INSTITUTO FEDERAL DE SANTA CATARINA

PAULO FYLIPPE SELL

Atualização de *Firmware* em Sistemas Embarcados de Forma Segura e Confiável

São José - SC

abril/2021

ATUALIZAÇÃO DE *FIRMWARE* EM SISTEMAS EMBARCADOS DE FORMA SEGURA E CONFIÁVEL

Trabalho de conclusão de curso apresentado à Coordenadoria do Curso de Engenharia de Telecomunicações do campus São José do Instituto Federal de Santa Catarina para a obtenção do diploma de Engenheiro de Telecomunicações.

Orientador: Emerson Ribeiro de Mello, Dr.

Coorientador: Roberto de Matos, Dr.

São José - SC

abril/2021

PAULO FYLIPPE SELL

ATUALIZAÇÃO DE *FIRMWARE* EM SISTEMAS EMBARCADOS DE FORMA SEGURA E CONFIÁVEL

Este trabalho foi julgado adequado para obtenção do título de Engenheiro de Telecomunicações, pelo Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina, e aprovado na sua forma final pela comissão avaliadora abaixo indicada.

São José - SC, 23 de abril de 2021:

Emerson Ribeiro de Mello, Dr., Dr.
Orientador
Instituto Federal de Santa Catarina

Roberto de Matos, Dr., Dr.
Coorientador
Instituto Federal de Santa Catarina

Arliones Stevert Hoeller Junior, Dr.
Instituto Federal de Santa Catarina

Eduardo Augusto Bezerra, Dr.
Universidade Federal de Santa Catarina

Aos meus pais e a minha irmã.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Carlos e Adelir, que nunca mediram esforços para que eu e minha irmã Ana Paula pudéssemos ter nossos estudos como prioridade.

À Julia, que esteve ao ao meu lado durante o período da graduação.

Agradeço aos meus colegas de curso, Marina, Renan e principalmente à Maria, que dividiu comigo inúmeros momentos bons durante a graduação.

Agradeço também meus amigos Ihan, Henrique, Lucas e Felipe, que representam o que uma amizade deve ser.

Por fim, agradeço ao corpo docente do IFSC, em especial aos meus orientadores, Emerson e Roberto, que, mesmo com os percalços que uma pandemia gerou nos nossos trabalhos, foram sempre atenciosos e permitiram que este trabalho pudesse sair do papel.

"Não adentre a boa noite apenas com ternura
A velhice queima e clama ao cair do dia
Fúria, fúria contra a luz que já não fulgura."
(Dylan Thomas.)

RESUMO

Sistemas embarcados estão presentes em grande parte do dia-a-dia das pessoas. Equipamentos que monitoram a saúde, controlam casas e carros inteligentes ou ainda cidades inteiras permitem uma melhor qualidade de vida dos cidadãos. Estes dispositivos precisam manter seus *firmwares* atualizados e seguros, evitando que ataques maliciosos que comprometam seu funcionamento sejam realizados. A atualização, por sua vez, deve ser realizada de tal maneira que seja garantido neste processo que apenas *firmwares* de fontes conhecidas sejam aceitos, bem como garanta que o dispositivo não ficará inutilizável após o processo de atualização. Este trabalho propõe uma solução de atualização de *firmware* em sistemas embarcados microcontrolados, de forma que a atualização seja realizada de forma segura e confiável. A proposta apresenta um organização de classes, na linguagem de programação *C++*, de forma que a implementação da solução de atualização de *firmware* seja portável em diferentes plataformas. Para validar a solução proposta, uma implementação foi realizada utilizando o *kit* de desenvolvimento *STM32L562-DK Discovery*. Um conjunto de experimentos foram conduzidos a fim de verificar se a solução de atualização de *firmware*, bem como a implementação realizada, conseguiria garantir a atualização segura e confiável. Os resultados demonstraram que a solução é capaz de garantir que a atualização seja realizada de forma segura e confiável.

Palavras-chave: Atualização de *firmware*. Confiabilidade. Integridade.

ABSTRACT

Embedded systems are present in a large part of people's daily lives. Devices that monitor health, control smart homes and cars or even entire cities allow for a better quality of life for citizens. These devices need to keep their firmwares up-to-date and safe, preventing malicious attacks that might compromise their regular operation. The update process must occur in a way that it is guaranteed that only firmwares generated from known sources can be installed in the device, as well as guarantee that the device will not be unusable after the update process.This work proposes a firmware update solution in embedded micro-controlled systems, so that the safety and reliability properties are guaranteed. The proposal presents an organization of classes, developed in the *C++*, so that an implementation of the update solution is portable across different platforms. To validate the proposal, an implementation was performed using the *STM32L562-DK Discovery* development kit. A set of experiments were conducted in order to verify whether the *firmware* update solution, as well as the implementation performed, were able to guarantee a safe and reliable update. The results showed that the solution is capable of ensuring that the update is carried out in a safe and reliable manner.

Keywords: Firmware update. Reliability. Integrity.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

Dispositivos que possuem sistemas embarcados compartilham um grande espaço na vida das pessoas. De cafeteiras a *smartphones*, tais equipamentos facilitam o cotidiano e proporcionam uma maior comodidade para as pessoas. Uma vez implantados, esses dispositivos podem apresentar problemas não previstos durante o desenvolvimento, fazendo com que algum mecanismo de atualização de *firmware* que previna ou corrija o mal funcionamento seja projetado, de preferência sem a necessidade de retornar o dispositivo ao fabricante, poupando tempo e dinheiro do consumidor e do fabricante. Além disso, atualizar o *firmware* sem devolver o dispositivo ao fabricante provê uma maior agilidade na melhoria dos equipamentos, uma vez que os dispositivos podem ser atualizados assim que uma nova versão for disponibilizada. (JAIN; MALI; KULKARNI, 2016).

A atualização de *firmware* de um dispositivo pode dar-se de algumas formas. O dispositivo pode detectar que existe uma atualização disponível e aplicar a melhoria de forma automática, como é o caso de alguns *smartphones*. Existem também casos onde o usuário do dispositivo deve verificar se existe um novo *firmware* disponibilizado pelo fabricante do equipamento e ele mesmo realizar o procedimento para o *upgrade*, como ocorre com alguns roteadores domésticos ou placas-mãe de computadores (JAIN; MALI; KULKARNI, 2016). Neste trabalho, o foco será em atualização de *firmware* em dispositivos com sistemas embarcados em microcontroladores e a atualização dependerá de uma ação manual do usuário para transferir o *firmware* para o dispositivo.

A fim de que seja possível atualizar o *firmware* do dispositivo, uma pequena aplicação, chamada *bootloader*, deve ser embarcada no equipamento. O *bootloader* é a aplicação responsável por executar a atualização do *firmware* de um dispositivo, bem como garantir que o mesmo não fique inutilizável em função de uma falha no processo de atualização ou caso o *firmware* do dispositivo esteja corrompido (BENINGO, 2015).

Uma atualização de *firmware* é considerada segura quando a mesma não permite que *firmware* gerados por fontes desconhecidas sejam instalados no dispositivo (NIKOLOV, 2018). Para que isso seja possível, deve-se garantir que o *firmware* recebido é íntegro e autêntico. O uso de criptografia de chave pública permite que fornecedor do *firmware* assine o mesmo utilizando sua chave privada e o envie ao dispositivo que será atualizado, juntamente da assinatura gerada. O dispositivo deverá ter conhecimento prévio da chave pública do emissor do dispositivo e utilizá-la para verificar a autenticidade e integridade do *firmware* durante o processo de atualização.

Já uma atualização confiável é aquela que garante que o dispositivo não ficará inutilizável após uma tentativa de atualização, mesmo se a mesma não for bem-sucedida (NIKOLOV, 2018). Uma atualização confiável visa prevenir que o dispositivo entre em um estado de execução inesperado, recuperando-se de uma eventual tentativa de atualização que tenha falhado ou ainda quando seja detectado que o *firmware* instalado não pode entrar em execução. Para tanto, o *bootloader* deve identificar quando uma tentativa de atualização não foi bem-sucedida, além de verificar, a cada inicialização, se o dispositivo possui um *firmware* instalado e o mesmo está íntegro, isto é, o *firmware* que fora instalado previamente não foi alterado ou corrompido. A verificação da integridade do *firmware* do dispositivo pode ser realizada a partir do uso de assinaturas digitais, além de também ser possível com o uso de funções de *hash*, que são funções de embaralhamento que geram saídas únicas para cada entrada (STALLINGS, 2008). Deve-se comparar a saída da função de *hash* que teve como entrada o *firmware* instalado no dispositivo com um valor de *hash* armazenado durante o processo de atualização e, caso os valores sejam diferentes, o dispositivo não deve

colocar o *firmware* em execução, fazendo com que uma nova atualização seja necessária.

Fabricantes de microcontroladores disponibilizam soluções de atualização de *firmware* e *bootloader* para serem utilizadas por desenvolvedores (STMICROELECTRONICS, 2020c; STMICROELECTRONICS, 2020b). Entretanto, essas soluções são limitadas a famílias específicas de microcontroladores, de forma que uma nova implementação de *bootloader* tem que ser realizada para cada microcontrolador diferente utilizado. Uma solução de atualização de *firmware* genérica, como a que será proposta nesse trabalho, permite que fabricantes de dispositivos possam utilizar uma mesma implementação de *bootloader* ao mesmo tempo que os microcontroladores diferentes sejam utilizados.

1.1 Objetivo Geral

O objetivo deste trabalho é propor um modelo de atualização de *firmware*, de modo que o mesmo modelo possa ser utilizado por diferentes soluções de sistemas embarcados que precisem realizar a atualização de seu *firmware*, e possibilitando que esta atualização seja realizada de forma segura e confiável.

1.2 Objetivos específicos

Para concluir o objetivo geral, alguns objetivos específicos foram definidos:

1. Definir os requisitos de *hardware* e *software* que permitam que uma atualização de *firmware* de forma segura e confiável seja realizada;
2. Disponibilizar um modelo de *bootloader* que permita que a solução de atualização de *firmware* seja portável entre diferentes plataformas;
3. Desenvolver uma prova de conceito da solução proposta utilizando o mesmo modelo de *bootloader* disponibilizado.

1.3 Organização do texto

Este trabalho está dividido da seguinte maneira: O Capítulo 2 aborda os assuntos necessários para o entendimento e desenvolvimento deste trabalho. Já o Capítulo 3 apresenta a proposta de solução de atualização de *firmware*. No Capítulo 4 são apresentados a implementação da proposta de solução de atualização e os experimentos realizados para validar a solução e a implementação. Por fim, no Capítulo 5 estão presentes as conclusões deste trabalho, bem como a sugestão de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda os tópicos julgados relevantes para o desenvolvimento deste trabalho. A seção 2.1 aborda as propriedades básicas de segurança. Já na seção 2.2 são abordadas as criptografias simétricas e assimétricas. A seção 2.3, por sua vez, comenta sobre alguns ambientes seguros de execução baseados em *hardware* e *software*. Por fim, a seção 2.4 aborda alguns pontos necessários para a atualização de *firmware* de dispositivos com sistemas embarcados.

2.1 Propriedades básicas de segurança

Segundo Mello et al. (2006), segurança é um serviço que garante que um sistema permaneça em execução mesmo quando ações não previstas aconteçam ou usuários não autorizados tentem acessar o sistema, sem que ocorra violação de segurança. De acordo com Russel e Gangemi (1991), existem três diferentes propriedades que sustentam a segurança: confidencialidade, integridade e disponibilidade.

Uma informação não deve ser exposta a usuários que não possuem acesso à mesma. **Confidencialidade** é a garantia de que dados sensíveis sejam mantidos seguros, permitindo que apenas usuários autorizados possam interpretar a informação. Sem a confidencialidade, segredos industriais ou até militares poderiam ser obtidos (RUSSEL; GANGEMI, 1991). A confidencialidade é, portanto, um pilar importante da segurança computacional.

Informações não devem ser corrompidas ou alteradas de forma não intencional. A **integridade** garante que o sistema não irá permitir nenhuma modificação não autorizada da informação, seja de forma maliciosa ou acidental (RUSSEL; GANGEMI, 1991). Entretanto, caso haja uma modificação da informação, o sistema deve possibilitar a identificação dessa mudança (LEE, 2013).

Um sistema deve estar disponível sempre que requisitado. Desta forma, a **disponibilidade** diz que o acesso de usuários autorizados a um sistema não deve ser negado, mesmo de forma maliciosa (MELLO et al., 2006). Segundo Russel e Gangemi (1991), quando necessário, o sistema deve ser capaz de recuperar seu funcionamento padrão.

Além das já citadas, Landwehr (2001) aborda outras duas propriedades de segurança: autenticidade e não-repúdio. **Autenticidade** é a garantia de que um usuário é quem ele diz ser. Já o **não-repúdio** traz a garantia de que um usuário não poderá afirmar que o mesmo não participou de uma transação.

Apesar das propriedades de segurança, ataques de usuários maliciosos podem acontecer, a partir de vulnerabilidades presentes nos sistemas. Bishop e Bailey (1996) definem vulnerabilidade como a caracterização de um estado não autorizado do sistema que pode ser alcançado a partir de um estado autorizado, comprometendo seu funcionamento. De acordo com Shirey (2007), um ataque é um ato intencional o qual um usuário tenta violar as políticas de segurança de um sistema, comprometendo assim as propriedades básicas de segurança. Estes podem ser classificados em duas categorias: ataques passivos e ativos.

Ataques passivos têm a intenção de visualizar a troca de informação entre os pares, sem fazer alguma alteração nas mensagens (STALLINGS, 2008). Um ataque passivo pode acontecer quando uma terceira entidade intercepta uma troca de mensagens e tem acesso à informação obtida.

De acordo com Stallings (2008), ataques ativos fazem modificações nas mensagens interceptadas, permitindo que o atacante tenha o controle da troca de mensagens. Ainda de acordo com Stallings (2008),

ataques ativos podem ser classificados em quatro categorias: disfarce, repetição, modificação de mensagens e negação de serviço.

- Disfarce: Ocorre quando o atacante se faz passar por uma entidade diferente, ou seja, um dos pares comunicantes acha que está conversando com alguém confiável quando na verdade está falando com o atacante;
- Repetição: Quando o atacante captura uma troca de mensagens e posteriormente repassa a mensagem ao receptor, na tentativa de causar algum efeito diferente do esperado pelo sistema;
- Modificação de mensagens: É a alteração de uma mensagem interceptada entre os pares comunicantes, comprometendo a integridade da troca de informações. Ou seja, o par receptor pode receber uma mensagem completamente diferente do que o transmissor enviou, de acordo com as intenções do atacante;
- Negação de serviço: É o impedimento do uso normal das instalações de comunicação, como quando o atacante intercepta todas as mensagens do transmissor e não faz o repasse ao receptor.

2.2 Criptografia

De acordo com Stallings (2008), criptografia é o mapeamento de uma informação (letras, *bits*) em um outro símbolo, de modo que o mapeamento inverso só possa ser feito por um usuário que tenha informações suficientes para realizar a descriptografia. Ao criptografar um dado, é importante que nenhuma informação seja perdida, fazendo com que a mensagem criptografada possa ser revertida.

Se a mesma chave criptográfica for utilizada na transmissão e recepção de uma informação, para criptografar e descriptografar a mesma, respectivamente, este sistema é chamado de criptografia simétrica (ou de chave única). Caso chaves diferentes sejam utilizadas na criptografia, este sistema é chamado de criptografia assimétrica (ou de chave pública).

Para o uso correto de criptografia, é necessário que o algoritmo utilizado seja robusto. Na prática, mesmo que o atacante possua várias mensagens criptografadas, ele não deve ser capaz de obter a informação original a partir de padrões característicos do algoritmo utilizado. A informação só deverá ser obtida a partir do uso da chave utilizada durante a cifragem da informação (no caso da criptografia simétrica). Desta forma, não é necessário omitir a informação do algoritmo utilizado. Faz-se necessário apenas proteger a chave criptográfica (STALLINGS, 2008).

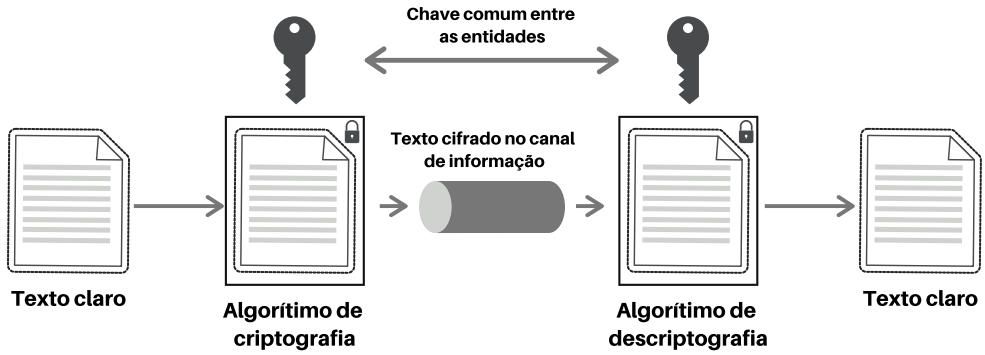
2.2.1 Criptografia simétrica

A criptografia simétrica é aquela onde apenas uma chave é utilizada para criptografar e descriptografar a informação. Desta maneira, a informação é criptografada a partir do algoritmo de criptografia e da chave secreta. A Figura 1 apresenta o modelo de criptografia simétrica. O algoritmo criptográfico gera saídas diferentes para cada entrada (chave e texto claro) recebida. Portanto, para obter a informação original novamente, é necessário ter informação da chave utilizada (STALLINGS, 2008).

2.2.2 Criptografia assimétrica

Diferentemente da criptografia simétrica, na criptografia assimétrica (ou de chave pública) existem duas chaves distintas para criptografar e descriptografar uma informação: a chave pública e a chave privada (LEE, 2013). A chave privada deve ser protegida pelo usuário e nunca compartilhada. Já a chave pública deverá ser distribuída entre os outros usuários. Com a criptografia de chave pública é possível

Figura 1 – Modelo de criptografia simétrica



Fonte: Adaptada de (STALLINGS, 2008).

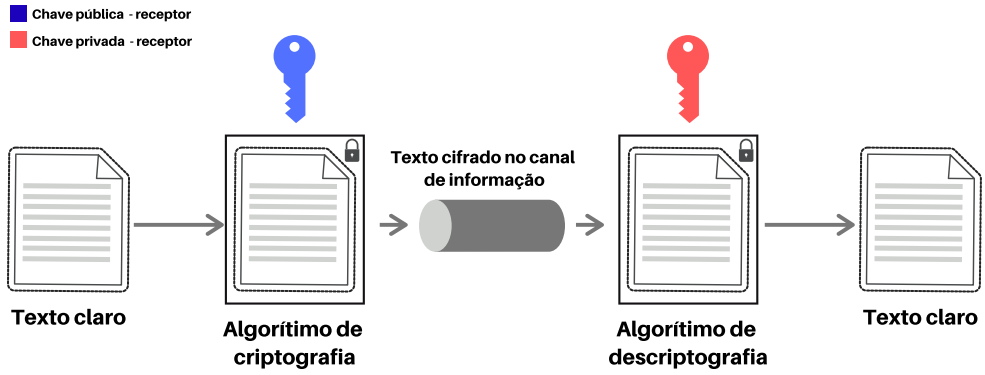
garantir a confidencialidade da informação. A confidencialidade é atingida ao cifrar uma informação com a **chave pública** da entidade destino. Desta forma, é garantido que apenas a chave privada par da chave pública utilizada será capaz de descriptografar a mensagem. Este processo pode ser conferido na Figura 2.

A partir do uso de chaves privadas, desenvolveu-se as assinaturas digitais. Assinaturas digitais são criadas com o uso de funções de *hash*, que possuem como entrada apenas a mensagem a ser embaralhada, podendo essa ser de tamanho variado. Portanto, não existe uma chave criptográfica nesta função. Sua saída retorna um valor (código de *hash*) de tamanho fixo e não previsível, isto é, uma pequena mudança na mensagem pode gerar um valor totalmente diferente no código de *hash* (STALLINGS, 2008). A função *hash* é irreversível, não existindo possibilidade de identificar a mensagem que gerou um código *hash* (JOHNER et al., 2000).

A assinatura digital é o processo de criptografar o código *hash* gerado de uma função *hash*, utilizando a chave privada do emissor. Esta assinatura é então anexada à mensagem a ser transmitida, ficando a cargo do receptor gerar um código *hash* da mensagem recebida, descriptografar a assinatura digital anexada à mensagem com a chave pública do emissor e comparar os dois códigos *hash* obtidos (JOHNER et al., 2000). Enquanto as chaves privadas permanecerem seguras, um usuário não poderá afirmar que o mesmo não assinou uma mensagem outrora assinada por ele, garantindo assim a propriedade do não-repúdio (LEE, 2013). A Figura 3 descreve o processo de assinatura digital de uma mensagem.

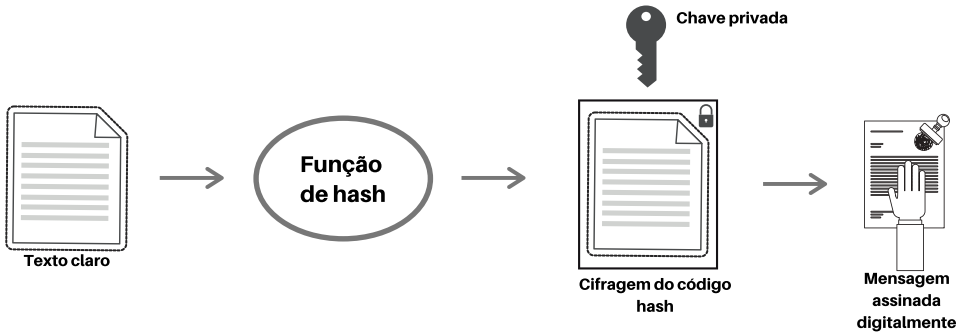
Apesar de garantir a integridade da informação recebida, o uso de funções de *hash* e assinaturas

Figura 2 – Modelo de criptografia assimétrica



Fonte: Própria.

Figura 3 – Processo de assinatura digital



Fonte: Adaptada de (JOHNER et al., 2000)

digitais não garantem a autenticidade do emissor, uma vez que um atacante pode obter uma chave privada de uma entidade e assumir seu papel em uma troca de informações. Para isto, os certificados digitais foram criados.

Certificado digital é um mecanismo utilizado para prover a autenticidade de uma entidade. Um certificado digital contêm a chave pública de uma entidade e algumas outras informações que possam ajudar a identificar um usuário. Todo certificado digital deve ser assinado por uma entidade, chamada de autoridade certificadora, que é responsável por atestar, utilizando assinaturas digitais, as identidades dos usuários (JOHNER et al., 2000).

Quando um usuário deseja compartilhar sua chave pública ou trocar informação com outro par, é anexado à mensagem seu certificado digital recebido pela autoridade certificadora. Como esta entidade é confiável por ambas as partes da comunicação, o segundo usuário aceita a chave pública do emissor e ambos podem trocar informações de forma confiável (JOHNER et al., 2000).

Um dos padrões mais utilizados para a criação de certificados digitais é o formato sugerido na recomendação X.509 da *Telecommunication Standardization Sector* (ITU-T). Segundo Stallings (2008), a recomendação X.509 tem como base o uso de criptografia de chave pública e assinaturas digitais e prevê no seu modelo de certificado digital onze campos distintos, descritos na Tabela 1.

Tabela 1 – Campos de um certificado digital - recomendação X.509

Campo	Descrição
Versão	Variante sendo utilizada da recomendação X.509
Número de série	Um valor inteiro que distingue cada certificado
Identificador do algorítimo	O algoritmo utilizado para assinar digitalmente o certificado
Nome do emissor	Este campo informa o nome da autoridade certificadora que assinou o certificado
Período de validade	As datas de início e fim do período de abrangência do certificado
Nome do titular	O nome do usuário ao qual o certificado digital se refere
Informação da chave pública do titular	É o campo aonde a chave pública do titular do certificado é armazenada
Identificador exclusivo do emissor	Esta identificação distingue autoridades certificadoras que possuem o mesmo nome
Identificador exclusivo do titular	Esta identificação distingue usuários que possuem o mesmo nome.
Extensões	Informações a respeito do certificado, como políticas em que o certificado se aplica e nomes alternativos de emissor e titular
Assinatura	A assinatura contêm os códigos *hash* dos outros campos do certificado, criptografados com a chave privada da autoridade certificadora

Fonte: Própria

Tabela 2 – Comparação entre os tipos de criptografia e as propriedades básicas de segurança

Propriedade	Criptografia simétrica	Criptografia assimétrica
Confidencialidade	√	√
Integridade		√
Autenticidade		√
Não-repúdio		√
Disponibilidade	não se aplica	não se aplica

Fonte: Própria

2.2.3 Comparativo entre criptografia simétrica e assimétrica

A fim de comparação, a Tabela 2 apresenta quais propriedades básicas de segurança cada tipo de criptografia atende. Ambas as criptografias atendem a propriedade de confidencialidade, uma vez que a informação é cifrada utilizando algoritmos e chaves criptográficas. Entretanto, apenas a criptografia assimétrica atende a integridade e o não-repúdio, utilizando a assinatura digital, e a autenticidade, utilizando os certificados digitais.

Também é possível fazer um comparativo entre os tipos de criptografia e os tipos de ataques descritos na seção 2.1. Uma vez criptografada, a mensagem só pode ser lida pelo usuário que possuir a chave descriptográfica, podendo ser a mesma chave utilizada para criptografar a informação ou não, no caso do uso da criptografia simétrica e assimétrica, respectivamente. Já a modificação só pode ser detectada utilizando assinatura digital, mecanismo que apenas a criptografia assimétrica possui. O disfarce pode ser evitado fazendo o uso de certificados digitais, outro mecanismo que apenas a criptografia de chave pública provê.

Tabela 3 – Comparação entre os tipos de criptografia e ataques

Tipos de ataque	Criptografia simétrica	Criptografia assimétrica
Ataques passivos	√	√
Disfarce		√
Modificação		√
Negação de serviço	não se aplica	não se aplica
Repetição	não se aplica	não se aplica

Fonte: Própria

2.3 Ambientes seguros de execução

Ambientes seguros de execução são aqueles que proveem armazenamento e execução seguros de aplicações e dados, prevenindo que ataques visando obter ou modificar dados de forma não autorizada sejam bem-sucedidos. Os mesmos podem ser desenvolvidos baseados tanto em *software* quanto em *hardware*, o último sendo o foco deste trabalho. De acordo com Bouazzouni, Conchon e Peyrard (2018), as características que definem um ambiente seguro de execução são atingidas baseadas nas seguintes premissas:

- Execução isolada: Toda aplicação sensível deve ser executada de forma isolada. Desta forma, aplicações maliciosas não possuem acesso a dados manipulados por uma aplicação ou ao seu código em execução;

- Armazenamento seguro: A integridade e confidencialidade de uma aplicação deve ser mantida. Além disso, deve-se também manter chaves criptográficas, certificados e senhas armazenados de forma segura;
- Adaptação segura: Deve-se prover a integridade e confidencialidade de dados trocados entre aplicações, como chaves criptográficas e certificados.

Dentre as soluções baseadas em *hardware* para alcançar as propriedades de segurança de ambientes seguros de execução, destacam-se três: *Trusted Platform Module* (TPM), *Secure Element* (SE) e *Trusted Execution Environment* (TEE). Estas soluções serão detalhadas nas subseções seguintes.

2.3.1 Trusted Platform Module

Trusted Platform Module (TPM) é um microcontrolador seguro capaz de executar funções criptográficas complexas (BOUAZZOUNI; CONCHON; PEYRARD, 2018). O TPM conecta-se a um outro dispositivo, por exemplo a placa-mãe de um computador de mesa, atuando de forma externa e isolada a mesma, possuindo como uma de suas principais funções a garantia de integridade do dispositivo o qual o TPM está acoplado, guardando informações que podem ser utilizadas, por exemplo, para prover uma inicialização íntegra e segura do sistema (NYMAN; EKBERG; ASOKAN, 2014). As informações entre o TPM e o dispositivo em que ele se conecta-se podem ser trocadas a partir de uma *Application Programming Interface* (API) que o próprio TPM fornece. Segundo Bouazzouni, Conchon e Peyrard (2018), o TPM possui quatro elementos principais:

- Chave de endosso: Consiste em um par de chaves pública e privada. A chave privada é gerada dentro do TPM e nunca utilizada externamente;
- Chaves de atestado de identidade: Estas chaves tem o propósito de atestar a autenticação com um provedor de serviço.;
- Certificados: O TPM guarda três certificados:
  - Certificado de endosso: Este certificado tem a função de garantir a integridade da chave de endosso;
  - Certificado de plataforma: Este certificado é provido pelo fornecedor do módulo e tem a função de garantir que todos os componentes de segurança do TPM são genuínos;
  - Certificado de conformidade: Este certificado pode ser provido por uma terceira parte ou pelo próprio fornecedor do TPM e tem a responsabilidade de garantir que o TPM realmente possui todas as propriedades de segurança que o fabricante informa que o módulo possui.
- Registradores de configuração da plataforma: Estes registradores são utilizados para guardar informações de vínculo entre dados sensíveis a dispositivos ou aplicações.

2.3.2 Secure Element

Secure Element (SE) é um dispositivo inviolável que provê um alto nível de segurança, capaz de armazenar dados sensíveis. Diferentemente do TPM, o SE consegue executar pequenos *software* embarcados no dispositivo, e não apenas operações criptográficas (SHEPHERD et al., 2016). Bouazzouni, Conchon e Peyrard (2018) descrevem três tipos de SE: SE embarcado, *Universal Integrated Circuit Card* (UICC) SE e micro SD SE:

- SE embarcado: Neste tipo de solução, o elemento seguro é soldado diretamente com a plataforma onde o mesmo irá atuar. Não existe um padrão a respeito dos direitos de acesso ao SE, cabendo ao fabricante do dispositivo definir que tipo de aplicação os desenvolvedores podem ou não instalar no elemento seguro;
- UICC SE: Estes elementos seguros são amplamente utilizados por empresas de telecomunicações, na forma de um cartão que permite a autenticação em suas respectivas redes. Esta solução prevê certa mobilidade, uma vez que estes elementos seguros não estão soldados junto à um *host*, podendo ser conectado em dispositivos diferentes;
- Micro SD SE: Este tipo de elemento seguro também provê mobilidade para o usuário, uma vez que o mesmo não é soldado na plataforma que irá utilizá-lo. Entretanto, este SE possui o mesmo nível de segurança que os anteriores. Em contraponto com UICC, o Micro SD possui um maior espaço de armazenamento (REVEILHAC; PASQUET, 2009).

2.3.3 Trusted Execution Environment

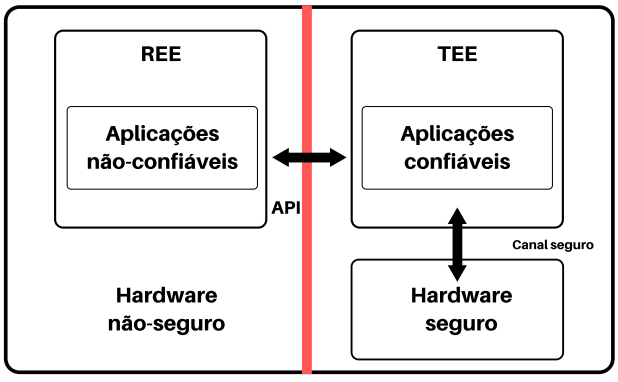
O TEE é um ambiente que possui seu próprio sistema operacional, chamado de sistema operacional seguro. A GlobalPlatform - associação entre empresas para padronização de tecnologias para computação segura (GLOBALPLATFORM, 2018) - especifica o TEE em *software* e *hardware*.

A arquitetura de *hardware* de um sistema que possui um TEE é composta por dois ambientes de execução distintos: *Rich Execution Environment* (REE) e TEE. Estes dois ambientes operam de forma isolada e completamente independentes entre si (ARFAOUI; GHAROUT; TRAORé, 2014).

O REE é o ambiente padrão de um dispositivo, onde aplicações que não necessitam de altos níveis de segurança são executadas, possibilitando uma menor complexidade de implementação. Desta maneira, as aplicações que são executadas no REE não possuem acesso direto às aplicações seguras que estão sendo executadas no outro ambiente (BOUAZZOUNI; CONCHON; PEYRARD, 2018).

O TEE, por sua vez, é um ambiente seguro onde aplicações que precisam de altos níveis de segurança são executadas. O TEE possui um sistema operacional seguro, que é responsável por executar operações sensíveis como funções criptográficas ou capturar dados de entrada de periféricos confiáveis. As aplicações executadas no TEE são chamadas de *Trusted Application* (TA) e a única maneira de aplicações executadas no REE terem acesso às aplicações seguras é a partir de interfaces de aplicação (API) disponibilizadas pelo TEE (BOUAZZOUNI; CONCHON; PEYRARD, 2018). A Figura 4 ilustra de forma simplificada a arquitetura de *software* e *hardware* de um TEE.

Figura 4 – Arquitetura simplificada de um TEE



Fonte: Adaptada de (ARFAOUI; GHAROUT; TRAORé, 2014)

2.3.4 Comparação entre os ambientes seguros de execução

Bouazzouni, Conchon e Peyrard (2018) trazem um comparativo sobre as tecnologias de ambientes seguros de execução, que pode ser observado na Tabela 4. É possível atestar que o TEE possui mais limitações físicas quanto a segurança, uma vez que o mesmo não é fisicamente inviolável. Entretanto, o TEE consegue estabelecer um canal de comunicação seguro entre a entrada de dados do dispositivo e as aplicações que estão sendo executadas no mesmo, além de prover maiores capacidades de processamento e armazenamento.

Tabela 4 – Comparação entre tecnologias de ambientes seguros de execução

Critério	TPM	SE	TEE
Resistência a violações	√	√	
Entrada segura de dados			√
Alta capacidade de processamento	√		√
Alta capacidade de armazenamento			√
Dependência do fabricante	√	√	√

Fonte: Adaptado de (BOUAZZOUNI; CONCHON; PEYRARD, 2018)

2.4 Atualização de *firmware*

Por inúmeras razões, durante seu ciclo de vida, um dispositivo com um sistema embarcado poderá precisar em algum momento atualizar o seu *software* embarcado, também chamado de *firmware*. Desta forma, faz-se necessário desenvolver mecanismos para que seja possível atualizar o *firmware* de um dispositivo após sua implantação, de preferência de forma remota. Jain, Mali e Kulkarni (2016) defendem que a possibilidade de atualizar o *firmware* de um dispositivo de forma remota traz consigo alguns benefícios, como evitar que o usuário retorne o equipamento ao fabricante para fazer a atualização, bem como facilita e acelera a atualização dos vários dispositivos implantados, uma vez que eles podem ser atualizados no momento em que o fabricante disponibilize o novo código.

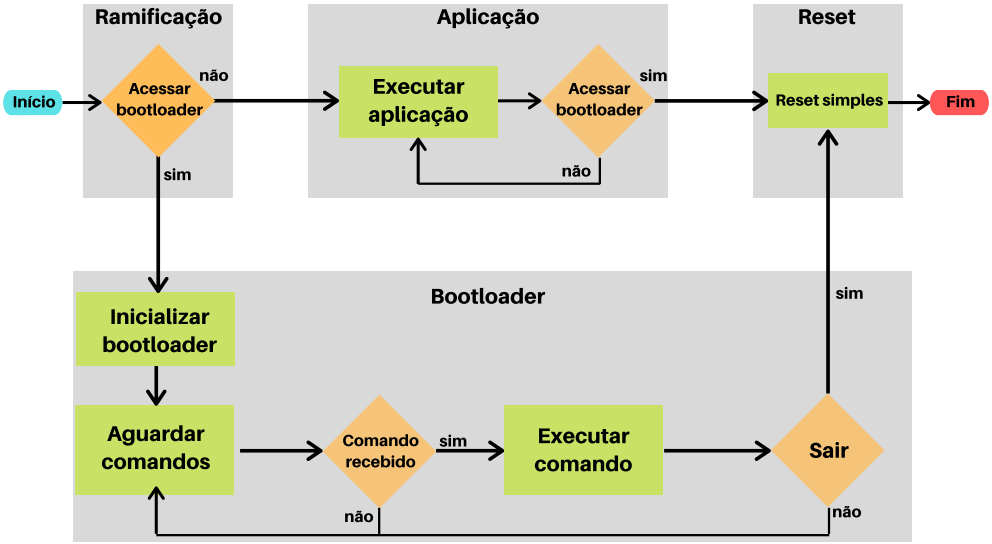
Para que a atualização de um dispositivo possa ser realizada sem a necessidade de um *hardware* externo, uma pequena aplicação - *bootloader* - deve ser embarcada no equipamento. O *bootloader* é uma aplicação que compartilha espaço da memória *flash* de um microcontrolador com o *firmware*, sendo capaz de apagar e escrever novos códigos na área de memória compartilhada, bem como verificar a integridade de um *firmware* (BENINGO, 2015).

O *bootloader* possui dois comportamentos padrão. No primeiro, o processo de execução do *bootloader* é feita de forma totalmente automática e isolada do restante do sistema, o que significa dizer que a detecção de um novo *firmware* e sua posterior instalação é feita de forma autônoma. No segundo modelo, o sistema não executa o *bootloader* de forma autônoma. Ao invés disto, o sistema entra em um estado ocioso e aguardará os comando para a atualização do *firmware* de uma fonte externa ao sistema (JAIN; MALI; KULKARNI, 2016).

Beningo (2015) afirma que cada projeto de *bootloader* irá possuir seus requisitos particulares necessários. Todavia, existem alguns requisitos fundamentais que todo projeto de *bootloader* deve possuir:

1. Habilidade de selecionar os modos de operação do dispositivo (*bootloader* ou aplicação);
2. Interfaces de comunicação com outros dispositivos;
3. Padrão definido quanto ao formato de arquivo do código do *firmware*;

Figura 5 – Componentes de um sistema com *bootloader*



Fonte: Adaptada de (BENINGO, 2015)

4. Possibilidade de manipulação das memórias do sistema;
5. Verificação de integridade do *firmware*;
6. Armazenamento do *bootloader* de forma que seu código não possa ser modificado;

Em geral, um sistema embarcado deve possuir três componentes principais: o código de ramificação, a aplicação (*firmware*) e o *bootloader*. O código de ramificação é responsável por informar se o sistema deve executar a aplicação do sistema ou o *bootloader*. A aplicação é executada quando o código de ramificação define que o sistema não deve executar o *bootloader*. Em certos casos, como quando a integridade da aplicação deve ser verificada na inicialização do dispositivo, o código de ramificação e o *bootloader* são uma coisa só. Deve existir também um bloco de reinicialização do sistema (BENINGO, 2015). A Figura 5 ilustra os componentes do sistema, juntamente com um comportamento padrão do mesmo.

Quando selecionado, o *bootloader* entrará em execução, devendo prover suporte para pelo menos três comandos:

- Remover o *firmware* da memória *flash*;
- Escrever na memória *flash* um novo *firmware*;
- Sair do *bootloader* e carregar o *firmware* instalado.

Figura 6 – Processo padrão de atualização de *firmware*

Fonte: Adaptada de (BENINGO, 2015)

Alguns outros comandos devem ser previstos, de acordo com os requisitos fundamentais já descritos, a fim de melhorar as capacidades de atualização de *firmware* do *bootloader*, tais como bloquear e desbloquear a manipulação da memória *flash* e gerar um valor a partir de uma função de *hash* que garanta a integridade do *firmware* instalado. Beningo (2015) descreve uma sequência padrão para a atualização de *firmware* de um sistema, que pode ser observada na Figura 6.

3 UMA SOLUÇÃO DE ATUALIZAÇÃO DE *FIRMWARE*

A solução de atualização de *firmware* proposta neste trabalho é destinada a sistemas embarcados compostos por componentes com recursos de armazenamento e processamento limitados (*e.g.* microcontroladores) e com função específica. O processo de atualização proposto é do tipo passivo, no qual a transferência do novo *firmware* deverá ser iniciada por um agente externo. Uma visão geral da solução é apresentada na Figura 7 e onde podemos identificar os seguintes elementos principais: i) emissor do *firmware*; ii) binário; ii) dispositivo alvo.

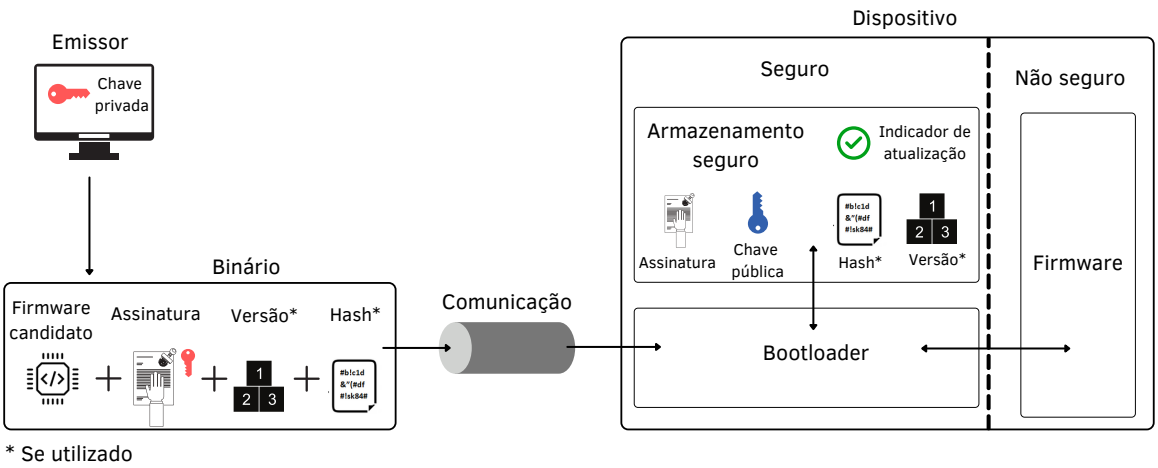
A proposta faz uso de criptografia de chave pública, conforme discutido na seção 2.2, para garantir que apenas *firmware* autênticos sejam instalados no dispositivo. O detentor da chave privada é o emissor, que deve mantê-la em segurança para garantia do processo, enquanto a chave pública é embarcada no dispositivo alvo de forma segura, em tempo de manufatura, e não deve ser alterada, salvo nos cenários de atualização onde isso é previsto. Neste trabalho não é proposto uma maneira de alterar a chave pública do emissor. Uma assinatura digital válida é a garantia de que a atualização será realizada de forma segura e que o dispositivo só aceitará um novo *firmware*, chamado de *firmware* candidato, assinado pelo emissor.

O binário é a definição da proposta para o pacote que contém o *firmware* candidato e as informações essenciais que serão utilizadas nos processos de atualização e, futuramente, de inicialização. Essas informações essenciais são a assinatura do emissor e, dependendo dos requisitos do cenário, a versão e o resumo criptográfico do candidato. O binário é enviado ao dispositivo alvo por algum meio de comunicação, o qual é abstraído na solução, podendo ser implementado por qualquer tecnologia disponível.

O dispositivo alvo, por sua vez, deve ser composto por três componentes principais: i) *bootloader*, responsável por gerenciar o processo de atualização e inicialização do *firmware*; ii) armazenamento seguro, no qual as informações essenciais devem ser gravadas; iii) *firmware*, que será substituído no processo de atualização.

O *bootloader* é descrito em detalhes na seção 3.1. Ele deve ser o primeiro código executado no

Figura 7 – Componentes da solução de atualização de *firmware*.



Fonte: Própria.

dispositivo, sendo responsável por identificar e controlar o processo de atualização. Além disso, o *bootloader* deve garantir a confiabilidade desse processo, permitindo o seu reinício quantas vezes forem necessárias e verificando a integridade do *firmware* antes de colocá-lo em execução. Dessa forma, o dispositivo nunca entrará em um estado desconhecido, não sendo inutilizado por uma atualização malsucedida.

Para tal, três prerrogativas devem ser garantidas pelo *hardware* para o perfeito funcionamento do *bootloader*. A primeira é o armazenamento seguro, para guardar as informações essenciais que o *bootloader* irá utilizar durante os processos de atualização e inicialização do dispositivo. Esse armazenamento deve ser de acesso exclusivo do *bootloader*. A segunda é a garantia de imutabilidade do *bootloader*, isto é, uma vez que ele tenha sido inserido no dispositivo, não deve ser alterado. Finalmente, a terceira é a execução prioritária do *bootloader*, ou seja, ele sempre deve ser o primeiro código a entrar em execução a cada inicialização.

As duas últimas prerrogativas são estáticas e mais simples de garantir. Entretanto, garantir exclusividade de leitura e escrita ao armazenamento seguro é uma tarefa para *hardware* seguros, como os citados na seção 2.3. É possível utilizar, por exemplo, um SE em conjunto com um microcontrolador, de forma que o *bootloader* executado no microcontrolador busque as informações essenciais no elemento seguro sempre que necessário. Uma outra organização possível é aquela na qual o TEE é utilizado para armazenar as informações essenciais e, sendo possível ainda, armazenar o próprio *bootloader* no ambiente seguro do TEE. Neste cenário, tanto o *bootloader* quanto as informações utilizadas para o processo de atualização são armazenadas de forma isolada ao *firmware* do dispositivo.

Além da garantia que somente um *firmware* assinado seja instalado, opcionalmente pode ser desejado que um *firmware* antigo, mesmo assinado, não seja reinstalado em um dispositivo. Para isso, a proposta prevê o uso da informação de versão que pode ser incluída no binário, de forma que o valor da versão seja sempre crescente. Nesses casos, em cada tentativa de atualização, o *bootloader* deve comparar a informação da versão contida no binário com a versão do *firmware* instalado, de modo a impedir que o candidato seja instalado no dispositivo caso sua versão seja inferior.

A verificação da versão por si só não garante que um *firmware* antigo não seja reinstalado no dispositivo. Um atacante poderia alterar apenas a versão contida no binário para uma versão superior ao *firmware* instalado e transmiti-lo para o dispositivo. Para evitar tal ataque, a versão deve ser utilizada como entrada da função que gera a assinatura digital. Com isso, o emissor consegue garantir que a assinatura só é válida para uma versão específica de um determinado candidato. O uso da versão do *firmware* durante o processo de atualização não é requisito para que a mesma seja considerada segura, porém quando for utilizada, essa informação também deve ser de acesso exclusivo do *bootloader* e armazenada de forma segura.

O *firmware* do dispositivo deve ser armazenado em uma região de memória na qual o *bootloader* possa ler e escrever. Essa região, que a partir deste ponto será chamada de partição, deve ser definida de acordo com a quantidade de memória que o dispositivo possui. Nos casos em que apenas uma partição pode ser definida, o *bootlodaer* deve sobrescrever o *firmware* do dispositivo em cada tentativa de atualização, armazenando o candidato nessa partição. Caso a atualização falhe, o *bootloader* não deve colocar o candidato em execução, demandando uma nova atualização.

Já nos casos que duas partições podem ser definidas, pode-se determinar uma partição de execução atual na qual o *firmware* do dispositivo está armazenado, e outra onde o candidato é recebido durante o processo de atualização enquanto o *bootloader* não realiza as verificações necessárias para validá-lo. Uma vez que o candidato tenha sido validado, o *bootloader* pode transferi-lo para a partição de execução, no caso da mesma ser fixa, ou simplesmente chavear entre as partições e apagar o *firmware* antigo.

Na próxima seção os componentes e comportamento do *bootloader* são detalhados.

3.1 Modelo de *bootloader*

O *bootloader* pode identificar a necessidade de atualização de duas maneiras. A primeira é quando o *bootloader* do dispositivo verifica que o *firmware* instalado não está íntegro, de forma que o dispositivo tenha que ser atualizado para que possa voltar a funcionar de forma adequada. A segunda maneira é nos casos em que um agente externo (*e.g* o usuário do dispositivo) faz a solicitação atualização de forma manual. Nesses cenários, deve ser possível avisar ao *bootloader* de que a atualização é necessária, fazendo com que a execução do *firmware* do dispositivo seja interrompida e o *bootloader* volte a ser executado. Esse aviso ao *bootloader* pode ser feito a partir de uma reinicialização do dispositivo, seguido de um acionamento de um botão, de forma que o *bootloader* identifique esse comportamento como sendo o aviso de que a atualização é necessária. Uma vez que o *bootloader* tenha identificado que a atualização é necessária, ele deve armazenar essa informação em um ambiente de armazenamento seguro, e essa indicação só deve ser desativada após uma atualização bem-sucedida.

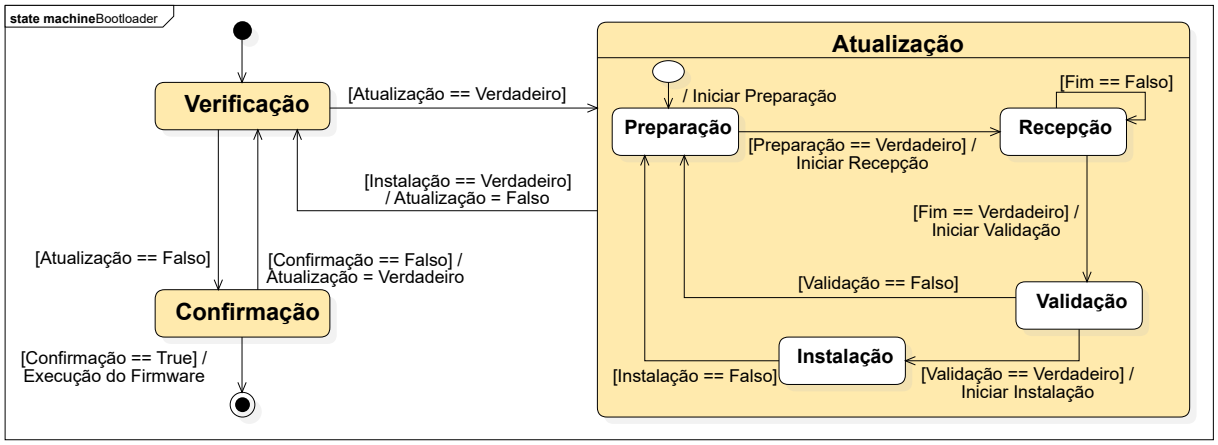
Como falado na seção anterior, o *bootloader* deve utilizar a assinatura digital do candidato no processo de atualização para verificar sua autenticidade, de forma a permitir que apenas um *firmware* autêntico seja instalado. A assinatura digital também pode ser utilizada para realizar a verificação de integridade do *firmware* instalado no dispositivo. Para isso, a assinatura digital recebida deve ser armazenada em um ambiente seguro, após a verificação da autenticidade do candidato.

Além da assinatura digital, é possível utilizar o resumo criptográfico do *firmware* para a verificação de integridade do mesmo. Nesse cenário, o resumo criptográfico pode ter sido transmitido ao dispositivo ou pode ter sido gerado pelo *bootloader*, após a verificação de autenticidade do candidato recebido. Caso o resumo criptográfico seja utilizado para a verificação de integridade do *firmware* instalado, deve-se armazená-lo no ambiente seguro.

A Figura 8 apresenta o diagrama da *Finite State Machine* (FSM) em UML (OMG, 2017) do *bootloader* proposto. A FSM do *bootloader* é dividida em três estados distintos: Verificação, Confirmação e Atualização, sendo que o estado Atualização é dividido em quatro sub-estados, sendo eles Preparação, Recepção, Validação e Instalação.

No estado inicial, Verificação, o *bootloader* precisa identificar se uma tentativa de atualização fora requisitada previamente. Essa verificação deve ser feita a partir do indicador de atualização. Caso o

Figura 8 – Máquina de estados finitos UML para o *bootloader*.



Fonte: Própria.

indicador de atualização esteja ativado, o *bootloader* deve então ir para o estado Atualização.

É no estado Atualização que o binário será recebido, verificado e o *firmware* candidato instalado. No sub-estado Preparação, o *bootloader* deve preparar o dispositivo para a recepção do binário, que contém o candidato e as informações essenciais para o processo de atualização. Essa preparação pode variar de sistema para sistema, mas ações comuns nesse estado estão relacionadas com a limpeza das regiões de armazenamento do binário. No segundo sub-estado, Recepção, a transferência e armazenamento do binário deve ser feita.

No sub-estado Validação, o *bootloader* deve sempre validar a assinatura digital e, quando for o caso, a versão do candidato. Caso a assinatura digital não seja válida ou a versão seja inferior ao *firmware* instalado, nos cenários nos quais isso é previsto, o processo de atualização deve ser interrompido e o *bootloader* deve voltar para o sub-estado Preparação. Uma vez que as verificações sejam bem-sucedidas, o *bootloader* deve ir para o estado Instalação, no qual o candidato é movido para a partição de execução ou o *bootloader* realiza o chaveamento entre as partições, nos cenários nos quais duas partições foram definidas. Nos dispositivos que possuam apenas uma partição, essa transferência não é necessária. Caso o resumo criptográfico do *firmware* seja utilizado para a verificação de integridade e o mesmo não tenha sido transmitido ao dispositivo, é neste momento que o *bootloader* deve gerar e armazenar essa informação. É no estado Instalação que o indicador de atualização é desabilitado pelo *bootloader*.

Por fim, antes de passar o controle do dispositivo para o *firmware*, o *bootloader* deve passar pelo estado Verificação novamente e pelo estado Confirmação. O estado Verificação foi discutido anteriormente, enquanto que no estado Confirmação é analisada a integridade do *firmware* instalado, validando o resumo criptográfico ou a assinatura digital. Se a análise retornar sucesso, o *bootloader* deve encerrar sua execução e passar o controle do dispositivo ao *firmware*. Caso seja identificado que o *firmware* não é íntegro, o indicador de atualização é ativado novamente e o *bootloader* deve retornar ao estado inicial, para que uma nova atualização seja realizada, evitando que o dispositivo fique inutilizado.

Para apresentar esse processo de forma mais dinâmica, um fluxograma do comportamento do *bootloader* é apresentado na Figura 9. É demonstrado todo fluxo de atualização e verificação da integridade do *firmware* instalado, bem como o tratamento das possíveis exceções para manter o dispositivo sempre em um estado conhecido.

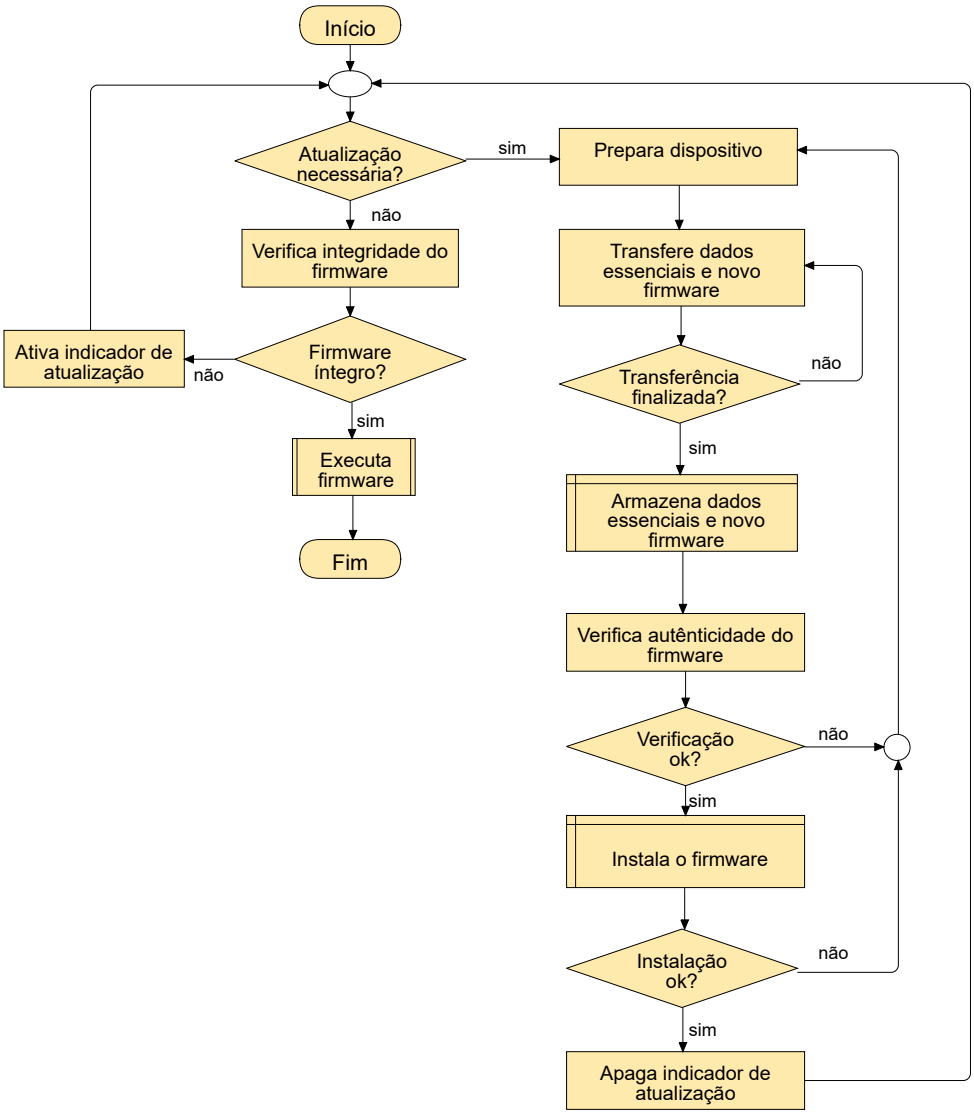
Além da modelagem da máquina de estados finita e do fluxograma, na Figura 10 propõe-se um diagrama de classes UML (OMG, 2017) para a implementação do *bootloader*. Essas classes possuem métodos abstratos e têm o objetivo de simplificar a implementação do *bootloader* em diferentes plataformas, de forma que seja necessário apenas realizar a especialização dos métodos que interagem com o *hardware* de cada microcontrolador. Cada classe abstrata já prevê os métodos necessários para que o *bootloader* seja capaz de realizar o processo de atualização proposto. As classes podem ser encontradas no Apêndice A e são disponibilizadas no *GitHub*[1].

A classe de comunicação é responsável por estabelecer uma troca de dados entre o dispositivo alvo e o transmissor do binário, de modo que o dispositivo seja capaz de receber o mesmo. Como discutido anteriormente, o processo de atualização proposto é independente do tipo de tecnologia de comunicação. Assim, nesta classe, são previstos apenas métodos para o envio e recebimento de dados.

A classe de criptografia é responsável por realizar as operações criptográficas das quais esta solução de atualização de *firmware* depende. Nesta classe, portanto, é necessário implementar três métodos, sendo eles: geração e verificação de resumos criptográficos e validação de assinatura digital. O método de validação de assinatura digital é utilizado no processo de atualização para validar a autenticidade do

[1] https://github.com/paulosell/secure-firmware-update/tree/master/classes-abstratas

Figura 9 – Fluxograma do modelo de *bootloader*.



Fonte: Própria.

binário e também pode ser utilizada a cada inicialização do dispositivo, a fim de verificar a integridade do *firmware* instalado, enquanto que os métodos de geração e verificação de resumos criptográficos são executados apenas para a verificação da integridade do *firmware*, nos casos em que essa abordagem é utilizada.

A classe de gerência de memória tem como principal função gerenciar o acesso e manipular a memória de programa do dispositivo. As principais operações desta classe são a escrita e leitura da memória de programa. Desta forma, nesta classe definem-se métodos para cada uma dessas operações. É de responsabilidade do *bootloader* conhecer de antemão os endereços de memória que ele tem de escrever, ler ou apagar, de acordo com seu estado e o momento do processo de atualização ou inicialização que ele se encontra, portanto, é nessa classe que as partições do dispositivo são definidas. Em alguns microcontroladores, antes de fazer a escrita em algum endereço de memória de programa, é necessário liberar o acesso à mesma. Em razão disso, além dos métodos de escrita e leitura, é previsto nesta classe métodos para bloquear e desbloquear o acesso à memória de programa do dispositivo.

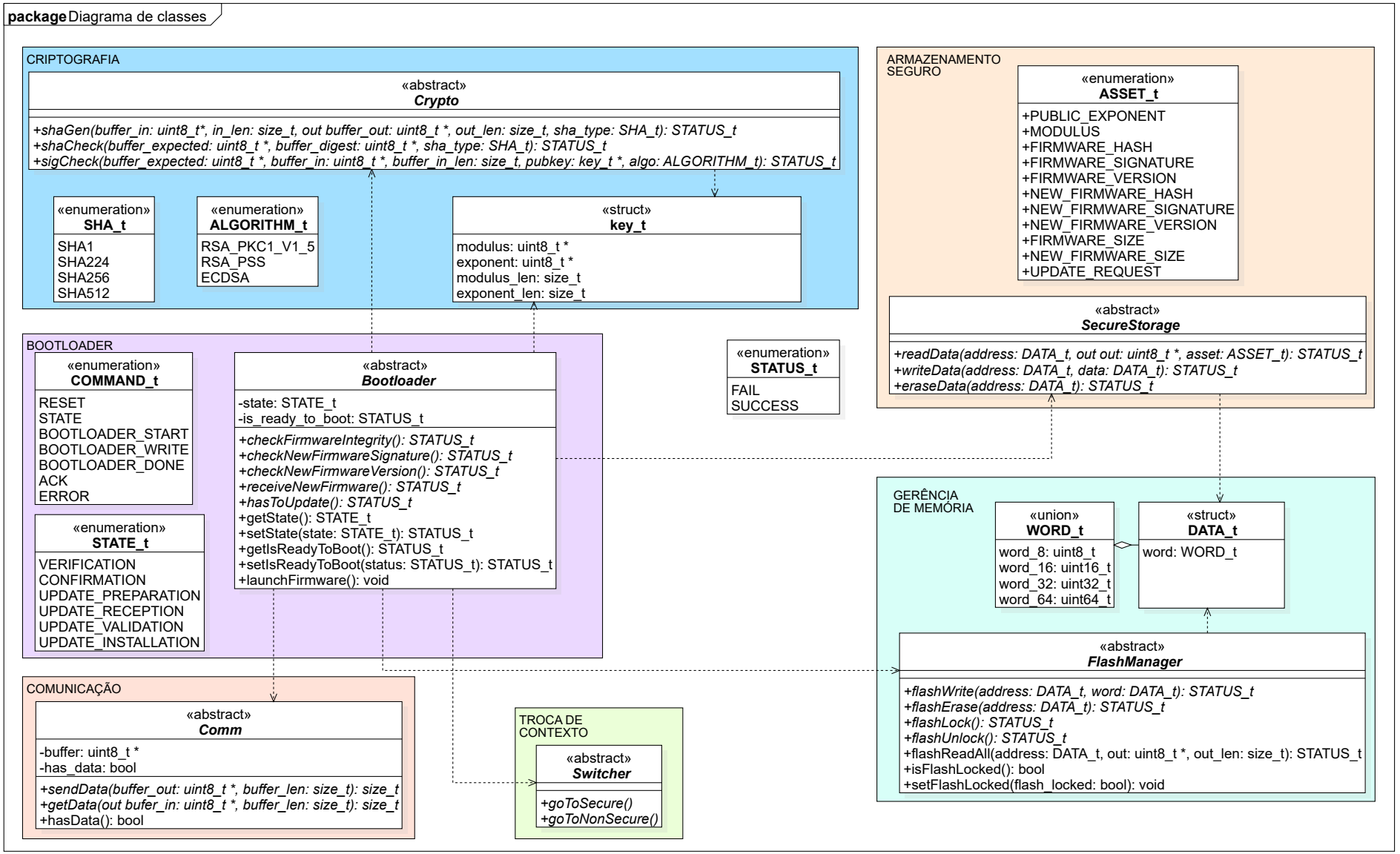
Como as informações essenciais ao processo de atualização devem ser armazenadas em um ambiente de execução seguro, é necessário uma classe que realize a interface entre o *bootloader* e o local

onde essas informações estão armazenadas. O ambiente seguro deve permitir a escrita e leitura das informações essenciais. O ambiente seguro no qual essas informações são armazenadas não é fixado pela proposta, podendo ser um dos ambientes seguros de execução baseados em *hardware* citados na seção 2.3.

Além de fazer parte do *bootloader*, a classe de troca de contexto pode também ser especializada pelo *firmware* do dispositivo. Sua função é fazer a troca de contexto entre o *bootloader* e o *firmware*, por exemplo, o momento em que o *bootloader* é encerrado para que o *firmware* entre em execução. Desta forma, propõe-se nesta classe apenas dois métodos: um para a troca de contexto entre *bootloader* e *firmware* e outro para gerar o aviso ao *bootloader*, a partir do *firmware*, de que um processo de atualização deve ser iniciado.

A classe do *bootloader* depende de todas as classes já citadas. Ela é responsável por implementar a atualização e inicialização do dispositivo, ou seja, é esta classe que implementa os processos apresentados na Figura 9. São propostos métodos para verificação da integridade do *firmware*, verificação da assinatura digital, recepção e armazenamento do binário, assim como finalização do procedimento de instalação, além de um método para verificação da versão do candidato, para os casos em que essa informação é relevante.

Figura 10 – Diagrama de classes UML do *bootloader* proposto.



Fonte: Própria.

4 IMPLEMENTAÇÃO DA SOLUÇÃO DE ATUALIZAÇÃO DE *FIRMWARE*

A fim de mostrar o funcionamento da solução de atualização de *firmware* e o *bootloader* propostos, as classes abstratas foram especializadas no microcontrolador STM32L562QEI6Q, da fabricante *STMicroelectronics*. Foi utilizado o *kit* de desenvolvimento *STM32L562-DK Discovery*, apresentado na Figura 11. A utilização deste *kit* se dá em função da integração do mesmo com os *software* de desenvolvimento que a *STMicroelectronics* disponibiliza, como a *Integrated Development Environment* (IDE) *STM32CubeIDE*, que permite realizar a depuração do código sendo executado no microcontrolador, bem como o *STM32CubeProgrammer*, utilizado para fazer escritas na memória *flash* do microcontrolador e do *STM32CubeMX*, utilizado para fazer a inicialização do projeto (STMICROELECTRONICS, 2021a). O *kit*, apresentado na Figura 11, também possui *LEDs* e botões, interfaces que auxiliam o desenvolvimento junto ao microcontrolador. O projeto com a implementação da solução pode ser encontrado no *GitHub*[1].

Dentre as características do microcontrolador, estão seu núcleo baseado no *Cortex ARM M-33*, que possui o *TrustZone*, a implementação de um *Trusted Execution Environment* (TEE) da empresa *ARM*, e seu armazenamento de 512 *kBytes*, que pode ser divido em dois bancos de 256 *kBytes*. Além disso, este microcontrolador possui aceleradores criptográficos baseados em *hardware*, que auxiliam no processo de verificação de assinaturas e geração de resumos criptográficos.

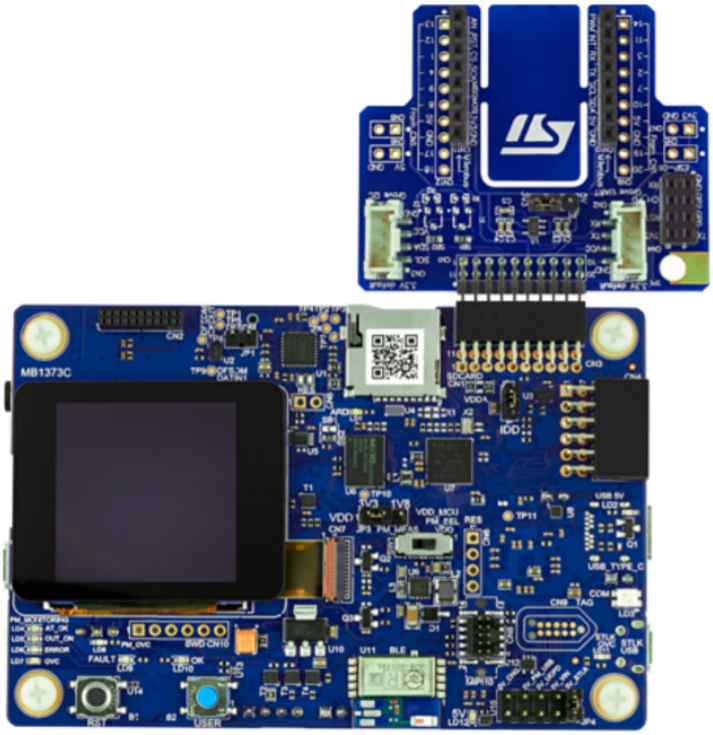
Quando o *TrustZone* está habilitado, é possível separar a memória de programa do microcontrolador em uma área segura e uma área não segura. Um código sendo executado na área não segura é totalmente isolado e não possui acesso à área segura, a não ser pela possibilidade de chamadas feitas para funções armazenadas em uma região chamada *Non-Secure Callable*. Essa região serve para armazenar código com funções de interface, as quais permitem a transição controlada de um ambiente não seguro para um ambiente seguro. Entretanto, essas chamadas precisam ser pré-definidas e acordadas em tempo de compilação. Por outro lado, um código executado na área segura tem acesso a toda memória de programa do microcontrolador (STMICROELECTRONICS, 2020d).

O uso do *TrustZone* oferece algumas funcionalidades de segurança que foram utilizadas para garantir a imutabilidade do *bootloader*, bem como garantir que o mesmo seja sempre executado a cada inicialização do dispositivo. Para tanto, ativou-se no microcontrolador o mecanismo de segurança *Write Protection* (WRP), o qual faz com que uma área específica da memória de programa do microcontrolador seja imutável. As regiões que tiveram essa funcionalidade ativada foram as que armazenavam o *bootloader* e a chave pública do emissor do *firmware*. Além disso, um ponto único de início de execução foi definido, sendo este ponto o endereço de memória onde o *bootloader* é iniciado.

Para a comunicação entre o microcontrolador e o responsável por transferir o binário, optou-se por utilizar a classe *Communications Device Class* (CDC) do protocolo *Universal Serial Bus* (USB) (USB-IF, 2010). Não é escopo deste trabalho apresentar com detalhes a especificação desta classe USB, mas basicamente, a classe CDC permite estabelecer uma comunicação serial entre o microcontrolador e o transmissor do binário, fazendo com que o estabelecimento da comunicação entre o par comunicante seja realizado de forma simples. A fabricante do microcontrolador disponibiliza uma biblioteca de fácil compreensão e utilização, facilitando a especialização da classe abstrata de comunicação.

[1] https://github.com/paulosell/secure-firmware-update/tree/master/implementacao

Figura 11 – *Kit* de desenvolvimento *STM32L562-DK Discovery*.



Fonte: (STMICROELECTRONICS, 2021b).

A classe de criptografia foi especializada a partir de uma biblioteca criptográfica que a fabricante do microcontrolador disponibiliza. Como algoritmo de chave pública, optou-se por utilizar o algoritmo *Rivest-Shamir-Adleman* (RSA) (JONSSON; KALISKI, 2003) com chaves de 2048 *bits*. Nesta implementação, a assinatura digital do *firmware* foi utilizada apenas para verificar a autenticidade do mesmo. Para a verificação da integridade do *firmware*, foi utilizado o algoritmo de resumo criptográfico SHA-256 (NIST, 2015).

A classe de gerência da memória também foi especializada a partir de uma biblioteca própria da fabricante do microcontrolador. Nesta implementação, foram definidas duas partições para o armazenamento do *firmware*, bem como as definições de endereços que indicam o início e fim de cada partição. Além dos métodos obrigatórios definidos na classe abstrata correspondente, implementou-se também alguns métodos para facilitar o acesso ao armazenamento em função do microcontrolador escolhido. Para tal, métodos para obter a página e o banco no qual um endereço da memória se encontra, foram implementados.

Em virtude da utilização de um TEE como *hardware* seguro, foi possível armazenar as informações essenciais ao processo de atualização de *firmware* e inicialização do microcontrolador na área da memória de programa segura. Desta forma, um *firmware* sendo executado na área não segura não tem acesso à essas informações. A especialização da classe de armazenamento seguro utiliza como base a especialização da classe de gerência da memória. A manipulação destes dados não passa de um acesso à memória segura do microcontrolador, portanto, é possível utilizar os métodos da classe de gerência da memória para as operações necessárias. É na classe de armazenamento seguro que as definições de endereços de memória, nos quais cada informação sensível ao processo de atualização de *firmware* ou inicialização do dispositivo, foram realizadas. Todos esses endereços são da área segura da memória de programa. A Figura 12 apresenta a segmentação da memória de programa do microcontrolador, indicando os ambientes de execução seguro e não seguro, bem como indicando as áreas nas quais o *bootloader*, as informações essenciais, as chamadas *Non-Secure Callable* (NSC) e os *firmware* são armazenados.

Dos dois métodos propostos na classe abstrata de troca de contexto, o *bootloader* implementa apenas o método que inicia a execução do *firmware*. A especialização deste método foi feita a partir de um ponteiro para uma função que faz o salto da memória de programa segura (*bootloader*) para a memória de programa não segura, colocando o *firmware* em execução. O uso de um ponteiro para função como método para sair do ambiente seguro é uma recomendação da própria fabricante do dispositivo (STMICROELECTRONICS, 2020a).

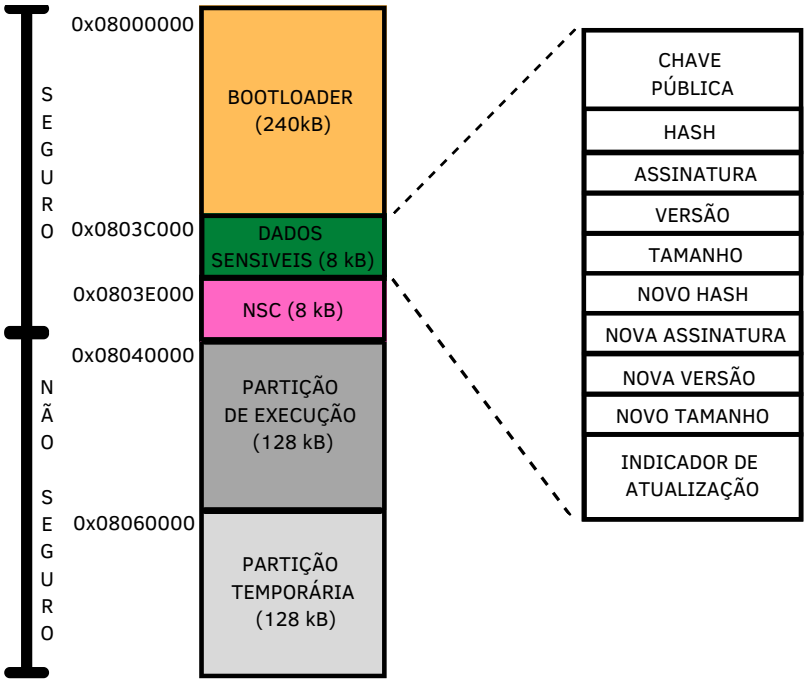
Por fim, a especialização da classe do *bootloader* foi feita a partir das demais especializações das classes que o *bootloader* é dependente. Para a recepção do novo *firmware*, estruturou-se um pequeno protocolo de comunicação.

A partir do momento em que o *bootloader* está pronto para a recepção, o responsável pela transferência deve enviar ao dispositivo um quadro contendo o resumo criptográfico, a assinatura digital e a versão do *firmware* candidato. Nesta implementação o transmissor também envia ao dispositivo o tamanho do *firmware* candidato, a fim de facilitar a leitura do mesmo quando esse processo for necessário.

Após o recebimento dessas informações, o *bootloader* espera a transmissão do candidato. O transmissor deve enviar ao dispositivo um quadro contendo uma parte do *firmware* candidato, bem como o tamanho desta parte. A transmissão do candidato ocorre em quadros de até 480 *bytes*, até que o mesmo seja transmitido por completo. Uma vez que a transmissão tenha sido finalizada, o transmissor avisa ao *bootloader* o fim da transmissão.

A partir deste ponto, o *bootloader* inicia as verificações para identificar a validade do candidato. Por ser menos custosa, primeiramente é realizada a verificação da versão. Uma vez que essa versão seja maior que a versão do *firmware* instalado, o *bootloader* verifica a autenticidade do candidato, por meio da assinatura digital recebida. Após o sucesso nas duas verificações, o *bootloader* pode instalar o novo *firmware* candidato, transferindo-o da partição temporária para a partição de execução, além de desativar o indicador de atualização.

Figura 12 – Segmentação da memória de programa do microcontrolador.



Fonte: Própria.

4.1 Experimentos e resultados

A fim de verificar o funcionamento da implementação da solução proposta no Capítulo 3, cinco experimentos foram realizados. Um resumo das características de cada experimento e seus resultados é apresentado na Tabela 5. No cenário de testes, um computador realizou o papel de transmissor do binário, a partir de programas desenvolvidos com a linguagem de programação *Python3*. Os códigos de cada programa podem ser encontrados no *GitHub*[2]. Para facilitar a identificação da mudança dos *firmware* após o processo de atualização, foram desenvolvidos dois *firmware* diferentes, identificados a partir de agora como Tipo 1 e Tipo 2. O *firmware* Tipo 1 tem como característica um *LED* que pisca com frequência de 100 milissegundos, enquanto o *firmware Tipo 2* pisca seu *LED* a cada 1000 milissegundos.

Cada *firmware* utilizado nesta implementação especializa a classe abstrata de troca de contexto, a fim de que o mesmo possa avisar o *bootloader* quando uma atualização foi solicitada. Para esse fim, é utilizado uma chamada NSC a partir de uma API compartilhada entre as áreas segura e não segura. O *firmware*, sendo executado na área não segura, não tem conhecimento em relação à implementação da chamada NSC, apenas tem conhecimento da API compartilhada entre os dois ambientes. A implementação da chamada NSC é realizada na área segura. Nesta implementação, o *firmware*, a partir do acionamento de um botão do *kit* de desenvolvimento, faz a chamada NSC ao *bootloader*, informando que um processo de atualização foi solicitado. O *bootloader*, por sua vez, escreve em um indicador de atualização armazenado na região de memória segura, e reinicia o microcontrolador. Esse indicador só é apagado pelo *bootloader* uma vez que uma atualização tenha sido bem-sucedida. Também é possível avisar o *bootloader* de que uma atualização foi solicitada durante sua execução, ao reiniciar o dispositivo e manter o botão do *kit* de desenvolvimento pressionado.

O primeiro experimento realizado tinha como objetivo verificar se uma atualização seria bem-sucedida caso fosse enviado ao dispositivo as informações corretas. Os passos realizados neste experimento são apresentados na Figura 13. Como estado inicial, o dispositivo possuía o *firmware* Tipo 1 sendo executado. Após a solicitação de atualização, foi enviado para o dispositivo o binário com as informações essenciais corretas e o *firmware* Tipo 2. Uma vez que a transferência foi finalizada, o *bootloader* realizou as verificações de versão e assinatura, e após a validação das verificações, instalou o candidato com sucesso. Por fim, o *bootloader* encerrou sua execução, fazendo a verificação da integridade do *firmware* antes de passar o controle do dispositivo para o mesmo, finalizando com sucesso o experimento.

O segundo experimento foi a tentativa de atualização de *firmware* enviando ao dispositivo um binário com versão inferior ao *firmware* instalado. Um fluxograma do experimento é apresentado na Figura 14. O dispositivo tinha como estado inicial o *firmware* Tipo 1 sendo executado, e como versão do *firmware* o valor 2. Na primeira tentativa de atualização, o binário completo com o candidato Tipo 2 foi enviado, mas com o valor 1 na informação da versão, inferior à instalada. Após o envio do binário, o

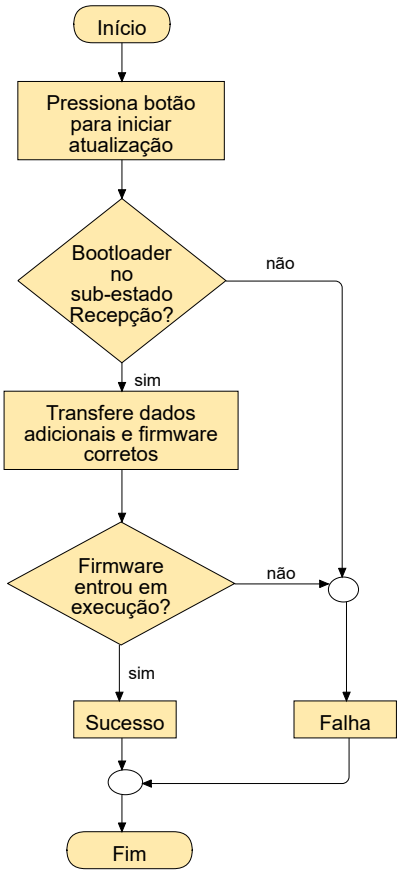
[2] https://github.com/paulosell/secure-firmware-update/tree/master/codigos-experimentos

Tabela 5 – Resumo dos experimentos realizados

	Resumo criptográfico	Assinatura	Versão	Tipo de transmissão	Resultado
Experimento 1	correto	válida	superior	completa	sucesso
Experimento 2	correto	válida	inferior	completa	sucesso
Experimento 3	correto	inválida	superior	completo	sucesso
Experimento 4	incorreto	válida	superior	completa	sucesso
Experimento 5	correto	válida	superior	incompleta	sucesso

Fonte: Própria

Figura 13 – Fluxograma do processo executado para o experimento 1.



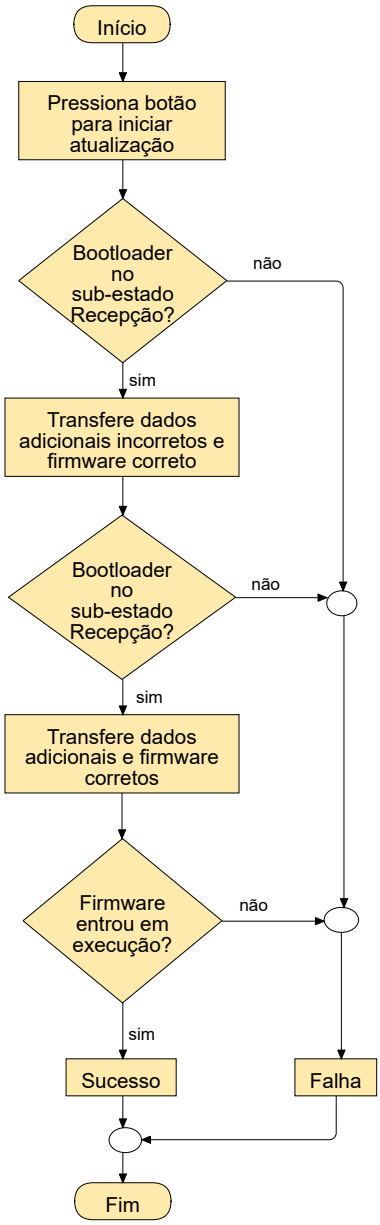
Fonte: Própria.

bootloader iniciou as verificações necessárias, falhando na validação da versão. O *bootloader* não finalizou a instalação e aguardou uma nova tentativa, conforme esperado. Em seguida, uma nova tentativa de atualização foi iniciada, dessa vez enviando um valor de versão superior à versão do *firmware* instalado. Depois de todos os processos já descritos o *firmware* TIPO 2 foi colocado em execução. Essa atualização bem-sucedida demonstrou a recuperação do *bootloader* após a falha na primeira atualização.

Também foi realizado o experimento no qual o transmissor envia para o *bootloader* uma assinatura inválida durante o processo de atualização. Este experimento também espera uma falha na primeira tentativa de atualização, uma vez que o *bootloader* não deve instalar um *firmware* no dispositivo sem fazer a validação da assinatura digital. O passo-a-passo do experimento também é apresentado na Figura 14. O dispositivo tinha o *firmware* TIPO 1 sendo executado. Uma vez que o *bootloader* estava pronto, o dispositivo recebeu do transmissor o binário com o candidato TIPO 2, mas com a assinatura inválida. Após a transferência, o *bootloader* iniciou as verificações, passando com sucesso na verificação da versão e falhando na verificação da assinatura digital. Sendo assim, o *bootloader* não finalizou a instalação e aguardou para que uma nova transmissão fosse iniciada. Na nova tentativa de atualização, o transmissor enviou o binário correto, o que resultou no *firmware* TIPO 2 sendo colocado em execução, novamente demonstrando a recuperação após uma tentativa de atualização com falha.

Além de testar as verificações durante o processo de atualização, também realizou-se um experimento para verificar o comportamento do *bootloader* ao identificar que o *firmware* instalado não está íntegro. Os passos realizados no experimento são apresentados na Figura 14. Inicialmente, é executado no dispositivo o *firmware* TIPO 1. O binário com o *firmware* candidato TIPO 2 foi enviado ao dispositivo. Entretanto, o resumo criptográfico foi substituído por 32 *bytes* aleatórios. Com o término da transferência,

Figura 14 – Fluxograma do processo executado para os experimentos 2, 3 e 4.

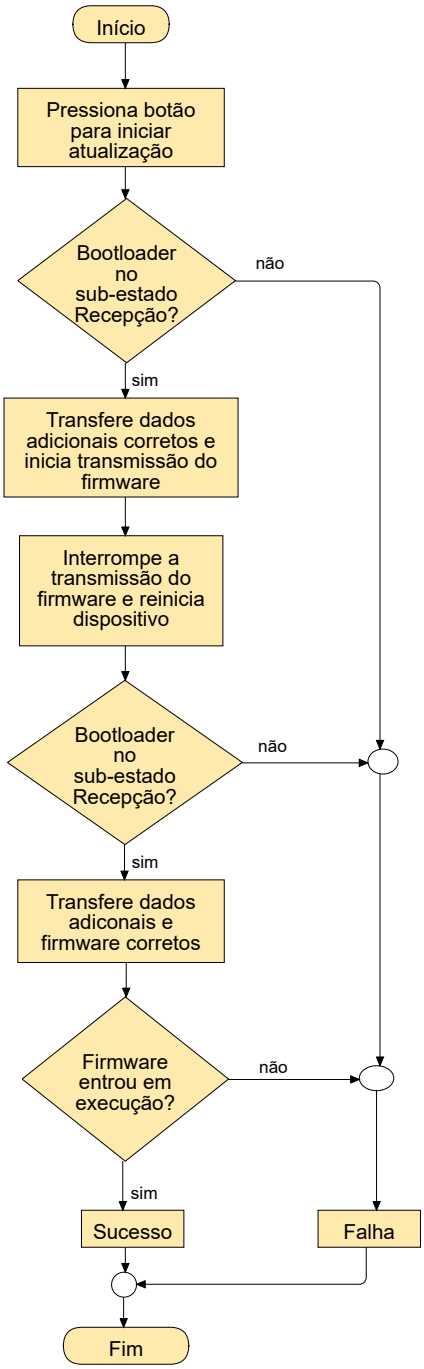


Fonte: Própria.

o *bootloader* iniciou as verificações, validando a instalação. Finalmente, o *bootloader* verificou a integridade do *firmware* instalado, comparando o resumo criptográfico recém gerado com o valor armazenado durante o processo de atualização. Como o resumo criptográfico armazenado é diferente do resumo gerado, o *bootloader* não permitiu que o *firmware* entrasse em execução e aguardou uma nova tentativa de atualização. Após a segunda tentativa de atualização, dessa vez com o resumo criptográfico correto, o processo de atualização foi executado com sucesso e o *firmware* Tipo 2 entrou em execução.

O último experimento verifica o comportamento do *bootloader* a partir de uma tentativa de atualização que não é finalizada. A Figura 15 apresenta o passo-a-passo desse experimento. Com o *firmware* Tipo 1 sendo executado, uma tentativa de atualização foi iniciada. O transmissor enviou o binário com o *firmware* candidato Tipo 2, porém, após iniciar a transferência do binário, o transmissor interrompeu esse processo. Em seguida, o microcontrolador foi reiniciado e após o *bootloader* entrar em execução, o mesmo identificou que uma tentativa de atualização fora solicitada anteriormente. Desta forma, o *bootloader*

Figura 15 – Fluxograma do processo executado para o experimento 5.



Fonte: Própria.

aguardou para que uma nova tentativa de atualização tivesse início, a qual foi realizada com sucesso.

Todos os experimentos realizados tinham como objetivo validar a implementação realizada e, consequentemente, a solução de atualização apresentada no Capítulo 3. Todos os experimentos foram bem-sucedidos, demonstrando que a proposta de atualização é robusta e atende os requisitos de segurança e confiabilidade de atualização.

5 CONCLUSÕES

Este trabalho teve como objetivo propor uma solução de atualização de *firmware* em dispositivos com sistemas embarcados microcontrolados, de forma que a atualização destes dispositivos ocorra de forma segura e confiável. Uma atualização segura é aquela que permite que apenas *firmware* emitidos por fontes confiáveis sejam instalados no dispositivo, enquanto que uma atualização confiável é aquela que garante que o dispositivo não ficará inutilizável após o processo de atualização.

Um modelo de *bootloader* foi proposto, de modo que o mesmo possa ser utilizado em diferentes soluções e arquiteturas de sistemas embarcados. Uma organização de classes foi projetada, que implementam o comportamento do *bootloader*, sendo necessário apenas especializar os métodos abstratos de cada classe que tenham interação com o *hardware* de um microcontrolador.

Uma implementação da solução foi realizada, a partir do modelo de *bootloader* proposto. A implementação foi realizada em um microcontrolador que possui um TEE, sendo este o responsável pelo armazenamento seguro das informações que são essenciais para a atualização do *firmware*. Algumas classes do *bootloader* foram implementadas a partir de biblioteca que a fabricante do microcontrolador disponibiliza.

Experimentos foram conduzidos para validar a solução de atualização de *firmware* e o modelo de *bootloader*, bem como validar a implementação realizada. Nos cenários nos quais uma tentativa de atualização foi realizada enviando ao dispositivo alguma informação incorreta, como uma assinatura inválida, o dispositivo não permitiu que a atualização fosse completada ou colocou o *firmware* em execução, demandando uma nova tentativa de atualização. Ao interromper uma tentativa de atualização no meio do processo de transferência do novo *firmware*, o dispositivo foi capaz de identificar que uma tentativa de atualização não fora completada e não colocou o *firmware* em execução até que uma atualização fosse bem-sucedida. Em nenhum dos experimentos o dispositivo foi para um estado de execução o qual não era previsto e sempre que uma atualização foi realizada enviando ao dispositivo todas as informações de forma correta, o procedimento foi concluído com êxito.

Os testes realizados demonstraram que a proposta apresentada neste trabalho permite que uma atualização de *firmware* seja realizada de forma segura e confiável, bem como permitiram validar a especialização das classes do *bootloader* para o cenário no qual um microcontrolador com um TEE é utilizado.

5.1 Trabalhos futuros

Este trabalho cita diferentes tipos de *hardware* seguro que podem ser utilizados para o armazenamento do *bootloader* e das informações essenciais ao processo de atualização. Desta forma, sugere-se utilizar um *hardware* seguro diferente do utilizado na implementação apresentada, como um *Secure Element* (SE), em conjunto com um microcontrolador não seguro, a fim de fazer a validação da solução de atualização com uma organização diferente. O elemento seguro seria responsável por armazenar as informações essenciais e deve ser consultado pelo microcontrolador seguro a cada processo de atualização ou inicialização. O microcontrolador não seguro deveria também prover algum mecanismo que garanta que o *bootloader* seja imutável e sempre executado.

A implementação da solução de atualização não utilizou certificados digitais para obter a chave pública do emissor do *firmware*. Sugere-se que a mesma seja estendida, a fim de obter a chave pública

do emissor do *firmware* a partir de um certificado digital emitido por uma autoridade certificadora de confiança, permitindo que a solução de atualização seja utilizada em cenários no qual uma terceira entidade precise certificar um *firmware* que será instalado em um dispositivo.

A implementação do *bootloader* ocupou cerca de 85% dos 240kB disponíveis para o mesmo. Como o microcontrolador utilizado possui um grande espaço de memória *flash*, o tamanho do *bootloader* não teve efeitos negativos. Entretanto, em microcontroladores nos quais as memórias *flash* sejam menores, o tamanho do *bootloader* poderia ser um empecilho, impedindo seu armazenamento. Sugere-se então que a implementação deste *bootloader* seja modificada, identificando possíveis pontos nos quais a mesma possa ser otimizada, permitindo que ela possa ser portada para outros dispositivos com maior facilidade.

Uma comparação entre a solução de atualização de *firmware* proposta e diferentes soluções de atualização de *firmware* disponibilizadas por fabricantes de microcontroladores pode ser realizada, de forma que essa comparação identifique as vantagens e desvantagens de cada solução.

REFERÊNCIAS

ARFAOUI, G.; GHAROUT, S.; TRAORé, J. Trusted execution environments: A look under the hood. In: *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. [S.l.: s.n.], 2014. p. 259–266. Citado na página 31.

BENINGO, J. Bootloader design for microcontrollers in embedded systems. In: . [s.n.], 2015. Disponível em: <https://www.beningo.com/wp-content/uploads/images/Papers/bootloader_design_for_microcontrollers_in_embedded_systems%20.pdf>. Citado 4 vezes nas páginas 23, 32, 33 e 34.

BISHOP, M.; BAILEY, D. *A Critical Analysis of Vulnerability Taxonomies*. [S.l.], 1996. Citado na página 25.

BOUAZZOUNI, M. A.; CONCHON, E.; PEYRARD, F. Trusted mobile computing: An overview of existing solutions. *Future Generation Computer Systems*, v. 80, p. 596 – 612, 2018. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167739X16301510>. Citado 4 vezes nas páginas 29, 30, 31 e 32.

GLOBALPLATFORM. Trusted execution environments (tee): An introduction to tee functionality and how globalplatform supports it. In: _____. *Introduction to trusted execution environments*. 2018. Disponível em: <https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf>. Acesso em: 31 ago. 2019. Citado na página 31.

JAIN, N.; MALI, S. G.; KULKARNI, S. Infield firmware update: Challenges and solutions. In: *2016 International Conference on Communication and Signal Processing (ICCSP)*. [S.l.: s.n.], 2016. p. 1232–1236. Citado 2 vezes nas páginas 23 e 32.

JOHNER, H. et al. *Deploying a Public Key Infrastructure*. [S.l.]: IBM, 2000. ISBN 9780738415734. Citado 2 vezes nas páginas 27 e 28.

JONSSON, J.; KALISKI, B. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. [S.l.], 2003. Disponível em: <https://tools.ietf.org/html/rfc3447>. Citado na página 44.

LANDWEHR, C. Computer security. *International Journal of Information Security*, v. 1, p. 3–13, 2001. Citado na página 25.

LEE, R. B. *Security Basics for Computer Architects*. [S.l.]: Morgan & Claypool Publishers, 2013. ISBN 9781627051569. Citado 3 vezes nas páginas 25, 26 e 27.

MELLO, E. R. de et al. Segurança em serviços web. In: *Minicursos do Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2006*. [s.n.], 2006. Disponível em: <http://docente.ifsc.edu.br/mello/artigos/mellomcsbseg06.pdf>. Citado na página 25.

NIKOLOV, N. Research firmware update over the air from the cloud. In: *2018 IEEE XXVII International Scientific Conference Electronics - ET*. [S.l.: s.n.], 2018. p. 1–4. ISSN null. Citado na página 23.

NIST. *Secure Hash Standards*. National Institute of Standards and Technology, 2015. Federal Information Processing Standards Publications (FIPS PUBS) 180-4. Disponível em: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>. Citado na página 44.

NYMAN, T.; EKBERG, J.-E.; ASOKAN, N. Citizen electronic identities using tpm 2.0. In: *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*. New York, NY, USA: ACM, 2014. (TrustED '14), p. 37–48. ISBN 978-1-4503-3149-4. Disponível em: <http://doi-acm-org.ez130.periodicos.capes.gov.br/10.1145/2666141.2666146>. Citado na página 30.

OMG. *OMG Unified Modeling Language (OMG UML)*. 2017. Disponível em: <https://www.omg.org/spec/UML/2.5.1/PDF>. Citado 2 vezes nas páginas 37 e 38.

REVEILHAC, M.; PASQUET, M. Promising secure element alternatives for nfc technology. In: *2009 First International Workshop on Near Field Communication*. [S.l.: s.n.], 2009. p. 75–80. ISSN null. Citado na página 31.

RUSSEL, D.; GANGEMI, G. T. *Computer Security Basics*. [S.l.]: O'Reilly & Associates, 1991. ISBN 0-937175-71-4. Citado na página 25.

SHEPHERD, C. et al. Secure and trusted execution: Past, present and future – a critical review in the context of the internet of things and cyber-physical systems. In: . [S.l.: s.n.], 2016. Citado na página 30.

SHIREY, R. W. *Internet Security Glossary, Version 2*. RFC Editor, 2007. RFC 4949. (Request for Comments, 4949). Disponível em: <https://rfc-editor.org/rfc/rfc4949.txt>. Citado na página 25.

STALLINGS, W. *Criptografia E Segurança De Redes - Princípios e Práticas*. PEARSON BRASIL, 2008. ISBN 9788543005898. Disponível em: <https://books.google.com.br/books?id=KO8gvgAACAAJ>. Citado 5 vezes nas páginas 23, 25, 26, 27 e 28.

STMICROELECTRONICS. *Getting started with projects base on the STM32L5 Series in STM32CubeIDE*. 2020. Disponível em: <https://www.st.com/resource/en/application_note/dm00652038-getting-started-with-projects-based-on-the-stm32l5-series-in-stm32cubeide-stmicroelectronics.pdf>. Citado na página 45.

STMICROELECTRONICS. *Getting started with STM32CubeL5 TFM application*. 2020. Disponível em: <https://www.st.com/resource/en/user_manual/dm00678763-getting-started-with-stm32cubel5-tfm-application-stmicroelectronics.pdf>. Citado na página 24.

STMICROELECTRONICS. *Getting started with the X-CUBE-SBSFU STM32Cube Expansion Package*. 2020. Disponível em: <https://www.st.com/resource/en/user_manual/dm00414687-getting-started-with-the-xcubesbsfu-stm32cube-expansion-package-stmicroelectronics.pdf>. Citado na página 24.

STMICROELECTRONICS. *Reference manual - STM32L552xx and STM32l562xx advanced Arm-based 32-bit MCUs*. 2020. Disponível em: <https://www.st.com/resource/en/reference_manual/dm00346336-stm32l552xx-and-stm32l562xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf>. Citado na página 43.

STMICROELECTRONICS. *STM32Cube ecosystem overview*. 2021. Disponível em: <https://www.st.com/content/ccc/resource/sales_and_marketing/presentation/product_presentation/group0/5a/0b/83/43/e3/f1/4f/e7/STM32Cube_Ecosystem_Overview/files/STM32Cube_Ecosystem_Overview.pdf/jcr:content/translations/en.STM32Cube_Ecosystem_Overview.pdf>. Citado na página 43.

STMICROELECTRONICS. *User Manual - Discovery kit with STM32L562QE MCU*. 2021. Disponível em: <https://www.st.com/resource/en/user_manual/dm00635554-discovery-kit-with-stm32l562qe-mcu-stmicroelectronics.pdf>. Citado na página 44.

USB-IF. *Universal Serial Bus Class Definitions for Communications Devices*. [S.l.]: USB-IF, 2010. Revision 1.2 (Errata 1.0). Citado na página 43.

Apêndices

APÊNDICE A – CLASSES ABSTRATAS

As listagens a seguir apresentam os *headers* das classes abstratas propostas neste trabalho, desenvolvidas com a linguagem de programação *C++*. O Código A.1 apresenta a classe abstrata de comunicação. Já O Código A.2 apresenta a classe abstrata de troca de contexto entre áreas segura e não segura. O Código A.3 apresenta a classe abstrata de criptografia. As classes abstratas de gerência de memória e de armazenamento seguro são apresentadas nos Código A.4 e Código A.5, respectivamente. Por fim, o Código A.6 apresenta a classe abstrata do *Bootloader*.

Código A.1 – *Header* da classe abstrata de comunicação

```
#ifndef COMM_H_
#define COMM_H_

#include <iostream>

class Comm {

public:
  Comm();
  virtual size_t sendData(uint8_t *buffer_out, size_t buffer_len) = 0;
  virtual size_t getData(uint8_t *buffer_in, size_t * buffer_len) = 0;
  virtual bool hasData(void);

private:
  uint8_t *buffer;
  bool has_data;
};

#endif
```

Código A.2 – *Header* da classe abstrata de troca de contexto

```
#ifndef SWITCHER_H_
#define SWITCHER_H_

#include <iostream>

class Switcher {
public:

  Switcher();
  virtual void goToSecure(void) = 0;
  virtual void goToNonSecure(void) = 0;
};

#endif
```

Código A.3 – *Header* da classe abstrata de criptografia

```
#ifndef CRYPTOGRAPHY_H_
#define CRYPTOGRAPHY_H_

#include <iostream>

class Cryptography {
public:

  enum SHA_t {
    SHA1 = 1, SHA224 = 2, SHA256 = 3, SHA512 = 4
  };

  enum ALGORITHM_t {
    RSA_PKCS1_V1_5 = 1, RSA_PSS = 2, ECDSA = 3
  };

  enum STATUS_t {
    FAIL = 0, SUCCESS = 1
  };

  typedef struct {
    uint8_t *x;
    uint8_t *y;
  } ecc_key_t;

  typedef struct {
    uint8_t *modulus;
    uint8_t *exponent;
    size_t modulus_len;
    size_t exponent_len;
  } rsa_key_t;

  typedef union {
    ecc_key_t ecc_key;
    rsa_key_t rsa_key;
  } key_t;

  Cryptography();
  virtual STATUS_t shaGen(uint8_t *buffer_in, size_t in_len,
      uint8_t *buffer_out, size_t *out_len, SHA_t sha_type) = 0;

  virtual STATUS_t sigCheck(uint8_t *buffer_expected, uint8_t *buffer_in,
      size_t buffer_in_len, key_t *pubkey, ALGORITHM_t algo) = 0;

  STATUS_t shaCheck(uint8_t *buffer_expected, uint8_t *buffer_digest,
      SHA_t sha_type);

};

#endif
```

Código A.4 – *Header* da classe abstrata de gerência de memória

```
#ifndef FLASHMANAGER_H_
#define FLASHMANAGER_H_

#include <iostream>

#define FW_START_PAGE           'XXX'
#define FW_START_ADDRESS        'XXX'
#define FW_END_PAGE             'XXX'
#define FW_END_ADDRESS          'XXX'

#define NEW_FW_START_PAGE       'XXX'
#define NEW_FW_START_ADDRESS    'XXX'
#define NEW_FW_END_PAGE         'XXX'
#define NEW_FW_END_ADDRESS      'XXX'

#define NUM_OF_PAGES            'XXX'
#define PAGE_SIZE               'XXX'

union WORD_t {
  uint8_t word_8;
  uint16_t word_16;
  uint32_t word_32;
  uint64_t word_64;

};

typedef struct {
  WORD_t word;

} DATA_t;

class FlashManager {

public:

  enum STATUS_t {
    FAIL = 0, SUCCESS = 1
  };

  FlashManager();
  virtual STATUS_t flashWrite(DATA_t address, DATA_t word) = 0;
  virtual STATUS_t flashErase(DATA_t address) = 0;
  virtual STATUS_t flashLock(void) = 0;
  virtual STATUS_t flashUnlock(void) = 0;

  STATUS_t flashReadAll(DATA_t address, uint8_t *out, size_t out_len);
  bool isFlashLocked();
  void setFlashLocked(bool flashLocked);

private:
  bool flash_locked;
};

#endif
```

Código A.5 – *Header* da classe abstrata de armazenamento seguro

```
#ifndef SECURESTORAGE_H_
#define SECURESTORAGE_H_

#include <iostream>
#include "flashman.h"

#define ATTESTATION_KEY_PAGE            'XXX'
#define MODULUS_ADDRESS                 'XXX'
#define PUBLIC_EXPONENT_ADDRESS         'XXX'

#define FIRMWARE_ASSETS_PAGE            'XXX'
#define FIRMWARE_HASH_ADDRESS           'XXX'
#define FIRMWARE_SIGNATURE_ADDRESS      'XXX'
#define FIRMWARE_VERSION_ADDRESS        'XXX'
#define FIRMWARE_SIZE_ADDRESS           'XXX'

#define NEW_FIRMWARE_ASSETS_PAGE        'XXX'
#define NEW_FIRMWARE_HASH_ADDRESS       'XXX'
#define NEW_FIRMWARE_SIGNATURE_ADDRESS  'XXX'
#define NEW_FIRMWARE_VERSION_ADDRESS    'XXX'
#define NEW_FIRMWARE_SIZE_ADDRESS       'XXX'

#define UPDATE_REQUEST_PAGE             'XXX'
#define UPDATE_REQUEST_ADDRESS          'XXX'

class SecureStorage {

public:

  enum ASSET_t {
    PUBLIC_EXPONENT         = 1,
    MODULUS                 = 2,
    FIRMWARE_HASH           = 3,
    FIRMWARE_SIGNATURE      = 4,
    FIRMWARE_VERSION        = 5,
    NEW_FIRMWARE_HASH       = 6,
    NEW_FIRMWARE_SIGNATURE  = 7,
    NEW_FIRMWARE_VERSION    = 8,
    FIRMWARE_SIZE           = 9,
    NEW_FIRMWARE_SIZE       = 10,
    UPDATE_REQUEST          = 11
  };

  enum STATUS_t {
    FAIL = 0, SUCCESS = 1
  };

  SecureStorage();
  virtual STATUS_t readData(DATA_t address, uint8_t *out, ASSET_t asset) = 0;
  virtual STATUS_t writeData(DATA_t address, DATA_t data) = 0;
  virtual STATUS_t eraseData(DATA_t address) = 0;

};

#endif
```

Código A.6 – *Header* da classe abstrata do *Bootloader*

```
#ifndef BOOTLOADER_H_
#define BOOTLOADER_H_

#include <iostream>

class Bootloader{
public:

  enum STATE_t{
    VERIFICATION        = 0x01,
    CONFIRMATION        = 0x02,
    UPDATE_PREPARATION  = 0x03,
    UPDATE_RECEPTION    = 0x04,
    UPDATE_VALIDATION   = 0x05,
    UPDATE_INSTALLATION = 0x06
  };

  enum COMMAND_t{
    ACK                 = 0x00,
    RESET               = 0x01,
    STATE               = 0x02,
    BOOTLOADER_START    = 0x04,
    BOOTLOADER_WRITE    = 0x05,
    BOOTLOADER_DONE     = 0x06,
    ERROR               = 0x08
  };


  enum STATUS_t{
    FAIL = 0,
    SUCCESS = 1
  };


  Bootloader();
  virtual STATUS_t checkFirmwareIntegrity(void) = 0;
  virtual STATUS_t checkNewFirmwareSignature(void) = 0;
  virtual STATUS_t checkNewFirmwareVersion(void) = 0;
  virtual STATUS_t receiveNewFirmware(void) = 0;
  virtual STATUS_t hasToUpdate(void) = 0;
  virtual void launchFirmware(void) = 0;

  STATE_t getState(void);
  STATUS_t setState(STATE_t state);
  STATUS_t getIsReadyToBoot(void);
  STATUS_t setIsReadyToBoot(STATUS_t status);

private:

  STATE_t state;
  STATUS_t is_ready_to_boot;
};

#endif
```