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

Coorientador: Roberto de Matos

São José - SC

abril/2021

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 e principalmente 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.

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. Desta forma, 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 recebidos, bem como haja uma grande probabilidade de que o dispositivo não ficará inutilizável após o processo de atualização. A criptografia de chave pública e algoritmos de resumos criptográficos podem ser usados para prover tais garantias. À vista do que foi posto, este trabalho propõe uma solução de atualização de *firmware* em sistemas embarcados microcontrolados, de forma que os requisitos de segurança e confiabilidade da atualização sejam garantidas. Classes abstratas foram projetadas para que a proposta de atualização de *firmware* seja portável entre diferentes plataformas. A partir das classes abstratas, a solução foi implementada utilizando o *kit* de desenvolvimento *STM32L562-DK Discovery*. A implementação permitiu que apenas *firmwares* assinados por uma fonte conhecida fossem instalados no microcontrolador, bem como impediu que o dispositivo ficasse inutilizado após a tentativa de atualização.

Palavras-chave: Atualização de *firmware*. Segurança. 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 there is a high probability that the device will not be unusable after the update process. Public key cryptography and hash functions can be used to provide such guarantees. This work proposes a solution to update firmwares in embedded microcontrolled systems, so that the safety and reliability properties are guaranteed. Abstract classes were designed so that the proposed firmware update solution is portable across different platforms. The solution was implemented using the *STM32L562-DK Discovery* kit. The implementation allowed only firmwares signed by a known source to be installed on the microcontroller, as well as preventing the device from being unusable after the update attempt.

Keywords: Firmware update. Safety. Reliability. Integrity.

LISTA DE ILUSTRAÇÕES

Figura 1 – Modelo de criptografia simétrica 21
Figura 2 – Modelo de criptografia assimétrica 21
Figura 3 – Processo de assinatura digital 22
Figura 4 – Arquitetura simplificada de um *Trusted Execution Environment* (TEE) 25
Figura 5 – Componentes de um sistema com *bootloader* 27
Figura 6 – Processo padrão de atualização de *firmware* 27
Figura 7 – Uma máquina de estados finitos UML para o *bootloader*. 31
Figura 8 – Diagrama de classes UML das classes abstratas propostas. 32
Figura 9 – Diagrama de classes UML das classes abstratas propostas. 34
Figura 10 – *Kit* de desenvolvimento *STM32L562-DK Discovery*. 36
Figura 11 – Segmentação da memória do microcontrolador. 37
Figura 12 – Diagrama de atividades do experimento 1. 39
Figura 13 – Diagrama de atividades dos experimentos 2 e 3. 40
Figura 14 – Diagrama de atividades do experimento 4. 41
Figura 15 – Diagrama de atividades do experimento 5. 42

LISTA DE TABELAS

Tabela 1 – Campos de um certificado digital - recomendação X.509 22
Tabela 2 – Comparação entre os tipos de criptografia e as propriedades básicas de segurança . . . 23
Tabela 3 – Comparação entre os tipos de criptografia e ataques 23
Tabela 4 – Comparação entre tecnologias de ambientes seguros de execução 26
Tabela 5 – Resumo dos experimentos realizados 41

LISTA DE ABREVIATURAS E SIGLAS

ITU-T *Telecommunication Standardization Sector* 22
TPM *Trusted Platform Module* 24
SE *Secure Element* 24
TEE *Trusted Execution Environment* 9
UICC *Universal Integrated Circuit Card* 24
REE *Rich Execution Environment* 25
TA *Trusted Application* 25
USB *Universal Serial Bus* 32
API *Application Programming Interface* 24
UART *Universal Asynchronous Receiver and Transmiter* 32
WRP *Write Protection* 35
IDE *Integrated Development Environment* 35
CDC *Communications Device Class* 35
RSA *Rivest-Shamir-Adleman* 36
NSC *Non-Secure Callable* 37

SUMÁRIO

1 INTRODUÇÃO 17
1.1 Objetivo Geral 18
1.2 Objetivos específicos 18
1.3 Organização do texto 18

2 FUNDAMENTAÇÃO TEÓRICA 19
2.1 Propriedades básicas de segurança 19
2.2 Criptografia 20
2.2.1 Criptografia simétrica 20
2.2.2 Criptografia assimétrica 20
2.2.3 Comparativo entre criptografia simétrica e assimétrica 23
2.3 Ambientes seguros de execução 23
2.3.1 Trusted Platform Module 24
2.3.2 Secure Element 24
2.3.3 Trusted Execution Environment 25
2.3.4 Comparação entre os ambientes seguros de execução 26
2.4 Atualização de *firmware* 26

3 UMA SOLUÇÃO DE ATUALIZAÇÃO DE *FIRMWARE* 29
3.1 Um modelo de *bootloader* 30

4 IMPLEMENTAÇÃO DA SOLUÇÃO DE ATUALIZAÇÃO DE *FIRMWARE* 35
4.1 Experimentos e resultados 38

5 CONCLUSÕES 43
5.1 Trabalhos futuros 44

REFERÊNCIAS 45

APÊNDICES 47

APÊNDICE A – CLASSES ABSTRATAS 49

1 INTRODUÇÃO

Dispositivos que possuem algum tipo de sistema embarcado 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, e uma alteração no seu *software* embarcado - também chamado de *firmware* - pode corrigir a falha. Desta forma, os aparelhos devem possuir algum mecanismo de atualização de *firmware*, de preferência de forma remota, que previna ou corrija o mal funcionamento.

A possibilidade de atualizar o *firmware* de um dispositivo de forma remota traz consigo alguns benefícios. Primeiramente, evita que o usuário tenha que retornar o equipamento ao seu fabricante, poupando tempo e dinheiro do consumidor. Além disso, atualizar o *firmware* remotamente provê uma maior agilidade na melhoria dos equipamentos, uma vez que os dispositivos podem ser atualizados assim que o fabricante dos equipamentos disponibilizar a nova versão do *firmware* (JAIN; MALI; KULKARNI, 2016).

A atualização remota de *firmware* 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.

Antes de efetivamente atualizar o *firmware* de um dispositivo, deve-se observar os requisitos de segurança e confiabilidade do novo *firmware* que será recebido. Uma atualização segura é aquela que garante que apenas *firmwares* que sejam emitidos por fontes seguras, isto é, fontes conhecidas, possam ser instalados. Já uma atualização confiável é aquela que garante que o dispositivo não ficará inutilizável após o procedimento de atualização (NIKOLOV, 2018).

Para atingir uma atualização segura, é possível utilizar criptografia de chave pública. Nela, duas chaves distintas são geradas e utilizadas para cifrar e decifrar uma informação (LEE, 2013). Uma dessas chaves deve ser mantida em segredo pelo gerador das chaves e a outra deve ser compartilhada com quem deseja trocar informação com o gerador da chave. A partir disso, o emissor da informação, neste caso, o fornecedor do *firmware*, pode assinar a mensagem com sua chave privada e enviar a mensagem e a assinatura para o dispositivo que será atualizado. Como em teoria apenas o desenvolvedor do *firmware* possui a chave que cifrou o código, o dispositivo saberá que o *firmware* vem de uma fonte segura.

O requisito de confiabilidade, por sua vez, pode ser atingido com a verificação de integridade do *firmware* recebido. Esta verificação pode acontecer a partir do uso de funções de *hash*, que são funções de embaralhamento que geram saídas únicas para cada entrada (STALLINGS, 2008). Deve-se gerar um valor de *hash* tendo como entrada da função o novo *firmware*, e enviar para o dispositivo essa informação. Em seguida, o próprio dispositivo deverá realizar o mesmo procedimento, desta vez com o *firmware* que o mesmo recebeu, e comparar o valor gerado com o valor recebido anteriormente. Caso estes valores sejam diferentes, houve alguma modificação do *firmware* original para o recebido e desta forma o dispositivo não deverá aceitá-lo.

Diante do que foi posto, neste trabalho propõe-se uma solução de atualização de *firmware* em dispositivos microcontrolados, de forma que esta solução possua garantias de confiabilidade e segurança da atualização.

1.1 Objetivo Geral

O objetivo deste trabalho é propor uma solução que permita a atualização de *firmware* de um sistema embarcado em um dispositivo microcontrolado, garantindo que a atualização seja realizada de forma segura e confiável.

1.2 Objetivos específicos

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

1. Definir requisitos mínimos de *hardware* e os passos que uma atualização deve seguir para os requisitos de segurança e confiabilidade da atualização sejam atingidos;
2. Disponibilizar um conjunto de classes abstratas que permitam que a solução de atualização seja portável entre diferentes plataformas;
3. Desenvolver uma prova de conceito da solução a partir das classes abstratas projetadas.

1.3 Organização do texto

Este trabalho está dividido da seguinte maneira: O Capítulo 2 aborda os tópicos julgados necessários para o entendimento e desenvolvimento deste trabalho. Já o Capítulo 3 apresenta a proposta de solução de atualização de *firmware*, bem como apresenta as classes abstratas projetas. 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, retomando brevemente os conceitos debatidos e sugerindo tópicos relacionados que podem ser trabalhados no futuro.

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 *firmwares* 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).

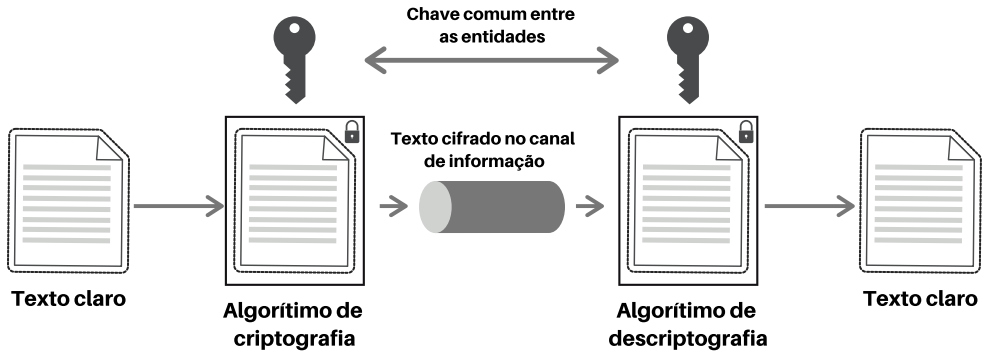
2.2.1 Criptografia simétrica

A criptografia simétrica é aquela onde apenas uma chave é utilizada para criptografar e descriptografar a informação. Desta maneira, a informação a ser criptografada (ou texto claro) é embaralhada a partir do algoritmo de criptografia e a 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

Figura 1 – Modelo de criptografia simétrica



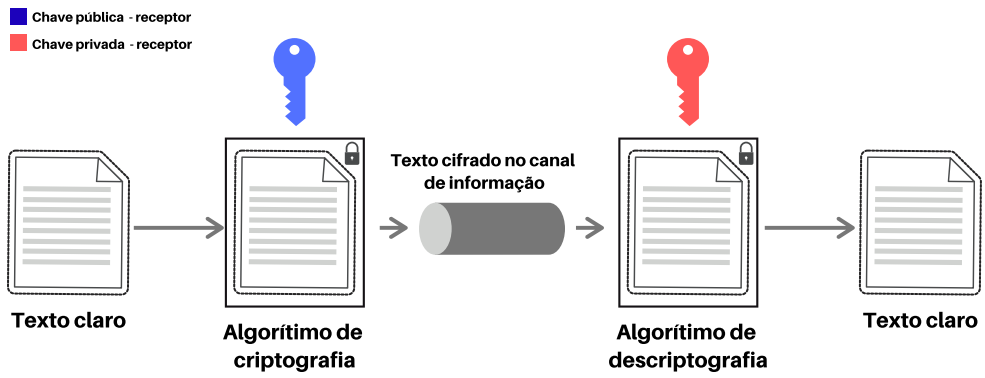
Fonte: Adaptada de (STALLINGS, 2008).

pública deverá ser distribuída entre os outros usuários. Com a criptografia de chave pública é possível 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 análoga 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).

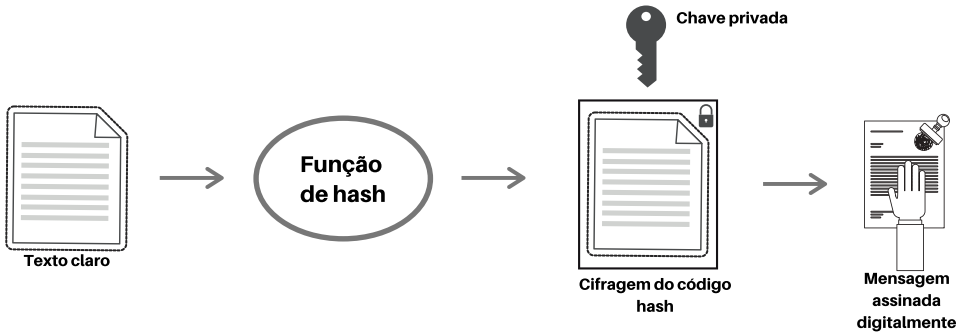
A assinatura digital é o processo de criptografar o código *hash* gerado de uma função *hash*, utilizando a chave privada do emissor. Esta assinatura é então anexada à mensagem a ser transmitida, ficando a cargo do receptor gerar um código *hash* da mensagem recebida, descriptografar a assinatura digital 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

Figura 2 – Modelo de criptografia assimétrica



Fonte: Própria.

Figura 3 – Processo de assinatura digital



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

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 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, através de 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

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

no seu modelo de certificado digital onze campos distintos, descritos na Tabela 1.

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 através de algoritmos e chaves criptográficas. Entretanto, apenas a criptografia assimétrica atende a integridade e o não-repúdio, através da assinatura digital, e a autenticidade, através dos 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 através do uso de assinatura digital, mecanismo que apenas a criptografia assimétrica possui. O disfarce pode ser evitado através 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 através das 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 *softwares* 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

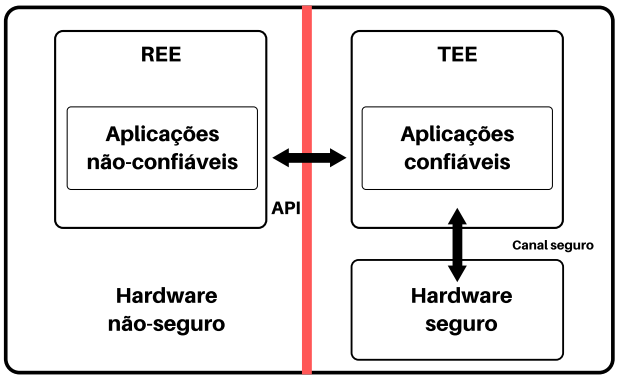
O TEE é um ambiente que possui seu próprio sistema operacional, chamado de sistema operacional seguro. A GlobalPlatform - associação entre empresas para padronização de tecnologias para computação segura (GLOBALPLATFORM, 2018) - especifica o TEE em *software* e *hardware*.

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)

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)

2.3.4 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.

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.

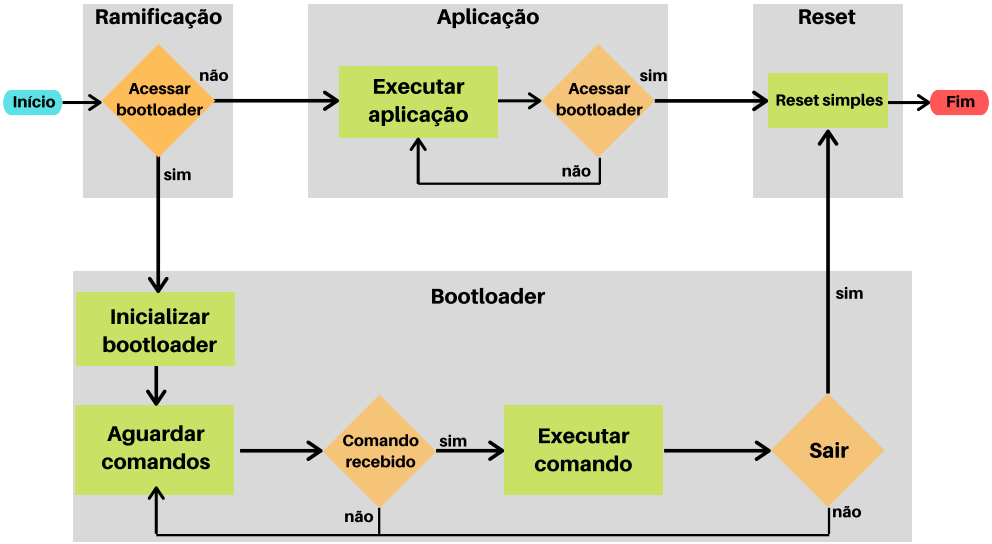
Para que a atualização de um dispositivo possa ser realizada sem a necessidade de um *hardware* externo, uma pequena aplicação - *bootloader* - deve ser embarcada no equipamento. O *bootloader* é uma aplicação que compartilha espaço da memória *flash* de um microcontrolador com o *firmware*, sendo capaz 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 possui 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* que este trabalho propõe utiliza os conceitos apresentados no Capítulo 2 para satisfazer os requisitos de segurança e confiabilidade da atualização. Para que o requisito de **segurança** seja atingido, deve-se utilizar criptografia de chave pública, empregando o uso de assinaturas digitais para garantir que um *firmware* foi gerado por uma fonte conhecida e confiável. Já o requisito de **confiabilidade** é atingido com o uso de funções de resumo criptográfico, utilizando o valor de *hash* do *firmware* que fora previamente armazenado e comparando com um valor de *hash* gerado a cada inicialização do dispositivo. Os dois valores de *hash* devem ser idênticos para que o *firmware* de dispositivo possa entrar em execução.

Uma assinatura digital válida garante as propriedades de autenticidade e integridade do *firmware*, portanto é possível transmitir para o dispositivo apenas a assinatura digital gerada pelo emissor do *firmware*, sem a necessidade de transmitir o resumo criptográfico do mesmo, desde que o resumo do *firmware* seja gerado e armazenado durante o processo de atualização, uma vez que a assinatura digital tenha sido validada.

Os processos de validação da assinatura digital do *firmware* durante o processo de atualização e de verificação do resumo criptográfico durante a inicialização do dispositivo são os pilares desta solução, sendo passos essenciais para que os requisitos de segurança e confiabilidade da atualização sejam alcançados.

Além das validações necessárias para uma atualização segura e confiável, é importante manter um versionamento dos *firmwares* que já foram instalados em um dispositivo, afim de que seja possível identificar *firmwares* que possuam um mal funcionamento e impedir que sejam instalados novamente, ou apenas impedir que um *firmware* de versão antiga seja instalado. Portanto, deve-se enviar para o dispositivo no momento do processo de atualização, não apenas a assinatura digital, mas também a versão do *firmware* que está sendo instalado. Essa versão deve ser armazenada no dispositivo e verificada a cada tentativa de instalação, devendo ser impossível que um *firmware* de versão antiga seja instalado. Pode-se também utilizar a versão do *firmware* como uma das entradas para a geração da assinatura digital enviada, garantindo que uma assinatura digital seja válida para um par *firmware* e versão específicos. Ao não utilizar a versão do *firmware* como entrada da assinatura digital, um atacante poderia obter um *firmware* antigo e sua respectiva assinatura digital, que é válida porém obsoleta, e enviá-los ao dispositivo juntamente de um valor de versão superior à versão do *firmware* já instalado.

Juntamente do *firmware* que será transmitido ao dispositivo, quatro informações são de grande importância para que a atualização seja realizada de forma segura e confiável: o resumo criptográfico do *firmware*, que pode ser transmitido ao dispositivo ou gerado no momento da atualização, a assinatura digital e a versão do *firmware*, que são enviadas ao dispositivo durante o processo de atualização, e a chave pública par da chave privada que assinou o *firmware*. Essas informações serão tratadas no decorrer do texto como informações essenciais ao processo de atualização.

O procedimento de atualização do *firmware* deve acontecer dentro do *bootloader* do dispositivo. Ele é responsável por identificar quando uma atualização foi requisitada de forma externa ou a atualização é necessária, uma vez detectado que o *firmware* do dispositivo não está íntegro. O *bootloader* também é responsável por liberar a execução do *firmware*, apenas quando uma atualização não esteja pendente e o *firwamre* esteja íntegro. O *bootloader*, por sua vez, deve ser imutável e deve ser executado a cada

inicialização do dispositivo, devendo ser impossível para um atacante impedir a sua execução.

As informações essenciais ao processo de atualização devem ser armazenadas em um ambiente seguro, isolado do *firmware* do dispositivo, de forma que o mesmo não possa ter acesso à elas. Apenas o *bootloader* deve ter acesso ou deve manipular essas informações, seja durante o processo de atualização ou durante a inicialização do dispositivo. Essas informações podem ser armazenadas, por exemplo, dentro de um elemento seguro, de forma que um *bootloader* sendo executado em microcontrolador que não tenha nenhuma característica de segurança faça chamadas à esse elemento seguro de acordo com o momento em que o processo de atualização se encontra. Pode-se ainda armazenar as informações essenciais em um *Trusted Execution Environment* (TEE), juntamente do *bootloader*, de forma que o *bootloader* seja executado no TEE e mesmo não precise fazer chamadas a um *hardware* externo para realizar a atualização do dispositivo.

Além da região no qual o *bootloader* é armazenado, é necessário definir regiões de memória para o armazenamento do *firmware* em execução, ou ainda do novo *firmware* recebido durante o processo de atualização. Essas regiões, também chamadas de partições, devem ser definidas de acordo com a quantidade de armazenamento que o dispositivo possui. Em microcontroladores com pouco espaço de armazenamento, pode não ser possível possuir mais de uma partição para armazenar o *firmware* instalado e o *firmware*. Nesses casos, uma partição única deve ser definida, e, em cada tentativa de atualização, o *firmware* em execução deve ser sobrescrito. Essa abordagem não traz prejuízos à solução de atualização de *firmware*, uma vez que, caso ocorra alguma falha na atualização, o *bootloader* continuará em execução, aguardando uma nova tentativa. Nos casos nos quais duas partições podem ser definidas no armazenamento do dispositivo, pode-se determinar uma partição de execução na qual o *firmware* do dispositivo deve sempre estar armazenado, e outra na qual o *firmware* recebido durante o processo de atualização ficará armazenado temporariamente, enquanto o *bootloader* não realiza as verificações necessárias para validar o *firmware* recebido. Uma vez que o *firmware* recebido tenha sido validado, o *bootloader* pode transferir o novo *firmware* da partição temporária para a partição definitiva. Ainda nos casos onde duas partições podem ser definidas, a transferência de um *firmware* da partição temporária para uma definitiva pode não ser necessária, desde que o *bootloader* saiba em qual partição o *firmware* antigo estava armazenado. Uma vez que o *bootloader* saiba a partição na qual o *firmware* antigo está em execução, basta esta informação ser armazenada de forma segura e ser alterada pelo *bootloader* a cada processo de atualização, fazendo uma simples troca em as partições em execução. Em todos os cenários possíveis de particionamento do armazenamento do dispositivo, o *bootloader* deve sempre remover o *firmware* recebido no momento da atualização uma vez que as verificações necessárias tenham falhado, além de remover ou impedir que um *firmware* antigo armazenado em alguma partição entre em execução.

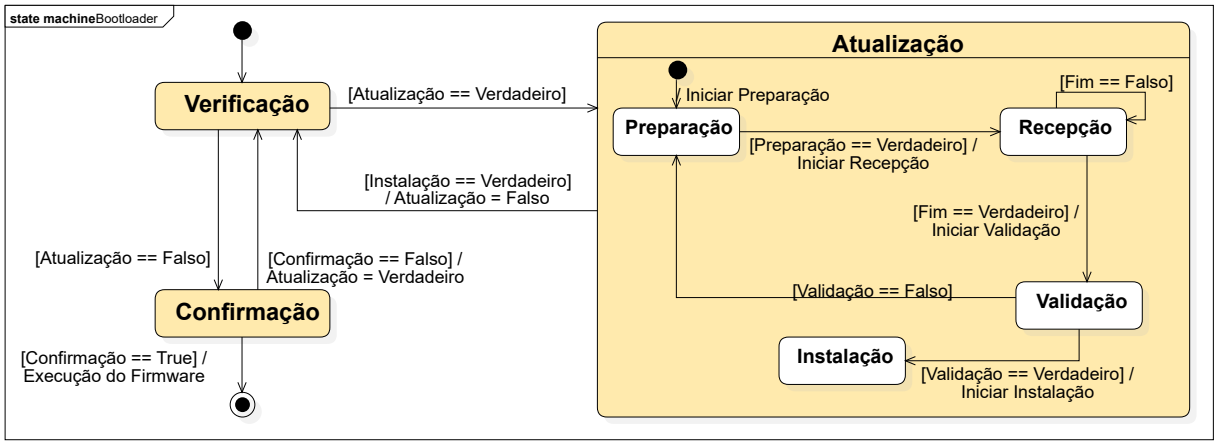
O gatilho para inicialização do processo de atualização do *firmware* do dispositivo pode ser acionado a partir do próprio *firmware*. Entretanto, deve ser possível acionar esse gatilho também a partir do *bootloader*, de forma que um possível *firmware* mal implementado não seja instalado e impeça o acionamento do processo de atualização. A indicação ao *bootloader* de que uma atualização é necessária deve ser armazenada em um ambiente de armazenamento seguro, de forma que o *firmware* do dispositivo não possa manipular essa informação, sendo ela de acesso exclusivo do *bootloader*.

3.1 Um modelo de *bootloader*

Um *bootloader* que atenda os requisitos de segurança e autenticidade da atualização pode ser modelado a partir de uma máquina de estado finita. A Figura 7 apresenta um diagrama de máquina de estados UML (OMG, 2017). Este modelo divide os estados do *bootloader* 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.

Figura 7 – Uma máquina de estados finitos UML para o *bootloader*.



Fonte: Própria.

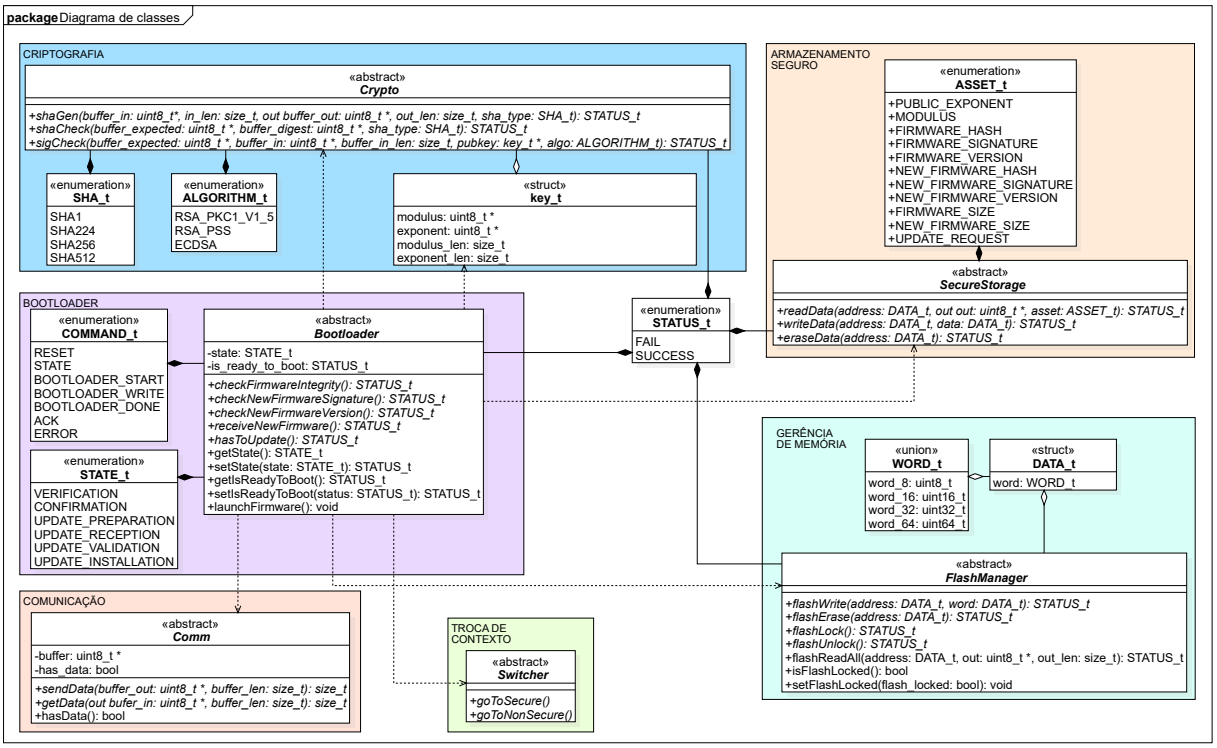
No estado inicial, Verificação, o *bootloader* precisa identificar se uma tentativa de atualização fora requisitada previamente. Essa verificação deve ser feita a partir do indicador de atualização de *firmware*. Caso o gatilho para que o processo de atualização de *firmware* seja iniciado tenha sido acionado, o *bootloader* deve então ir para o estado Atualização.

É no estado Atualização que o processo de atualização de fato ocorre. No sub-estado Preparação, o *bootloader* deve preparar o dispositivo para a recepção dos *firmware* e das informações essenciais ao processo de atualização. A preparação pode ocorrer na forma de limpeza da partição na qual o novo *firmware* será armazenado, bem como a limpezas das áreas seguras nas quais as informações essenciais serão armazenadas. No segundo sub-estado, Recepção, a transferência do novo *firmware* e das informações essenciais tem início. O *bootloader* deve armazenar os dados recebidos nas regiões adequadas e ficar neste estado até que a transferência tenha fim. No sub-estado Validação, o *bootloader* deve obter a versão do *firmware* recebido e fazer a comparação com a versão do *firmware* atual do dispositivo, devendo parar a atualização caso a versão do novo *firmware* seja inferior à versão do *firmware* já instalado. Em seguida, o *bootloader* deve validar a assinatura digital recebida no estado anterior. Assim como na verificação anterior, caso a assinatura digital recebida não seja válida, o *bootloader* deve interromper o processo de atualização atual e aguardar uma nova tentativa. Por fim, uma vez que as verificações sejam bem-sucedidas, o *bootloader* deve ir para o estado Instalação, podendo fazer a instalação do novo *firmware*, seja movendo o mesmo da partição temporária para a definitiva, ou seja alterando a indicação de qual partição detém o *firmware* que deve ser executado. Nos casos onde existe apenas uma partição no dispositivo, o sub-estado Instalação não é necessário.

Por fim, antes de passar o controle do dispositivo para o *firmware*, o *bootloader* deve passar pelo estado Confirmação. É nesse estado que o *bootloader* deve fazer a verificação da integridade do *firmware* instalado, a partir do resumo criptográfico do mesmo. Uma vez que seja identificado que o *firmware* não está íntegro, o *bootloader* deve ativar o indicador de que uma atualização é necessária e retornar ao estado inicial, de forma que a atualização seja iniciado. Caso contrário, o *bootloader* deve encerrar sua execução.

Além da modelagem da máquina de estados finita, é possível fazer uma modelagem de classes abstratas que implementem o comportamento do *bootloader* em cada estado. Essas classes abstratas são projetadas de forma que cada classe seja responsável por satisfazer um ponto específico do processo de atualização de *firmware*, ou seja, qualquer implementação de *bootloader* deverá de alguma forma implementar métodos similares ao métodos propostos em cada uma das classes abstratas. Por sua vez, o

Figura 8 – Diagrama de classes UML das classes abstratas propostas.



Fonte: Própria.

uso deste modelo de classes abstratas permite que um *bootloader* seja facilmente utilizado em mais de uma arquitetura ou microcontrolador, sendo necessário apenas especializar os métodos de cada classe que fazem a interação com o *hardware* do dispositivo. A Figura 9 apresenta o diagrama de classes UML (OMG, 2017) das classes abstratas propostas, que estão disponibilizadas no *GitHub*[1]

Para que a atualização de *firmware* seja possível, é necessário que exista uma interface de comunicação entre o dispositivo e o responsável pela transmissão do *firmware* e das informações essenciais ao processo de atualização. A comunicação entre o dispositivo e o transmissor pode acontecer de várias maneiras, tais como comunicação via *Universal Asynchronous Receiver and Transmiter* (UART) (Gupta et al., 2020), protocolo *Universal Serial Bus* (USB) (COMPAQ et al., 2000), *Bluetooth* (MARQUESS et al., 2019). Não é do escopo desse trabalho definir uma interface ou protocolo de comunicação específico para a comunicação do dispositivo. Entretanto, definem-se aqui métodos de envio e recepção de dados, que devem ser especializados de acordo com o protocolo de comunicação e com o microcontrolador utilizado.

A classe abstrata de criptografia é responsável por realizar s operações criptográficas necessárias para que os requisitos de segurança e confiabilidade da atualização possam ser contemplados. Nesta classe, portanto, é necessário implementar três métodos: uma para verificação de assinaturas digitais, um para a geração do resumos criptográficos e um para verificação de resumos criptográficos. O método de verificação de assinatura digital é utilizado no processo de atualização de *firmware*, enquanto que os métodos de geração e verificação de resumos criptográficos são executados a cada inicialização do dispositivo, a fim de verificar a integridade do *firmware* instalado.

A classe de gerência de memória tem como principal função gerenciar o acesso ao armazenamento do dispositivo e manipular as regiões de memória. As principais operações deste bloco são a escrita e leitura das regiões de memória. Desta forma, na classe que implementa este bloco definem-se métodos

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

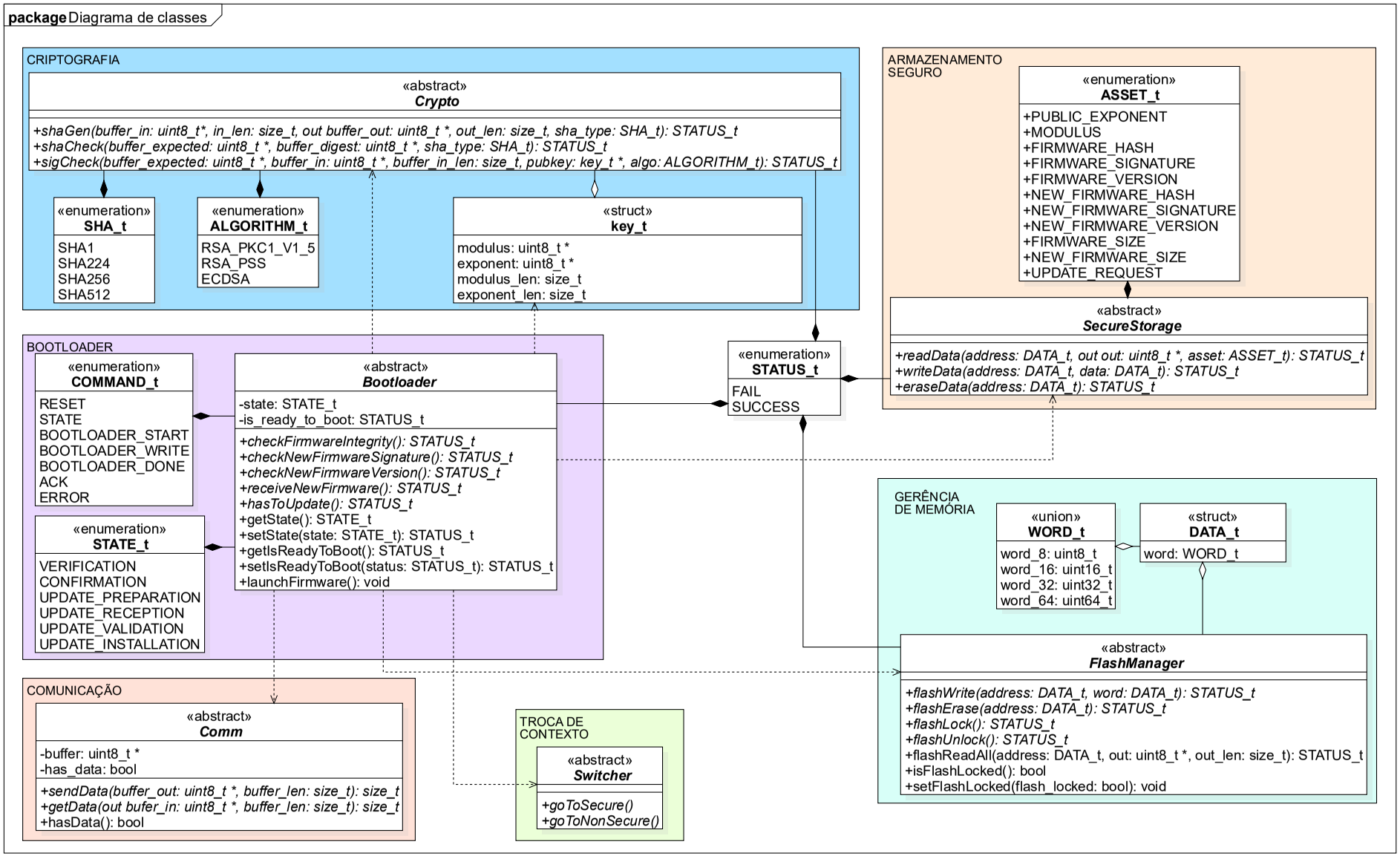
para cada uma dessas operações. É de responsabilidade do *bootloader* conhecer de antemão os endereços de memória que ele tem de escrever, ler ou apagar, de acordo com seu estado e o momento do processo de atualização ou inicialização que ele se encontra. Em alguns microcontroladores, antes de fazer a escrita em algum endereço de memória, é necessário fazer o desbloqueio do acesso à memória. Em razão disso, além dos métodos de escrita e leitura, é previsto nesta classe abstrata métodos para liberar o acesso ao armazenamento 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* do dispositivo e o local onde essas informações estão armazenadas. O ambiente seguro deve permitir a escrita e leitura das informações armazenadas. O ambiente seguro no qual essas informações são armazenadas não é definido por este trabalho, 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 especializa dentro do *firmware* do dispositivo. Sua função é fazer a troca de contexto entre o *bootloader* e o *firmware*, como por exemplo a chamada para que o *firmware* entre em execução a partir do *bootloader*. Propõe-se, na classe que implementa esse bloco, 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*, que um processo de atualização deve ser iniciado.

A classe do *bootloader* depende de todas as classes abstratas já citadas. Ela é responsável por implementar os processos de atualização e inicialização do dispositivo. Portanto, são propostos métodos para verificação da integridade do *firmware* atual, verificação da versão do *firmware* atual, bem como a versão do novo *firmware*, verificação da assinatura digital do novo *firmware*, recepção e armazenamento do novo *firmware*, assim como finalização do procedimento de instalação.

Figura 9 – Diagrama de classes UML das classes abstratas propostas.



Fonte: Própria.

4 IMPLEMENTAÇÃO DA SOLUÇÃO DE ATUALIZAÇÃO DE *FIRMWARE*

A fim de mostrar o funcionamento da solução de atualização de atualização de *firmware* proposta, as classes abstratas foram especializadas no microcontrolador STM32L562QEI6Q, da fabricante *STMicroelectronics*. Foi utilizado o *kit* de desenvolvimento *STM32L562-DK Discovery*, apresentado na Figura 10. A utilização deste *kit* se dá em função da integração do mesmo com os *softwares* 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 10, também possui *LEDs* e botões, interfaces que auxiliam o desenvolvimento junto ao microcontrolador. O projeto com a implementação da solução de atualização de *firmware* pode ser encontrado no *GitHub*[1].

Dentre as características do microcontrolador, estão seu *Core* baseado no *Cortex ARM M-33*, que possui o *ARM TrustZone* (a implementação de um *Trusted Execution Environment* (TEE)), 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.

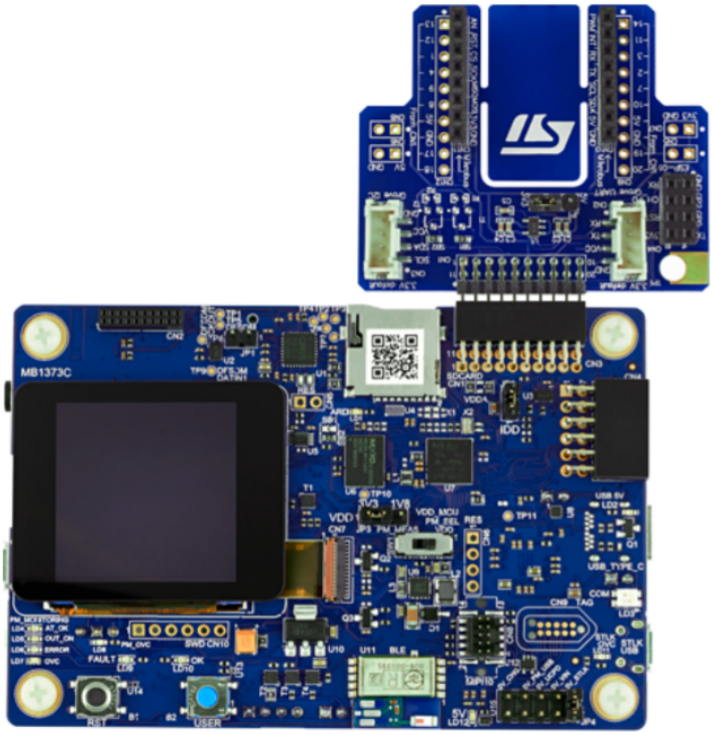
O *TrustZone* é a implementação da *ARM* de um TEE. Quando o *TrustZone* está habilitado, é possível separar o armazenamento do microcontrolador em uma área segura e uma área não segura. Um código sendo executado na área não segura da memória é totalmente isolado e não possui acesso à área segura da memória, 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 à toda memória do microcontrolador (STMICROELECTRONICS, 2020b).

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). Este mecanismo faz com uma área especifica da memória do microcontrolador (neste caso toda a àrea da memória composta pelo *bootloader*, bem como a área da memória onde a chave pública do emissor do *firmware* está armazenada) sejam imutáveis (STMICROELECTRONICS, 2020b). 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 *firmware*, optou-se por utilizar a classe *Communications Device Class* (CDC) do protocolo USB (USB-IF, 2010). Não é escopo deste trabalho apresentar com detalhes a especificação desta classe USB. Justifica-se o seu uso em função desta classe criar uma comunicação serial entre o microcontrolador e o transmissor do *firmware*, fazendo com que o estabelecimento da comunicação entre o par comunicante seja realizado de forma simples. A

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

Figura 10 – *Kit* de desenvolvimento *STM32L562-DK Discovery*.



Fonte: (STMICROELECTRONICS, 2021b).

fabricante do microcontrolador disponibiliza uma biblioteca de fácil compreensão e utilização, facilitando a especialização da classe abstrata de comunicação.

A classe de criptografia especializa 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*. Para o algoritmo de geração de resumos criptográficos, foi utilizado o algoritmo SHA-256 (NIST, 2015).

A classe de gerência da memória também foi especializada a partir de uma biblioteca própria da fabricante do microcontrolador. É na classe de gerência da memória que as definições de endereços de memória que indicam o início e fim de cada partição que armazena os *firmwares* foram realizadas. Além dos métodos obrigatórios definidos na classe abstrata correspondente, implementou-se também alguns métodos para facilitar o acesso a 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, é possível armazenar as informações essenciais ao processo de atualização de *firmware* e inicialização do microcontrolador na área da memória segura. Desta forma, um *firmware* sendo executado na área não segura da memória não tem acesso direto a essas informações. Por esta razão, 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.

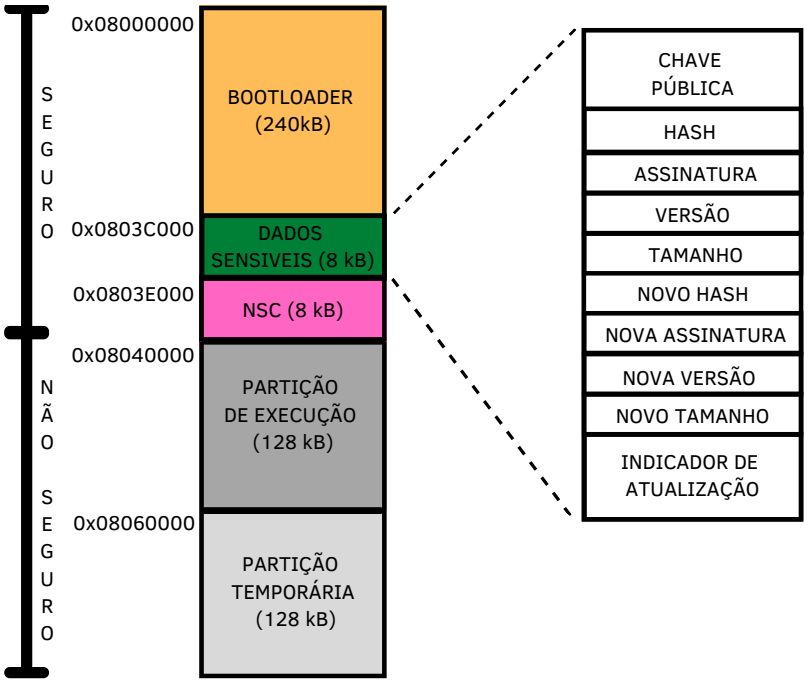
Dos dois métodos propostos na classe abstrata de troca de contexto, o *bootloader* implementa

apenas o que faz a troca de contexto entre o *bootloader* e o *firmware*. A especialização deste método foi feita a partir de um ponteiro para uma função que faz o salto da memória *flash* segura (*bootloader*) para a memória *flash* 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 o início da recepção, o responsável pela transferência deve mandar um comando indicando o início da transmissão, enviando um quadro no qual contém o comando de início da transmissão do *firmware*, o resumo criptográfico do novo *firmware*, a assinatura digital do emissor do *firmware*, o tamanho do novo *firmware* e sua versão, bem como o tamanho total da soma destas informações. Após o recebimento destas informações, o *bootloader* espera a transmissão do novo *firmware*. O transmissor, portanto, envia para o dispositivo um quadro iniciado pelo comando que informa a transmissão do novo *firmware*, bem como uma parte do novo *firmware* e o tamanho desta parte. A transmissão do novo *firmware* ocorre em quadros de até 480 *bytes*, até que o *firmware* seja transmitido por completo. Uma vez que a transmissão do *firmware* tenha sido finalizada, o transmissor avisa ao *bootloader*, enviando um comando de fim de transmissão. A partir do último comando recebido, o *bootloader* inicia as verificações para identificar a validade do novo *firmware*. Nesta implementação, foram utilizadas duas partições para armazenamento dos *firmwares*: uma temporária, que armazena o novo *firmware* antes da instalação, e uma partição de execução, no qual o *firmware* deve estar armazenado para ser executado. A Figura 11 apresenta a segmentação da memória do microcontrolador, indicando os ambientes de execução seguro e não seguro, bem como indicando as áreas no qual o *bootloader*, as informações sensíveis ao processo de atualização e inicialização, as chamadas *Non-Secure Callable* (NSC) e os *firmwares* são armazenados.

Cada *firmware* utilizado nesta implementação especializa a classe abstrata de troca de contexto,

Figura 11 – Segmentação da memória do microcontrolador.



Fonte: Própria.

afim de que o mesmo possa enviar um aviso ao *bootloader* quando uma atualização de *firmware* for requisitada. 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. No contexto desta solução de atualização, o *firmware*, a partir do acionamento de um gatilho externo, faz a chamada NSC ao *bootloader*, informando que uma atualização de *firmware* deve ser iniciada. O *bootloader*, por sua vez, escreve em um indicador armazenado na região de memória segura que uma atualização de *firmware* foi solicitada, e reinicia o microcontrolador. Esse indicador só é apagado pelo *bootloader* uma vez que uma atualização tenha sido bem-sucedida.

4.1 Experimentos e resultados

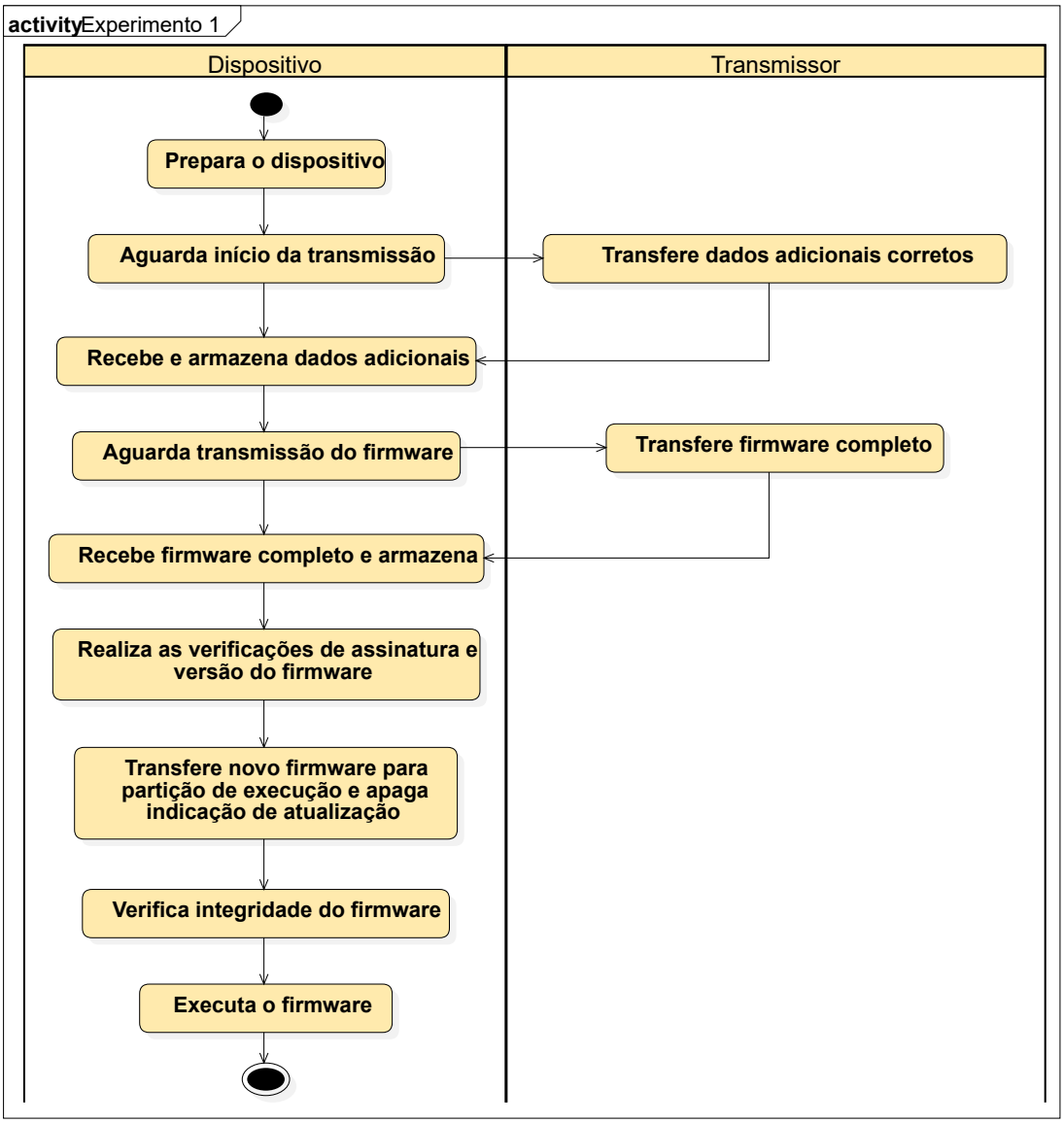
Afim de verificar o funcionamento da implementação da solução proposta no Capítulo 3, alguns experimentos foram realizados. No cenário de testes, um computador realizou o papel de transmissor do novo *firmware*, a partir de programas desenvolvidos com a linguagem de programação *Python3*. Os códigos de cada programa podem ser encontrados *GitHub*[2]. Para facilitar a identificação da mudança dos *firmwares* após o processo de atualização, foram desenvolvidos dois *firmwares* 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. Ambos os *firmwares* implementam a classe de troca de contexto, tendo como gatilho o acionamento de um botão presente no *kit* de desenvolvimento utilizado.

O primeiro experimento realizado para verificar o funcionamento da implementação foi uma tentativa de atualização bem-sucedida. O comportamento do dispositivo e do transmissor são apresentados na Figura 12. Como estado inicial, o dispositivo possuia o *firmware* Tipo 1 sendo executado. Após a solicitação de tentativa de atualização e o dispositivo ser reiniciado, o *bootloader* estava pronto para o recebimento do novo *firmware*. Foi enviado para o dispositivo os dados adicionais corretos (resumo criptográfico, assinatura digital, tamanho e versão do *firmware* Tipo 2). Após a confirmação de recebimento destas informações, transferiu-se o *firmware*. Uma vez que a transferência foi finalizada, o *bootloader* realizou as verificações de versão e assinatura, e após a validação das verificações, instalou o novo *firmware* com sucesso. Por fim, o *bootloader* encerrou sua execução, fazendo a verificação da integridade do novo *firmware* antes de passar o controle do dispositivo para o mesmo. Conclui-se que esse teste foi executado com sucesso e com resultado final esperado.

O segundo experimento foi a tentativa de atualização de *firmware* com a versão do novo *firmware* sendo inferior à versão do *firmware* sendo executado no microcontrolador. Esse experimento tem como resultado esperado a falha na atualização, seguida da espera pelo *bootloader* de uma nova tentativa de atualização. O dispositivo tinha como estado inicial o *firmware* Tipo 1 sendo executado, e como versão o valor 2 armazenado no endereço de memória correspondente. A partir do acionamento de um botão no *kit* de desenvolvimento, o pedido de atualização foi solicitado ao *bootloader*. Neste teste, foram enviados ao dispositivo os valores corretos de resumo criptográfico, assinatura digital e tamanho do *firmware* Tipo 2. Entretanto, enviou-se como valor da versão do *firmware* o valor 1, inferior à versão instalada. Após o envio das informações essenciais ao processe de atualização e do *firmware*, o *bootloader* iniciou as verificações necessárias, falhando na validação da versão do *firmware* recebido. Desta forma, o *bootloader* não finalizou a instalação e aguardou uma nova tentativa, conforme esperado inicialmente. Em seguida, uma nova tentativa de atualização fora iniciada, dessa vez enviando ao dispositivo uma versão de *firmware* superior à

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

Figura 12 – Diagrama de atividades do experimento 1.

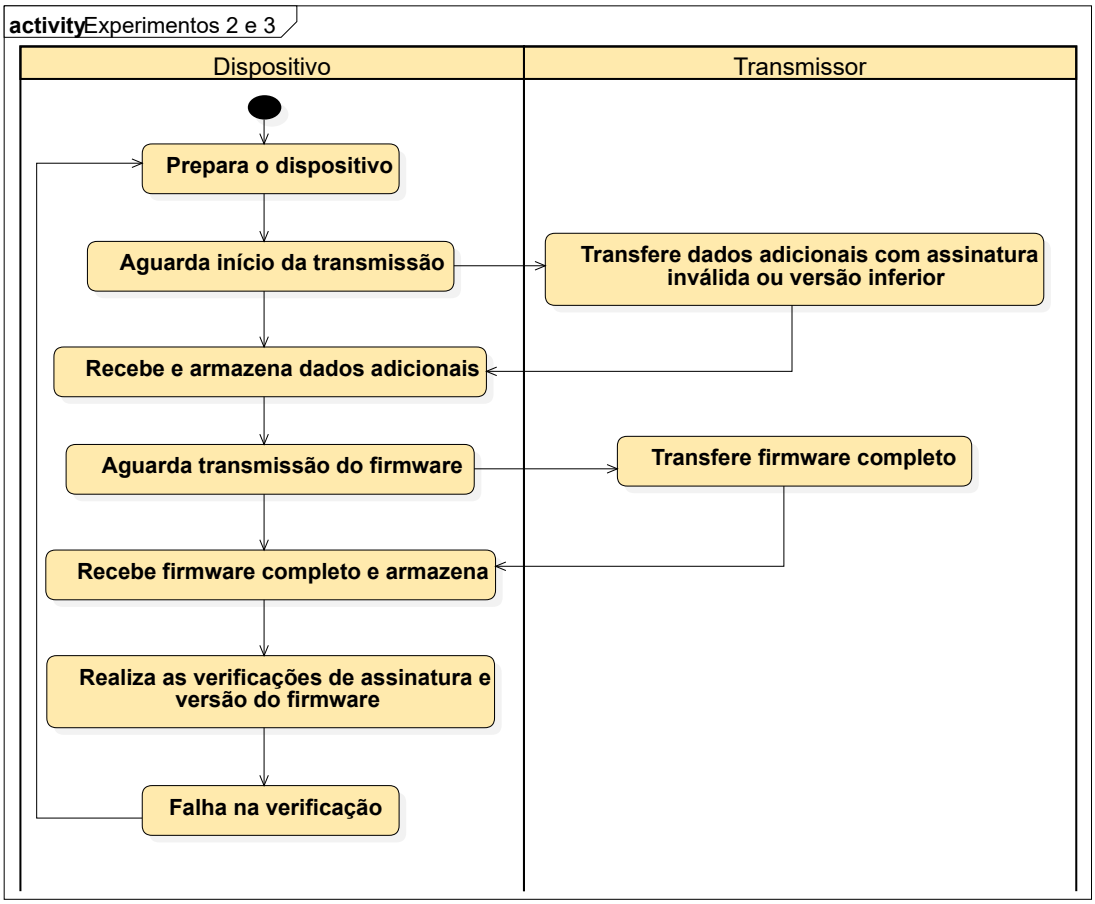


Fonte: Própria.

versão já instalada. Após a finalização das transferências e das verificações, o *bootloader* finalizou com êxito a tentativa de atualização, validando a integridade do *firmware* e colocando o mesmo 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 como resultado uma falha na tentativa de atualização, uma vez que o *bootloader* não deve instalar um *firmware* no dispositivo sem fazer a validação da assinatura digital. O dispositivo tinha como estado inicial o *firmware* Tipo 1 sendo executado. Uma vez que o *bootloader* estava pronto, o dispositivo recebeu do transmissor as informações essenciais ao processo de atualização, que enviou de forma correta o resumo criptográfico, tamanho e versão do *firmware* Tipo 2, porém enviou uma assinatura digital gerada sobre o *firmware* Tipo 1. Conforme esperado, após o recebimento das informações essenciais e do *firmware* Tipo 2, o *bootloader* iniciou as verificações, passando com sucesso na verificação da versão do *firmware* e falhando na verificação da assinatura digital, uma vez que a mesma fora gerada sobre um *firmware* diferente do recebido pelo dispositivo. Sendo assim, o *bootloader* não finalizou a instalação, aguardando que uma

Figura 13 – Diagrama de atividades dos experimentos 2 e 3.



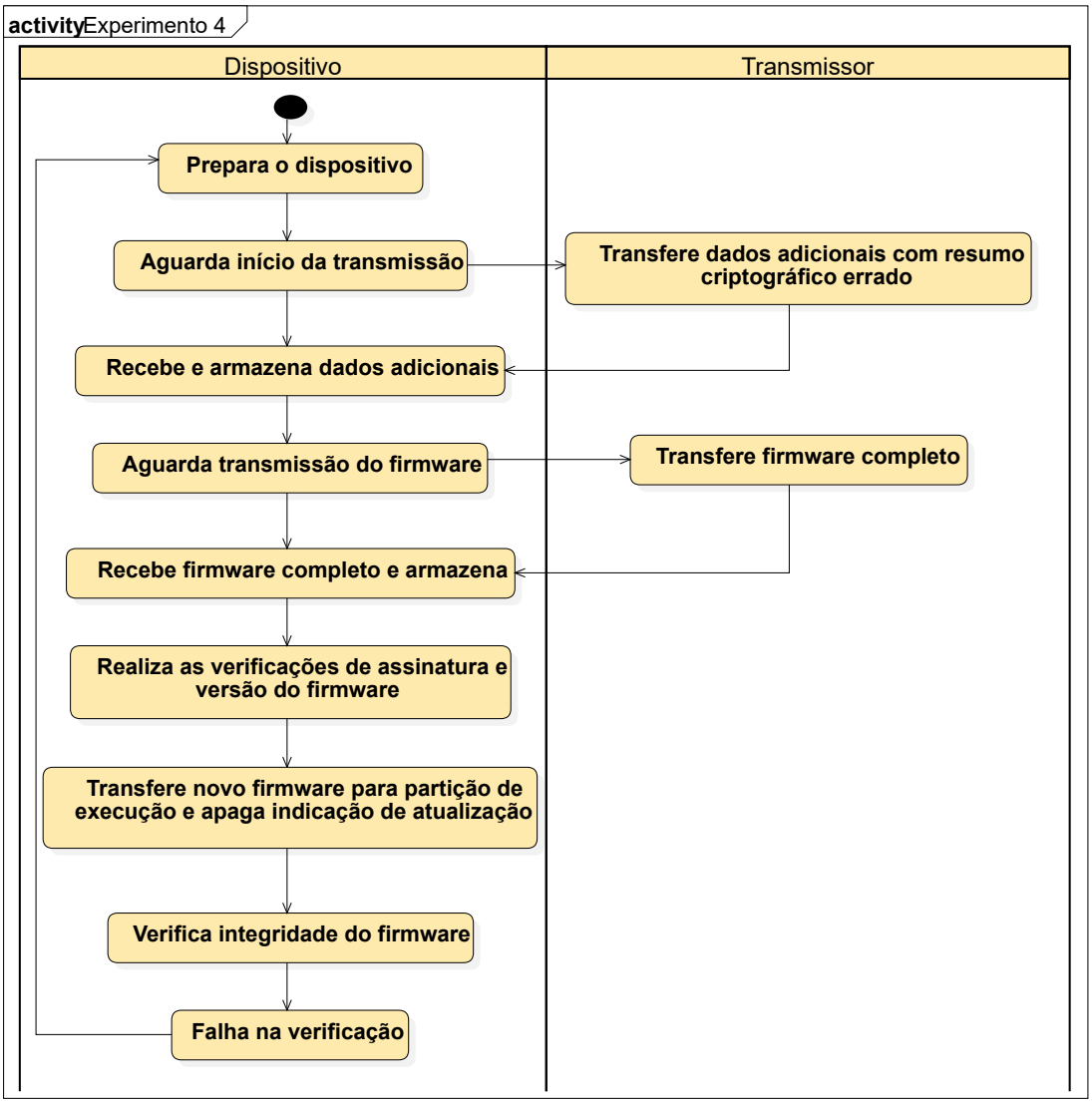
Fonte: Própria.

nova tentativa fosse iniciada. Na nova tentativa de atualização, o *host* enviou ao dispositivo a assinatura correspondente ao *firmware* Tipo 2, além das informações essenciais e o próprio *firmware*. Uma vez finalizada as transferências e as verificações do *bootloader*, a atualização foi bem-sucedida, novamente demonstrando a recuperação do *bootloader* após uma tentativa de atualização falha. Os comportamentos do dispositivo e do transmissor no segundo e no terceiro experimento são apresentados na Figura 13, uma vez que a única diferente do comportamento do transmissor nesses dois experimentos é o envio de uma informação essencial ao processo de atualização de forma errada.

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, com o comportamento do dispositivo e do transmissor sendo apresentado na Figura 14. Inicialmente, é executado no dispositivo o *firmware* Tipo 1. Após iniciado o processo de atualização, as informações essenciais corretas referentes ao *firmware* Tipo 2 são enviadas ao dispositivo, com exceção do resumo criptográfico, que foi substituído por 32 *bytes* aleatórios. Com o término da transferência do novo *firmware*, o *bootloader* inicia as verificações, validando a instalação. Finalmente, o *bootloader* verifica a integridade do *firmware* instalado, comparando o resumo criptográfico do *firmware* 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, conforme esperado.

O último experimento verifica o comportamento do *bootloader* a partir de uma tentativa de atualização iniciada, porém não finalizada. A Figura 15 apresenta o comportamento do dispositivo e

Figura 14 – Diagrama de atividades do experimento 4.



Fonte: Própria.

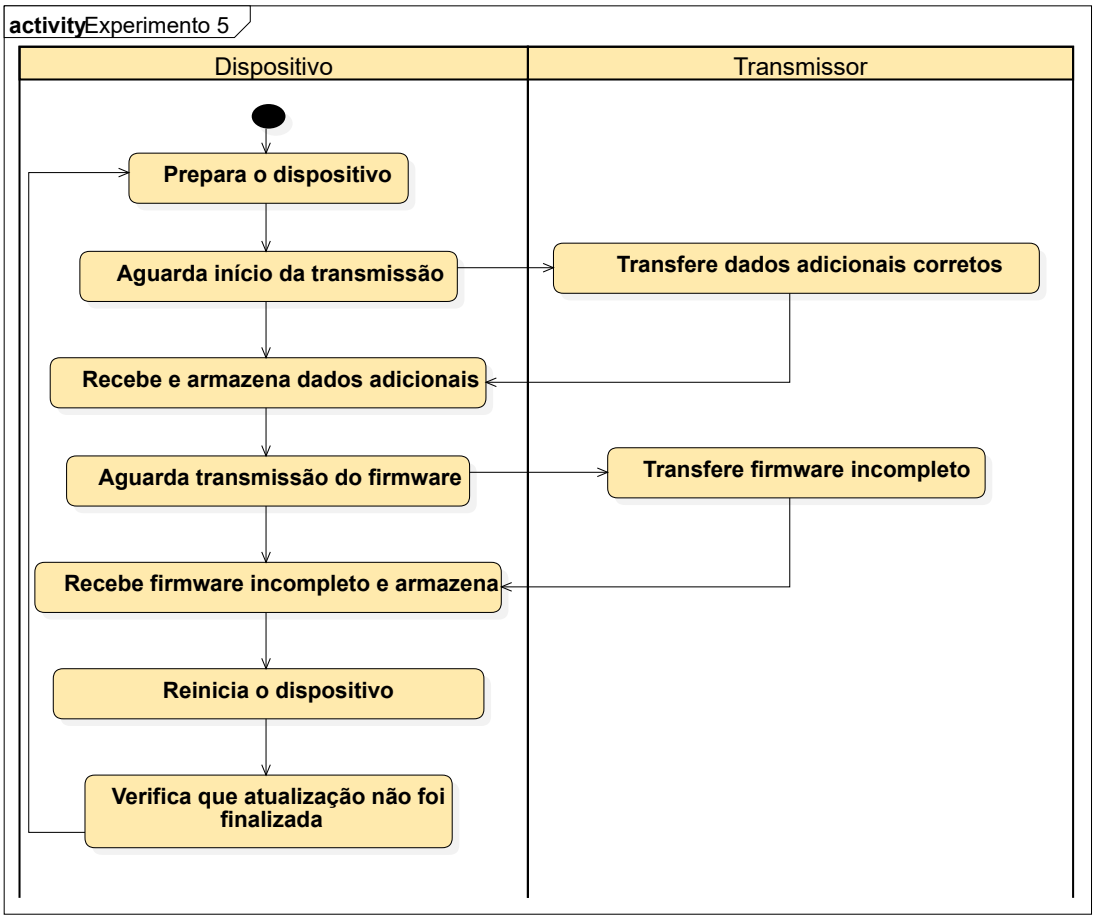
do transmissor para esse experimento. Com o *firmware* Tipo 1 sendo executado, uma tentativa de atualização foi iniciada. O transmissor enviou os dados adicionais corretos do *firmware* Tipo 2, porém após iniciar a transferência do novo *firmware*, o transmissor interrompeu a transferência. Após isso, 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* aguardou para que uma nova tentativa de atualização tivesse início.

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 15 – Diagrama de atividades do experimento 5.



Fonte: Própria.

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. Um resumo das características de cada teste e de cada resultado é apresentado na Tabela 5.

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 *firmwares* 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.

O requisito de segurança foi atingido a partir do uso de criptografia de chave pública, utilizando a chave privada do emissor do *firmware* para gerar uma assinatura digital sobre o código do *firmware*, em conjunto com a versão do mesmo. Assinar a versão do *firmware* juntamente do código do mesmo também garante que um *firmware* com versão inferior à versão do *firmware* já sendo executado no dispositivo não seja instalado.

Já o requisito de confiabilidade, por sua vez, foi alcançado a partir do uso de funções de resumo criptográfico, utilizando o resultado destas funções para realizar uma verificação da integridade do *firmware*. A cada inicialização do dispositivo, gera-se o resumo criptográfico do *firmware* instalado o resultado deste processamento é comparado com um valor de resumo criptográfico previamente armazenado. Caso os valores comparados sejam diferentes, o dispositivo não permite a execução do *firmware*, demandando uma nova tentativa de atualização.

Este trabalho disponibiliza um conjunto de classes abstratas que devem ser especializadas para cada plataforma a qual as classes forem utilizadas. Por serem classes abstratas, a portabilidade entre diferentes plataformas se dá de maneira facilitada, uma vez que as classes já preveem cada método que deve ser implementado para que a solução de atualização proposta seja executada de forma correta, bem como indicam o papel de cada método no processo de atualização. Estas classes abstratas tem como objetivo fazer uma modelagem da solução de atualização de *firmware* proposta, englobando todos os requisitos necessários que fazem com que uma atualização de *firmware* seja segura e confiável.

Na implementação da proposta, utilizou-se um microcontrolador seguro que possui um *Trusted Execution Environment* (TEE). O TEE é divido em áreas segura e não segura, permitindo que o *bootloader* e as informações essenciais que fazem parte da atualização do dispositivo, como resumo criptográfico e assinatura digital do *firmware* pudessem ser armazenadas de forma isolada do *firmware* do dispositivo. O *bootloader* e as informações essenciais ao processo de atualização foram armazenados na área segura do microcontrolador, enquanto que a área não segura foi reservada ao *firmware*. O microcontrolador utilizado possui alguns mecanismos de segurança adicionais, que permitem que o *bootloader* seja imutável e sempre execute a cada inicialização do dispositivo, bem como impede que a chave pública do emissor do *firmware* seja alterada. Esta implementação especializou o conjunto de classes abstratas de acordo com o microcontrolador escolhido, utilizando algumas bibliotecas disponibilizadas pelo fabricante do microcontrolador.

Testes foram realizados com a implementação desenvolvida, afim de verificação a robustez da proposta de atualização. O primeiro teste verificou que uma atualização é bem-sucedida quando as informações sensíveis ao processo de atualização, como versão, assinatura digital, tamanho e resumo criptográfico, bem como o código do *firmware* são transmitidos ao dispositivo de forma correta. Por outro lado, verificou-se que quando alguma informação essencial à atualização, como versão ou assinatura digital, é enviada de forma errada, o *bootloader* não permite que a instalação seja realizada no dispositivo. Além

disso, quando o resumo criptográfico do *firmware* foi enviado ao dispositivo de forma errada no processo de atualização, o *bootloader* executa com sucesso a atualização, entretanto, o *firmware* instalado não entra execução, falhando no processo de verificação de íntegridade do *firmware*. Por fim testou-se uma tentativa de atualização na qual a transmissão do *firmware* foi interrompida na metade do processo. Neste teste verificou-se que o *bootloader* retorna ao estado na qual ele aguarda uma nova tentativa de atualização, após reiniciar o dispositivo. Os testes realizados demonstraram que a proposta de solução de atualização de *firmware* apresentada neste trabalho permite que as requisitos de segurança e confiabilidade sejam contempladas.

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, afim 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, afim 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 248kB 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.

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 25.

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 3 vezes nas páginas 26, 27 e 28.

BISHOP, M.; BAILEY, D. *A Critical Analysis of Vulnerability Taxonomies*. [S.l.], 1996. Citado na página 19.

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 23, 24, 25 e 26.

COMPAQ et al. *Universal Serial Bus Specification*. [s.n.], 2000. Universal Serial Bus Specification Revision 2.0. Disponível em: <https://www.usb.org/document-library/usb-20-specification>. Citado na página 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 25.

Gupta, A. K. et al. Design and implementation of high-speed universal asynchronous receiver and transmitter (uart). In: *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)*. [S.l.: s.n.], 2020. p. 295–300. Citado na página 32.

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 17 e 26.

JOHNER, H. et al. *Deploying a Public Key Infrastructure*. [S.l.]: IBM, 2000. ISBN 9780738415734. Citado 2 vezes nas páginas 21 e 22.

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 36.

LANDWEHR, C. Computer security. *International Journal of Information Security*, v. 1, p. 3–13, 2001. Citado na página 19.

LEE, R. B. *Security Basics for Computer Architects*. [S.l.]: Morgan & Claypool Publishers, 2013. ISBN 9781627051569. Citado 4 vezes nas páginas 17, 19, 20 e 22.

MARQUESS, K. et al. *Bluetooth Core Specification*. [S.l.], 2019. Citado na página 32.

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 19.

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 17.

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 36.

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 24.

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 30 e 32.

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 25.

RUSSEL, D.; GANGEMI, G. T. *Computer Security Basics*. [S.l.]: O'Reilly & Associates, 1991. ISBN 0-937175-71-4. Citado na página 19.

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 24.

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 19.

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 17, 19, 20, 21 e 22.

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 37.

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 35.

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 35.

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 36.

USB-IF. *Universal Serial Bus Class Definitions for Communications Devices*. [S.l.]: USB-IF, 2010. Revision 1.2 (Errata 1.0). Citado na página 35.

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);
  virtual size_t getData(uint8_t *buffer_in, size_t * buffer_len);
  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);
  virtual void goToNonSecure(void);
};

#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);

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

  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);
  virtual STATUS_t flashErase(DATA_t address);
  virtual STATUS_t flashLock(void);
  virtual STATUS_t flashUnlock(void);

  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);
  virtual STATUS_t writeData(DATA_t address, DATA_t data);
  virtual STATUS_t eraseData(DATA_t address);

};

#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);
  virtual STATUS_t checkNewFirmwareSignature(void);
  virtual STATUS_t checkNewFirmwareVersion(void);
  virtual STATUS_t receiveNewFirmware(void);
  virtual STATUS_t hasToUpdate(void);
  virtual void launchFirmware(void);

  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
```