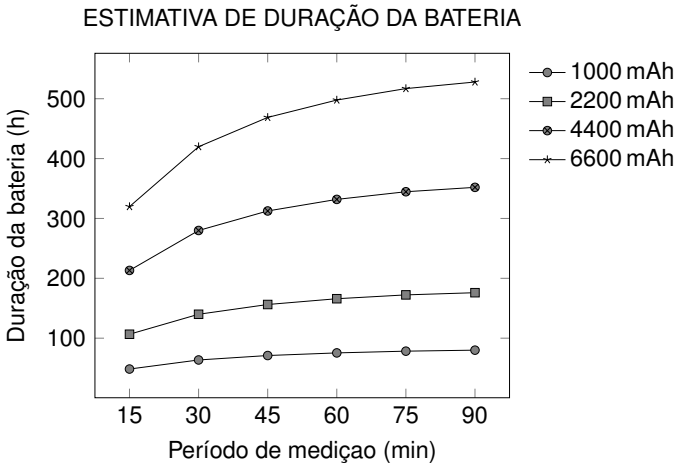


Figura 5.1: Gráfico de horas de funcionamento estimadas, obtido com os valores da Tabela 5.3

É possível observar que mesmo no pior caso, com uma carga de apenas 1000 mAh em uma aplicação periódica de 15 min, existe um ganho de horas de funcionamento de quase 1000% (de 5 para 48 horas), relativamente a uma aplicação onde o funcionamento é contínuo. No melhor caso, com uma bateria de 6600 mAh em uma aplicação com período de 90 min, o dispositivo pode funcionar por aproximadamente 22 dias. Também pode-se ver que quanto maior a carga nominal da bateria, mais o aumento do período influencia na quantidade de horas.

6

CONCLUSÃO

6.1 Considerações Finais

A segurança dos dispositivos e a privacidade das informações transmitidas são fatores-chave no desenvolvimento e adoção da Internet das Coisas. Este trabalho mostrou que é possível criar um dispositivo IoT seguro a partir de um microcontrolador de baixo custo, com restrições de memória e processamento, integrando-o a um *Smart Card*, o qual é especializado em executar operações de segurança e armazenamento seguro de informaçoes.

Todo o trabalho foi fundamentado em conceitos sólidos de criptografia e segurança discutidos no Capítulo 2 e também nos padrões internacionais criados para especificar o funcionamento e a interoperabilidade entre os diversos componentes envolvidos neste projeto.

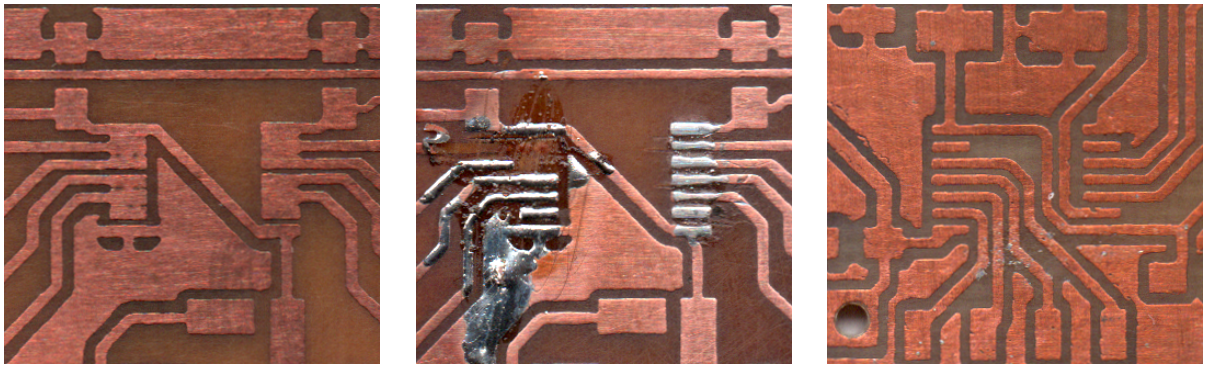
Como visto no Capítulo 4, o projeto foi iniciado com a criação de algo que seria um módulo de leitor de cartão para um Arduino UNO, um dos microcontroladores mais utilizados em prototipação de projetos eletrônicos e de IoT. Ao longo do seu desenvolvimento, o projeto foi alterado para ser uma placa única, incorporando o microcontrolador ESP8266, o qual já possui capacidades de comunicação sem fio, via Wi-Fi. Paralelamente ao desenvolvimento de *hardware*, foram criados *softwares* para o microcontrolador, o cartão e o servidor a partir de ferramentas livres e *open source*. A utilização do cartão no projeto permitiu a implementação do protocolo TLS, amplamente utilizado na Internet, que garante a privacidade dos dados que são trocados e a autenticação entre as entidades que estão se comunicando. Adicionalmente, foram desenvolvidas bibliotecas para Arduino e placas compatíveis, as quais implementam operações avançadas com o cartão, que vão além da simples troca de mensagens.

Por fim, as análises realizadas no Capítulo 5 mostraram que o dispositivo desenvolvido pode ter um custo compatível com outros dispositivos no mercado, se produzido em grande escala. Além disso, foi visto que o dispositivo, apesar de ter sido projetado para operar com uma conexão USB, é capaz de funcionar por dias com restrições de consumo de energia, sendo alimentado por uma bateria, em uma aplicação onde são realizadas leituras periódicas de um determinado sensor.

6.2 Dificuldades Encontradas

Foram encontradas dificuldades tanto na elaboração de *hardware*, quanto na de *software* do projeto. No que diz respeito ao *hardware*, destacam-se:

- Durante a fabricação das placas que seriam conectadas ao Arduino (Figura 4.5), o alinhamento do *Smart Card* com o *slot* foi complicado de ser realizado, visto que o *slot* era feito para cartões ID-000 e foi adaptado para cartões ID-1. A estratégia utilizada para conseguir posicionar o cartão corretamente foi criar um componente no próprio ECAD com um contorno do tamanho exato de um cartão, o qual não seria corroído durante a fabricação, e posicioná-lo de maneira alinhada com o componente do *slot*;
- A fixação do cartão no local correto, com o auxílio de parafusos e porcas posicionados nas bordas, também não era totalmente satisfatória. Por vezes, era necessário pressionar o cartão manualmente para conseguir realizar a conexão. Com a introdução da peça de plástico feita na impressora 3D (Figura 4.6), esse problema deixou de existir. No entanto, também foram necessários vários testes e impressões até se chegar a uma peça que estabelecesse uma conexão entre cartão e *slot* de forma confiável;
- A utilização do ESP8266 no lugar do Arduino adicionou um trabalho extra, pois os módulos fabricados com esse chip requerem de uma PCI adicional com alguns *switches* para que possam ser gravados, além de um conversor de serial/USB para conectá-los com o computador;
- Todas as placas citadas neste trabalho foram fabricadas e soldadas de forma manual. A partir da confecção da primeira placa do sistema englobando o módulo ESP8266 e o *Smart Card* (Figura 4.18), surgiram problemas relacionados aos tamanhos dos *chips* que faziam parte da placa. Por serem CIs muito pequenos, a resolução da impressão do *layout* da placa não era boa o suficiente, de forma que as trilhas de cada "perna" do *chip* às vezes se conectavam (Figuras 6.1a, 6.2a e 6.3a). Para não perder a placa inteira por causa de um *chip*, foram feitas correções manuais no cobre, com o auxílio de um estilete. Em algumas ocasiões, isso acabava por destruir completamente a trilha e inutilizar a placa (Figuras 6.2b e 6.3a). Esses problemas foram resolvidos com a utilização de *footprints* com trilhas mais finas, que apesar de não ficarem sempre perfeitas possuíam uma probabilidade menor de saírem unidas após a corrosão do cobre (Figuras 6.1c e 6.2c);
- Pelo mesmo motivo do item anterior, a espessura das trilhas, em alguns momentos durante a soldagem as trilhas se descolavam do substrato (Figuras 6.1b e 6.3b), também inutilizando a placa.

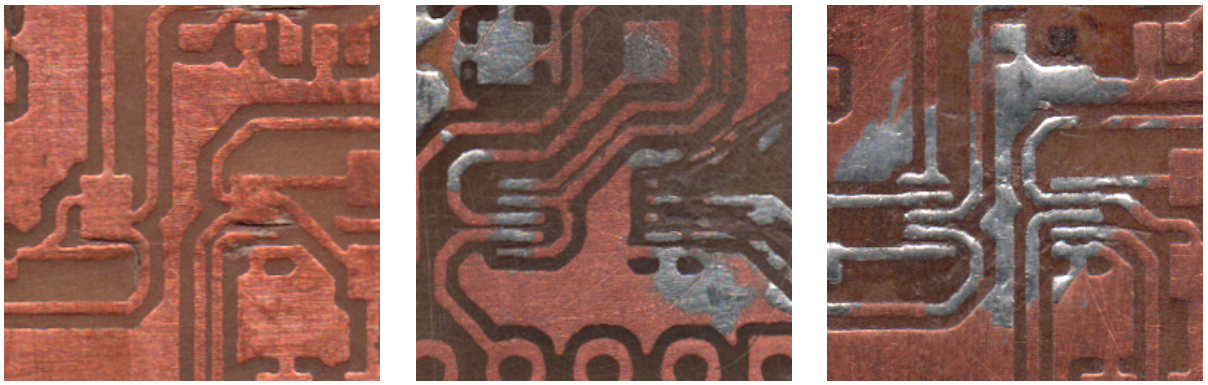


(a) Trilhas unidas após corrosão

(b) Trilhas destruídas durante a soldagem

(c) Trilhas sem defeito

Figura 6.1: Falhas na soldagem do *chip* conversor de tensão para GPIOs

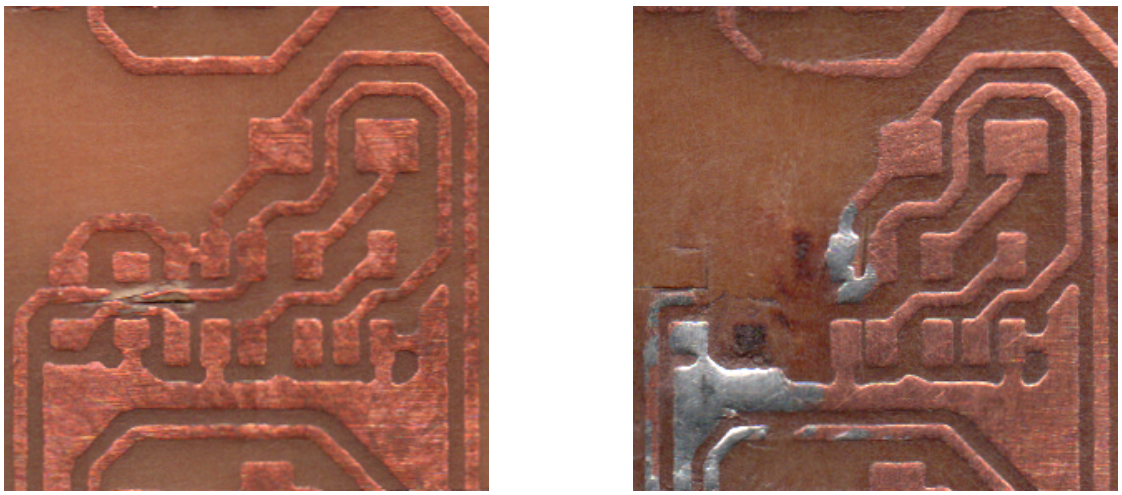


(a) Trilhas unidas após corrosão

(b) Trilhas destruídas após tentativa de correção

(c) Trilhas sem defeito

Figura 6.2: Falhas na soldagem do *chip* conversor de tensão para *Smart Card*s



(a) Trilhas unidas após corrosão e destruídas após tentativa de correção

(b) Trilhas destruídas durante a soldagem

Figura 6.3: Falhas na soldagem dos *chips* do circuito de relógio da Figura 4.17

Quanto ao desenvolvimento de *software*, foram encontrados os seguintes obstáculos:

- O acionamento do *Watchdog Timer* do ESP8266, como descrito na Subseção 4.2.1.
- A falta de experiência prévia com programação em Java Cards gerou algumas dificuldades relacionadas à memória do cartão. A implementação de funções proprietárias ou não presentes na API como o HMAC, PRF e outras relativas ao TLS geralmente exigiam a criação de *arrays* para armazenamento de dados intermediários dos cálculos. O cartão, no entanto, possui um limite de memória destinada a essas variáveis e quando esse limite é ultrapassado, a execução do *applet* não é feita da forma correta. Com o tempo, descobriu-se que esse era o motivo e as funções foram otimizadas para reduzir o uso da memória. Alguns textos com guias e boas práticas de programação para Java Card auxiliaram nesse processo (GEMALTO, 2009)(RUIMTOOLS, 2010);
- Durante o protocolo de *handshake* do TLS (Seção 2.4), é necessário armazenar todas as mensagens trocadas para criar as mensagens *CertificateVerify* e *Finished*, ou então calcular *hashes* intermediários que são atualizados a cada nova mensagem. Este último método é preferível, pois não utiliza tanta memória (nos testes realizados, as mensagens do *handshake* chegavam a um tamanho de aproximadamente 5 kB, enquanto que um *hash* possui poucas dezenas de bytes). Inicialmente, foram implementadas funções de *hashes* intermediários SHA256 no próprio MCU. No entanto, isso anula o objetivo do sistema de realizar todas as operações criptográficas no *Smart Card*. Dessa forma, foi testada a possibilidade de calcular os *hashes* intermediários dentro do cartão, através das funções na API. Porém, não era possível extrair os *hashes* intermediários do cartão, nem atualizá-los após seu desligamento e religamento. Com isso, foi utilizado o primeiro método, onde todas as mensagens foram armazenadas em uma variável e, quando necessário, seu *hash* é calculado dentro do cartão. Uma das desvantagens desse método é a incompatibilidade com MCUs com pouca memória.

6.3 Trabalhos Futuros

Há muito que ainda pode ser feito com relação a esse trabalho, incluindo melhorias de *hardware* e de *software* e avaliações e testes do sistema como um todo. A seguir, são citadas algumas dessas tarefas:

- Diminuição do tamanho dispositivo a partir de algumas modificações, como: utilização de *Smart Cards* menores (tamanho de nano SIM); soldagem do ESP8266 e componentes necessários diretamente na placa, ao invés do módulo; utilização de técnicas profissionais de fabricação de PCIs, principalmente para soldagem de componentes menores, confecção de placas com duas ou mais camadas e criação de *vias* (furos que conectam as diferentes camadas de uma placa de circuito). No

trabalho realizado, as vias são feitas com fios de cobre que atravessam furos na placa e são soldados em ambos os lados, o que toma mais espaço do que deveria. Em um processo de fabricação profissional, os furos são bem menores e revestidos com material condutor (não têm fios atravessando);

- Utilização de Java Cards com APIs mais recentes (atualmente, está na versão 3.0.5), que contém operações criptográficas mais modernas e seguras;
- Aumentar a conformidade da implementação do TLS com a especificação do RFC 5246. O código foi escrito com foco nos protocolos centrais do TLS, Registro e *Handshake* e a implementação não ficou completa. Um exemplo de algo que deveria ser adicionado é o Protocolo de Alerta, que define as mensagens de erros que podem ocorrer ao longo da execução dos outros protocolos;
- Avaliar a performance do dispositivo durante a realização do protocolo de *handshake* a partir de métricas como tempo de execução;
- Medir o consumo de corrente em um cenário de aplicação e comparar os resultados com a estimativa realizada na Seção 5.2;
- Desenvolver maneiras de avaliar a segurança do dispositivo,a partir de diferentes pontos de vista como: implementação de *software*, fabricação de *hardware* etc.;
- Testar os dispositivos em um cenário realista, para que seu funcionamento possa ser avaliado em uma aplicação que envolva a leitura de diferentes sensores e o envio das informações de forma segura para o servidor;
- Avaliar o desempenho do dispositivo durante a sua utilização em diferentes aplicações e estudar sua viabilidade para cada uma delas.

REFERÊNCIAS

ABOMHARA, M.; KøIEN, G. M. Security and privacy in the internet of things: Current status and open issues. In: *2014 International Conference on Privacy and Security in Mobile Systems (PRISMS)*. [S.l.: s.n.], 2014. p. 1–8.

ALLEN, C. *et al. The Transport Layer Security (TLS) Protocol Version 1.2*. [S.l.], 2008. Disponível em: <http://www.rfc-editor.org/rfc/rfc5246.txt>.

ARDIRI, A. *Is it possible to secure micro-controllers used within IoT?* 2014. Online. Disponível em: <https://evothings.com/is-it-possible-to-secure-micro-controllers-used-within-iot/>.

BADRA, M.; URIEN, P. Tls tandem. In: *2008 New Technologies, Mobility and Security*. [S.l.: s.n.], 2008. p. 1–5. ISSN 2157-4952.

BARGSTEDT, F. *ArduinoSCLib: Smart Card Library for Arduino compatible boards*. 2016. Online. Disponível em: <https://sourceforge.net/projects/arduinosclib/>.

BELLARE, M.; NAMPREMPRE, C. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In: ____. *Advances in Cryptology — ASIACRYPT 2000: 6th International Conference on the Theory and Application of Cryptology and Information Security Kyoto, Japan, December 3–7, 2000 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. p. 531–545. ISBN 978-3-540-44448-0. Disponível em: <https://doi.org/10.1007/3-540-44448-3_41>.

BERTINO, E. *et al.* Internet of things (iot): Smart and secure service delivery. *ACM Trans. Internet Technol.*, ACM, New York, NY, USA, v. 16, n. 4, p. 22:1–22:7, dez. 2016. ISSN 1533-5399. Disponível em: <http://doi.acm.org/10.1145/3013520>.

BLAKE-WILSON, S. *et al. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. [S.l.], 2006. Disponível em: <http://www.rfc-editor.org/rfc/rfc4492.txt>.

COOPER, D. *et al. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. [S.l.], 2008. Disponível em: <http://www.rfc-editor.org/rfc-/rfc5280.txt>.

CORCORAN, D.; CUCINOTTA, T. *MUSCLE Cryptographic Card Edge Definition for Java Enabled Smartcards*. [S.l.], 2001. Disponível em: <https://pcsclite.alioth.debian.org/musclecard-.com/musclecard/files/mcardprot-1.2.1.pdf>.

CUCINOTTA, T.; NATALE, M. D.; CORCORAN, D. A protocol for programmable smart cards. In: *14th International Workshop on Database and Expert Systems Applications, 2003. Proceedings.* [S.l.: s.n.], 2003. p. 369–374. ISSN 1529-4188.

DICHOU, K.; TOURTCHINE, V.; RAHMOUNE, F. Simulation of apdus exchanged between a microcontroller smart card and a reader. In: *7th International Conference on Modelling, Identification and Control (ICMIC)*. [S.l.: s.n.], 2015. p. 1–4.

EDSON, B. *Creating the Internet of Your Things*. [S.l.], 2015. Disponível em: <http:/-/download.microsoft.com/download/C/F/7/CF78575B-711E-4E1B-8BAB-3ED1657DFA82-/Creating_the_Internet_of_Your_Things.pdf>.

Espressif. *ESP8266EX Datasheet Version 5.4*. [S.l.], 2017. Disponível em: <http://espressif-.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf>.

GALLAGHER, S. *The future is the Internet of Things—deal with it*. 2015. Online. Disponível em: <http://arstechnica.com/unite/2015/10/the-future-is-the-internet-of-things-deal-with-it/>.

GEMALTO. *Java Card & STK Applet Development Guidelines, Version 2.0*. 2009.

GlobalPlatform. *Card Specification Version 2.1.1*. [S.l.], 2003. Disponível em: <http://www-.globalplatform.org/specificationscard.asp>.

GUTMANN, P. *Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. [S.l.], 2014. Disponível em: <http://www.rfc-editor.org/rfc/rfc7366-.txt>.

Hackster. *The 2016 Hackster.io Maker Survey Official Report*. [S.l.], 2016. Disponível em: <https://www.hackster.io/survey>.

HERDER, C. *et al.* Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, v. 102, n. 8, p. 1126–1141, Aug 2014. ISSN 0018-9219.

HILL, K. *The Half-Baked Security Of Our 'Internet Of Things'*. 2014. Online. Disponível em: <http://www.forbes.com/sites/kashmirhill/2014/05/27/article-may-scare-you-away-from-internet-of-things/#313a3c7023dd>.

HUMMEN, R. *et al.* Towards viable certificate-based authentication for the internet of things. In: *Proceedings of the 2Nd ACM Workshop on Hot Topics on Wireless Network Security and Privacy*. New York, NY, USA: ACM, 2013. (HotWiSec '13), p. 37–42. ISBN 978-1-4503-2003-0. Disponível em: <http://doi.acm.org/10.1145/2463183.2463193>.

INTEL. *A Guide to the Internet of Things*. 2015. Disponível em: <http://www.intel.com/content-/dam/www/public/us/en/images/iot/guide-to-iot-infographic.png>.

ISO/IEC. *ISO/IEC 7816-1: Identification cards – Integrated circuit(s) cards with contacts – Part 1: Physical characteristics*. Suíça, 1998. Disponível em: <https://www.iso.org/standard/54089-.html>.

ISO/IEC. *ISO/IEC 7810: Identification cards – Physical characteristics*. Suíça, 2003. Disponível em: <https://www.iso.org/standard/31432.html>.

ISO/IEC. *ISO/IEC 7816-8: Identification cards – Integrated circuit cards – Part 8: Commands for security operations*. Suíça, 2004. Disponível em: <https://www.iso.org/standard/37989.html>.

ISO/IEC. *ISO/IEC 7816-9: Identification cards – Integrated circuit cards – Part 9: Commands for card management*. Suíça, 2004. Disponível em: <https://www.iso.org/standard/37990.html>.

ISO/IEC. *ISO/IEC 7816-12: Identification cards – Integrated circuit cards – Part 12: Cards with contacts – USB electrical interface and operating procedures*. Suíça, 2005. Disponível em: <https://www.iso.org/standard/40604.html>.

ISO/IEC. *ISO/IEC 7816-4: Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange*. Suíça, 2005. Disponível em: <https://www.iso.org/standard/36134.html>.

ISO/IEC. *ISO/IEC 7816-3: Identification cards – Integrated circuit cards – Part 3: Cards with contacts – Electrical interface and transmission protocols*. Suíça, 2006. Disponível em: <https://www.iso.org/standard/38770.html>.

ISO/IEC. *ISO/IEC 7816-2: Identification cards – Integrated circuit cards – Part 2: Cards with contacts – Dimensions and location of the contacts*. Suíça, 2007. Disponível em: <https://www.iso.org/standard/45989.html>.

ITU-T. *Recommendation X.800: Data Communication Networks; Open Systems Interconnection (OSI); Security, Structure and Applications – Security Architecture for Open Systems Interconnection for CCITT Applications*. Geneva, 1991. Disponível em: <http://www.itu.int/rec/T-REC-X.800/en>.

ITU-T. *ITU-T Recommendation X.690: Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. [S.l.], 2002. Disponível em: <http://www.itu.int/rec/T-REC-X.690/en>.

ITU-T. *ITU-T Recommendation X.680: Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*. [S.l.], 2008. Disponível em: <http://www.itu.int/rec/T-REC-X.680/en>.

Java Card Forum. *20 Years of the Java Card Forum*. 2017. Disponível em: <https:/-/javacardforum.files.wordpress.com/2017/03/jcf_20infographic_final_1.jpg>.

KANUPARTHI, A.; KARRI, R.; ADDEPALLI, S. Hardware and embedded security in the context of internet of things. In: *Proceedings of the 2013 ACM Workshop on Security, Privacy & Dependability for Cyber Vehicles*. New York, NY, USA: ACM, 2013. (CyCAR '13), p. 61–64. ISBN 978-1-4503-2487-8. Disponível em: <http://doi.acm.org/10.1145/2517968-.2517976>.

KATZ, J.; LINDELL, Y. *Introduction to Modern Cryptography*. 2nd. ed. Boca Raton, Florida, USA: Chapman & Hall/CRC, 2014. (Cryptography and Network Security). ISBN 9781466570276.

KIM, D. S. *et al.* On the design of an embedded biometric smart card reader. *IEEE Transactions on Consumer Electronics*, v. 54, n. 2, p. 573–577, maio 2008. ISSN 0098-3063.

KRAWCZYK, H. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In: ____. *Advances in Cryptology — CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 310–331. ISBN 978-3-540-44647-7. Disponível em: <https://doi.org/10.1007/3-540-44647-8_19>.

KRAWCZYK, H.; BELLARE, M.; CANETTI, R. *HMAC: Keyed-Hashing for Message Authentication*. [S.l.], 1997. Disponível em: <http://www.rfc-editor.org/rfc/rfc2104.txt>.

KREBS, B. *This is Why People Fear the 'Internet of Things'*. 2016. Online. Disponível em: <http://krebsonsecurity.com/2016/02/this-is-why-people-fear-the-internet-of-things/>.

LIU, Y. *et al.* An efficient privacy protection solution for smart home application platform. In: *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*. [S.l.: s.n.], 2016. p. 2281–2285.

LUCHINI, L. *asn1js: JavaScript Generic ASN.1 Parser/Decoder*. 2017. Online. Disponível em: <https://github.com/lapo-luchini/asn1js>.

MANYIKA, J. *et al. The Internet of Things: Mapping the Value Beyond the Hype*. [S.l.], 2015. Disponível em: <http://www.mckinsey.com/business-functions/business-technology/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world>.

MUJI, S. Z. M. *et al.* Simulation of smart card interface with pic for vehicle security system. In: *2008 International Conference on Computer and Communication Engineering*. [S.l.: s.n.], 2008. p. 878–882.

NIST. *FIPS PUB 199: Standards for Security Categorization of Federal Information and Information Systems*. Gaithersburg, Maryland, USA, 2004. Disponível em: <http://csrc.nist.gov-/publications/fips/fips199/FIPS-PUB-199-final.pdf>.

Nordic Semiconductor. *nRF24L01+ Single Chip 2.4GHz Transceiver Product Specification v1.0*. Noruega, 2008. Disponível em: <http://www.nordicsemi.com/eng/nordic/download_resource-/8765/2/50113066/2726>.

OpenSC Team. *OpenSC: Open source Smart Card tools and middleware*. 2015. Online. Disponível em: <https://github.com/OpenSC/OpenSC>.

Oracle. *Java Card Technology: Providing a Secure and Ubiquitous Platform for Smart Cards*. [S.l.], 2012. Disponível em: <http://www.oracle.com/technetwork/java/embedded/javacard-/documentation/datasheet-149940.pdf>.

PAAR, C.; PELZL, J. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st. ed. [S.l.]: Springer, 2010. ISBN 9783642041006, 9783642041013.

PALJAK, M. *GlobalPlatformPro: Load and manage applets on compatible JavaCards from command line or from your Java project*. 2016. Online. Disponível em: <https://github.com-/martinpaljak/GlobalPlatformPro/>.

PALJAK, M. *Applet Playground: Educational repository for getting to know JavaCard development by learning from existing open source software*. 2017. Online. Disponível em: <https://github.com/martinpaljak/AppletPlayground>.

PAWAR, A. B.; GHUMBRE, S. A survey on iot applications, security challenges and counter measures. In: *2016 International Conference on Computing, Analytics and Security Trends (CAST)*. [S.l.: s.n.], 2016. p. 294–299.

PIETIG, A. *Functional Specification of the OpenPGP application on ISO Smart Card Operating Systems*. Detmold, Alemanha, 2015. Disponível em: <http://www.g10code.com/docs/openpgp-card-3.0.pdf>.

PORUP, J. M. *"Internet of Things"security is hilariously broken and getting worse*. 2016. Online. Disponível em: <https://arstechnica.com/information-technology/2016/01/how-to-search-the-internet-of-things-for-photos-of-sleeping-babies/>.

Python Cryptographic Authority. *Cryptography: Package Designed to Expose Cryptographic Primitives and Recipes to Python Developers*. 2017. Online. Disponível em: <https://github-.com/pyca/cryptography>.

RANKL, W.; EFFING, W. *Smart Card Handbook*. 4th. ed. [S.l.]: Wiley, 2010. ISBN 9780470743676.

RIBEIRO, J. *Securing MQTT communication between Ardruino and Mosquitto*. 2012. Online. Disponível em: <https://www.justinribeiro.com/chronicle/2012/11/08/securing-mqtt-communication-between-ardruino-and-mosquitto/>.

RIEMANN, T. *ArduinoDES: DES and Triples DES Encryption and Decryption for the Arduino Microcontroller Platform*. 2015. Online. Disponível em: <https://github.com/Octoate-/ArduinoDES>.

RSA Laboratories. *PKCS #15: Cryptographic Token Information Format Standard, Version 1.1*. Bedford, Massachusetts, USA, 2000. Disponível em: <http://www.emc.com/emc-plus/rsa-labs-/standards-initiatives/pkcs-15-cryptographic-token-information-format.htm>.

RUIMTOOLS. *Java Card: Programming Guidelines and Best Practise*. 2010. Online. Disponível em: <http://www.ruimtools.com/doc.php?doc=jc_best>.

SAS. *The Internet of Things: Get in on the next big thing*. 2016. Disponível em: <https://www.sas.com/content/sascom/en_us/insights/big-data/internet-of-things/the-internet-of-things-infographic/_jcr_content/par/styledcontainer_59e5/par/image_9900.img.png-/1448314822621.png>.

SCHNEIER, B. *Click Here to Kill Everyone: With the Internet of Things, we're building a world-size robot. How are we going to control it?* 2017. Online. Disponível em: <http://nymag-.com/selectall/2017/01/the-internet-of-things-dangerous-future-bruce-schneier.html>.

SHIREY, R. W. *Internet Security Glossary*. [S.l.], 2000. Disponível em: <http://www.rfc-editor-.org/rfc/rfc2828.txt>.

STALLINGS, W. *Cryptography and Network Security: Principles and Practice*. 5th. ed. [S.l.]: Prentice Hall, 2011. ISBN 9780136097044.

Sun Microsystems. *Application Programming Interface, Java Card Platform, Version 2.2.2*. Santa Clara, California, USA, 2006. Disponível em: <http://download.oracle.com/otndocs/jcp-/java_card_kit-2.2.2-fr-oth-JSpec/>.

Sun Microsystems. *Development Kit User's Guide For the Binary Release with Cryptography Extensions, Java Card Platform, Version 2.2.2*. Santa Clara, California, USA, 2006. Disponível em: <http://download.oracle.com/otndocs/jcp/java_card_kit-2.2.2-fr-oth-JSpec/>.

Sun Microsystems. *Virtual Machine Specification, Java Card Platform, Version 2.2.2*. Santa Clara, California, USA, 2006. Disponível em: <http://download.oracle.com/otndocs/jcp-/java_card_kit-2.2.2-fr-oth-JSpec/>.

TAYLOR, H. *How the 'Internet of Things' could be fatal*. 2016. Online. Disponível em: <http://www.cnbc.com/2016/03/04/how-the-internet-of-things-could-be-fatal.html>.

TEKEOGLU, A.; TOSUN, A. S. A testbed for security and privacy analysis of iot devices. In: *2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. [S.l.: s.n.], 2016. p. 343–348.

VELOSA, A.; SCHULTE, W. R.; LHEUREUX, B. J. *Hype Cycle for the Internet of Things*. [S.l.], 2015. Disponível em: <https://info.microsoft.com/CO-AAIoT-CNTNT-FY16-07Dec15-Gartner-HypeCycle-IoT-Register.html?ls=Website>.

WISNIEWSKI, C. *7 tips for securing the Internet of Things*. 2016. Online. Disponível em: <https://nakedsecurity.sophos.com/2016/03/07/7-tips-for-securing-the-internet-of-things/>.

YANG, Y. *et al.* A survey on security and privacy issues in internet-of-things. *IEEE Internet of Things Journal*, PP, n. 99, p. 1–1, 2017. ISSN 2327-4662.

ŠVENDA, P. *JCAlgTest: Automated testing tool for algorithms from Java Card API supported by a particular Smart Card*. 2016. Online. Disponível em: <https://github.com/crocs-muni-/JCAlgTest>.

APÊNDICE A – ALGORITMOS SUPORTADOS PELO SMART CARD

ALGORITMO			JAVA CARD
NOME	BYTES	PADDING	
DES CBC-MAC	4	N/A	$\leq$ 2.1
DES CBC-MAC	8	N/A	$\leq$ 2.1
DES CBC-MAC	4	Método 1 (ISO9797-1)	$\leq$ 2.1
DES CBC-MAC	8	Método 1 (ISO9797-1)	$\leq$ 2.1
DES CBC-MAC	4	Método 2 (ISO9797-1)	$\leq$ 2.1
DES CBC-MAC	8	Método 2 (ISO9797-1)	$\leq$ 2.1
AES128 CBC-MAC	16	N/A	2.2.0
DES *Retail MAC* (ISO9797-1)	4	Método 2 (ISO9797-1)	2.2.0
DES *Retail MAC* (ISO9797-1)	8	Método 2 (ISO9797-1)	2.2.0
SEED CBC-MAC	16	N/A	2.2.2

Tabela A.1: Algoritmos de código de autenticação de mensagem

ALGORITMO			JAVA CARD
NOME	HASH	PADDING	
RSA	SHA	ISO9796	$\leq$ 2.1
RSA	SHA	PKCS#1	$\leq$ 2.1
RSA	MD5	PKCS#1	$\leq$ 2.1
RSA	RIPEMD160	ISO9796	$\leq$ 2.1
RSA	RIPEMD160	PKCS#1	$\leq$ 2.1
ECDSA	SHA	N/A	2.2.0

Tabela A.2: Algoritmos de assinatura digital

ALGORITMO			JAVA CARD
NOME	MODO	PADDING	
DES	CBC	N/A	$\leq$ 2.1
DES	CBC	Método 1 (ISO9797-1)	$\leq$ 2.1
DES	CBC	Método 2 (ISO9797-1)	$\leq$ 2.1
DES	ECB	N/A	$\leq$ 2.1
DES	ECB	Método 1 (ISO9797-1)	$\leq$ 2.1
DES	ECB	Método 2 (ISO9797-1)	$\leq$ 2.1
AES128	CBC	N/A	2.2.0
AES128	ECB	N/A	2.2.0
SEED	CBC	N/A	2.2.2
SEED	ECB	N/A	2.2.2

Tabela A.3: Algoritmos de cifragem simétrica

ALGORITMO		JAVA CARD
NOME	PADDING	
RSA	PKCS#1	$\leq$ 2.1
RSA	ISO9796	$\leq$ 2.1
RSA	N/A	2.1.1

Tabela A.4: Algoritmos de cifragem assimétrica

ALGORITMO	JAVA CARD
DH	2.2.1
DH (multiplicação de cofator)	2.2.1

Tabela A.5: Algoritmos de troca de chaves

ALGORITMO	JAVA CARD
SHA	$\leq$ 2.1
MD5	$\leq$ 2.1
RIPEMD160	$\leq$ 2.1
SHA256	2.2.2

Tabela A.6: Algoritmos de *hash*

ALGORITMO		JAVA CARD
NOME	BITS	
RSA	512	$\leq$ 2.1
RSA	736	2.2.0
RSA	768	2.2.0
RSA	896	2.2.0
RSA	1024	$\leq$ 2.1
RSA	1280	2.2.0
RSA	1536	2.2.0
RSA	1984	2.2.0
RSA	2048	$\leq$ 2.1
Curvas Elípticas em F_p	160	2.2.0
Curvas Elípticas em F_p	192	2.2.0
Curvas Elípticas em F_p	224	3.0.1
Curvas Elípticas em F_p	256	3.0.1

Tabela A.7: Algoritmos de geração de chaves assimétricas

ALGORITMO		JAVA CARD
NOME	BITS	
DES	64	$\leq$ 2.1
3DES, 2 Chaves	128	$\leq$ 2.1
3DES, 3 Chaves	192	$\leq$ 2.1
AES128	128	2.2.0
SEED	128	2.2.2

Tabela A.8: Algoritmos de geração de chave simétrica

ALGORITMO	JAVA CARD
CRC16 (ISO3309)	2.2.1
CRC32 (ISO3309)	2.2.1

Tabela A.9: Algoritmos de *checksum*