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

Paulo Fylippe Sell[1], **Emerson Ribeiro de Mello**[1], **Roberto de Matos**[1]

[1]Instituto Federal de Santa Catarina (IFSC)
São José – SC – Brasil

paulo.fs@aluno.ifsc.edu.br, {mello, roberto.matos}@ifsc.edu.br

Resumo. *Uma atualização confiável de firmware garante que um dispositivo não ficará inutilizável ao final do processo. Uma atualização segura garante que somente firmware autênticos possam ser instalados no dispositivo. Fabricantes de microcontroladores possuem soluções próprias para uma atualização segura e confiável, porém cada solução possui particulares para uso. Este trabalho apresenta uma solução genérica para atualização segura e confiável de firmware de microcontroladores. Foi realizada uma implementação referência com o microcontrolador STM32L562QE de forma a validar a solução proposta e para demonstrar que é possível obter atualização segura e confiável sem depender de soluções proprietárias dos fabricantes de microcontroladores.*

Abstract. *A reliable firmware update ensures that a device will not become unusable at the end of the process. A secure update ensures that only authentic firmware can be installed on the device. Microcontroller manufacturers have their own solutions for a secure and reliable update, however, each solution is tailored to its devices. This work presents a generic solution for a secure and reliable firmware update process. A reference implementation was performed using the STM32L562QE microcontroller in order to validate the proposed solution and to demonstrate that it is possible to obtain a safe and reliable upgrade without relying on proprietary solutions from microcontroller manufacturers.*

1. Introdução

Mecanismos que permitem a atualização de *firmware* de forma remota poupam tempo e dinheiro, tanto do fabricante do dispositivo quanto do usuário, uma vez que o dispositivo não precisaria ser encaminhado ao fabricante para realizar o procedimento de atualização sempre que uma nova versão for lançada, seja para corrigir problemas ou para incluir novas funcionalidades ao mesmo [Jain et al. 2016]. Dispositivos conectados à Internet podem possuir soluções automatizadas que permitem realizar a atualização de forma espontânea, ou seja, sem qualquer intervenção do usuário. Porém, para os dispositivos que não possuem tal facilidade, o usuário é o responsável por baixar manualmente o arquivo do novo *firmware*, disponibilizado por exemplo no *site web* do fabricante, e realizar manualmente o processo de atualização no dispositivo.

A fim de que seja possível atualizar o *firmware* do dispositivo, uma pequena aplicação, chamada *bootloader*, deve ser embarcada no equipamento. O *bootloader* é invocado sempre que o dispositivo é energizado e seu principal papel é carregar o *firmware*

instalado, sendo este último a aplicação que contém toda a lógica com as funcionalidades que aquele dispositivo deverá prover. O *bootloader* também pode ser responsável pelo processo de atualização de *firmware* e deve garantir que a atualização seja confiável e, em alguns cenários, segura. Para tanto, o *bootloader* deve ser imutável, além de ser capaz de se comunicar com outros dispositivos para que seja possível receber novos *firmware* [Beningo 2015].

Um processo de atualização de *firmware* é considerado confiável se, ao término do processo, o dispositivo conseguir reiniciar corretamente com o novo *firmware*, em caso de sucesso na atualização, ou executar o *firmware* previamente instalado, em caso de insucesso [Nikolov 2018]. O *bootloader* é o responsável por receber o novo *firmware*, verificar se o mesmo está íntegro, usando por exemplo funções *hash* criptográficas, e por fim, atualizar os ponteiros internos para que o novo *firmware* seja usado nas próximas inicializações do dispositivo.

De outro modo, um processo de atualização de *firmware* é considerado seguro se fizer uso de mecanismos que garantam que somente *firmware* autênticos possam ser instalados no dispositivo [Nikolov 2018, Landwehr 2001, Russel and Gangemi 1991]. A autenticidade de um *firmware* pode ser verificada, por exemplo, por meio da criptografia de chave pública. Nesse caso, a chave pública do emissor do *firmware* deve ser inserida na memória do dispositivo e se deve garantir que a mesma não possa ser substituída por outra chave sem a aprovação do fabricante do dispositivo. Essa garantia pode ser atingida utilizando um *hardware* seguro, como um *Secure Element* (SE) ou um *Trusted Execution Environment* (TEE), que proveem armazenamento e execução segura de aplicações e dados, prevenindo que ataques que visam obter ou modificar uma informação sejam bem-sucedidos [Bouazzouni et al. 2018]. Por fim, no processo de atualização se deve encaminhar o novo *firmware* juntamente com uma assinatura que fora gerada pela chave privada, par da chave pública que está embarcada no dispositivo.

Fabricantes de microcontroladores disponibilizam soluções proprietárias para a atualização de *firmware* e que podem ser utilizadas por desenvolvedores. Essas soluções proveem um conjunto de APIs que devem ser utilizadas para que o desenvolvedor integre o *bootloader* do dispositivo às soluções de atualização de *firmware*. Entretanto, essas soluções não são interoperáveis entre os fabricantes, podendo ainda haver casos em que uma solução pode ser limitada a famílias específicas de microcontroladores dentro de um mesmo fabricante, de forma que o desenvolvedor seja obrigado a alterar integralmente o seu projeto de *bootloader* cada vez que utilizar um microcontrolador diferente [STMicroelectronics 2020b]. Por outro lado, uma solução de atualização de *firmware* genérica permite que fabricantes de dispositivos possam utilizar uma mesma implementação de *bootloader* ao mesmo tempo que microcontroladores diferentes sejam utilizados, sem que haja a necessidade de alterar o projeto do *bootloader* como um todo.

Este trabalho apresenta uma solução genérica para atualização de *firmware* de dispositivos embarcados de forma segura e confiável. A solução disponibiliza um projeto de *bootloader* que pode ser reaproveitado em diferentes microcontroladores, uma vez que o mesmo foi organizado em um conjunto de classes abstratas, sendo necessário apenas especializar os métodos que fazem a interação com o *hardware* dos microcontroladores, por exemplo, métodos que fazem leitura e escrita na memória de programa do microcontrolador. Uma implementação de referência foi realizada utilizando o microcontrolador

STM32L562QE [STMicroelectronics 2020c], a qual permitiu demonstrar que a solução proposta é capaz de atualizar o *firmware* do dispositivo de forma que apenas *firmware* autênticos fossem instalados, ao passo que o dispositivo não entrou em um estado de execução não previsto, mesmo quando uma atualização não fora concluída com êxito.

Este trabalho está organizado da seguinte forma: na Seção 2 são apresentados trabalhos relacionados a este; a Seção 3 apresenta a proposta de atualização de *firmware* deste trabalho; na Seção 4 é apresentada a implementação realizada, enquanto que os experimentos e resultados obtidos são apresentados na Seção 5; por fim, na Seção 6 são apresentadas as conclusões obtidas.

2. Trabalhos relacionados

Em [Jain et al. 2016] é proposta uma solução de atualização de *firmware* que faz uso de criptografia simétrica para cifrar o *firmware* a ser transmitido e assim garantir a propriedade de confidencialidade. O *firmware* é transmitido para o dispositivo em blocos e é feito uso do *Cyclic Redundancy Check* (CRC) para para verificar a integridade de cada bloco. Os autores optaram por segmentar a memória do microcontrolador em duas partições, uma para conter o *firmware* em execução e outra temporária, para armazenar o novo *firmware* a ser recebido. Essa abordagem permite que o dispositivo sempre consiga iniciar um *firmware*, mesmo que uma tentativa de atualização tenha sido encerrada com falha.

O trabalho de [Nikolov 2018] tem como principal foco a proposição de uma solução para transmissão do novo *firmware* por meio da comunicação sem fio, ou *Firmware-over-the-air* (FOTA). Neste cenário, o dispositivo possui conectividade com a Internet e assim é capaz de iniciar o processo de atualização de *firmware* de forma automática. Para garantir a confiabilidade do processo de atualização, o autor optou pela abordagem de segmentação da memória do microcontrolador em duas partições e o uso de soma de verificação (*checksum*) para verificar a integridade do *firmware* recebido. Porém, o trabalho não faz uso de mecanismos para garantir a autenticidade deste *firmware*.

Em [Dhobi et al. 2019] é feito uso de TEE como *hardware* seguro e as rotinas do processo de atualização de *firmware* foram dividas em duas aplicações, uma armazenada no ambiente seguro e outra no ambiente não seguro. No ambiente seguro estão as rotinas para verificar a autenticidade e integridade do *firmware* e para isso é feito uso de criptografia de chave pública RSA combinada com o algoritmo de resumo criptográfico SHA-256. Na área não segura estão as rotinas para recebimento do *firmware* via FOTA, uma vez que o ambiente seguro do TEE não possui conectividade com a Internet.

No presente trabalho optou-se pelo uso da criptografia de chave pública para garantir a integridade e autenticidade do novo *firmware*, contudo a cifragem do mesmo não foi considerada como um requisito essencial. Assim, foi proposto o uso de *hardware* seguro para o armazenamento da chave pública do emissor do *firmware*, bem como de todo o código *bootloader*. Com relação a confiabilidade do processo de atualização, ou seja, a garantia de que o dispositivo não ficará inutilizado após uma tentativa falha de atualização, este trabalho propõe uma máquina de estados e uma organização de classes para implementação do *bootloader* permitindo a ao desenvolvedor escolher entre a abordagem com uma única partição para armazenamento do *firmware* e a abordagem com duas partições, sendo uma partição temporária para armazenamento do novo *firmware* .

Dessa forma, acredita-se que a presente proposta seja adequada até mesmo para dispositivos com pouca área de armazenamento, mas ainda assim garantindo uma atualização segura e confiável.

3. Solução para atualização de *firmware* de forma segura e confiável

A solução de atualização de *firmware* apresentada neste trabalho é destinada a dispositivos com sistemas embarcados compostos por componentes com recursos de armazenamento e processamento limitados (*e.g.* microcontroladores) e com função específica. A proposta consiste na definição de: i) um conjunto mínimo de componentes de *hardware*, bem como sua organização; ii) os mecanismos e artefatos essenciais para garantir as propriedades de integridade e autenticidade do *firmware*, além da confiabilidade do processo de atualização; iii) a estruturação do arquivo binário que conterá o novo *firmware* a ser enviado ao dispositivo; iv) organização e conjunto de funcionalidades do *bootloader*.

3.1. Componentes de hardware

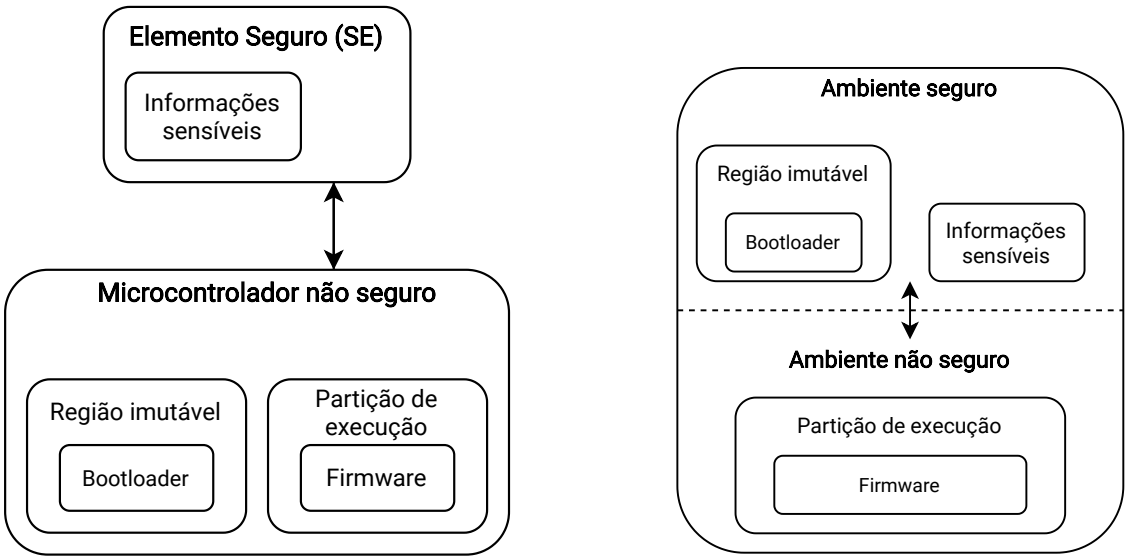
O *hardware* do dispositivo deve garantir as seguintes prerrogativas para o perfeito funcionamento do *bootloader* e da proposta aqui apresentada: i) armazenamento seguro – garantir que as informações essenciais utilizadas pelo *bootloader* durante os processos de atualização e inicialização do dispositivo sejam acessadas com exclusividade pelo *bootloader*, que não estejam expostas para o *firmware* e que não possam ser extraídas de maneira não autorizada; ii) garantia da imutabilidade do *bootloader* – o mesmo não deve ser alterado após ser inserido no dispositivo; iii) a execução do *bootloader* deve ser prioritária – este sempre deve ser o primeiro código a entrar em execução no dispositivo a cada inicialização.

A garantia de exclusividade de leitura e escrita pode ser alcançada com a utilização de *hardware* seguros. É possível integrar um *Secure Element* (SE) em conjunto com um microcontrolador não seguro, de modo que um *bootloader* armazenado no microcontrolador busque as informações essenciais no elemento seguro sempre que necessário. Uma segunda organização possível consiste em usar um *Trusted Execution Environment* (TEE), sendo possível guardar no ambiente de armazenamento seguro, além do material criptográfico sensível, o próprio *bootloader*. Neste cenário, tanto o *bootloader* quanto as informações essenciais aos processos de atualização e inicialização são armazenadas e executadas de forma isolada ao *firmware* do dispositivo. Na Figura 1(a) é apresentada a organização de *hardware* do dispositivo utilizando um microcontrolador não seguro em conjunto com um SE, enquanto que na Figura 1(b) é apresentada uma organização de *hardware* do dispositivo utilizando um TEE, sendo esta última a organização utilizada na implementação realizada e discutida na Seção 4.

O *firmware* do dispositivo deve ser armazenado em uma região de memória na qual o *bootloader* possa ler e escrever. Essa região, também chamada de partição, deve ser definida de acordo com a área de armazenamento que o dispositivo possui. Em dispositivos com pouco espaço de armazenamento, deve-se definir uma única partição e a cada tentativa de atualização o *bootloader* deverá sobrescrever o conteúdo dessa partição com o novo *firmware*, aqui chamado de *firmware* candidato, e se o processo ocorrer com sucesso, então o *firmware* candidato deverá ser executado na próxima inicialização do dispositivo. Contudo, caso a atualização não seja bem-sucedida, então o *bootloader* não

deve colocar o *firmware* candidato em execução e deverá indicar ao usuário (*e.g.* *led* piscando em vermelho) que alguma ação precisa ser tomada.

Quando a área de armazenamento for grande o suficiente para permitir a criação de duas partições, então pode ser definido uma partição de execução, a qual conterá o *firmware* a ser iniciado quando dispositivo for energizado, e outra partição na qual o *firmware* candidato é armazenado durante o processo de atualização. Dessa forma, se o processo de atualização ocorrer com sucesso, então o *bootloader* faz o chaveamento de partições, indicando que a partição de atualização será agora a partição de execução. Essa abordagem tem como vantagem, em relação a abordagem com partição única, a possibilidade de que o dispositivo nunca ficará em um estado de exceção que demande uma ação do usuário. Ou seja, é possível iniciar o *firmware* antigo, mesmo que o processo de atualização tenha sido finalizado sem sucesso.



(a) Organização com SE. (b) Organização com TEE.

Figura 1. Organizações de *hardware* do dispositivo.

3.2. Mecanismos para garantir e integridade e autenticidade do *firmware*

O *bootloader* precisa garantir que o *firmware* instalado na partição de execução esteja íntegro antes que possa executá-lo, e na presente proposta isso é alcançado utilizando resumos criptográficos (*hash*). Ao receber um *firmware* candidato e garantir que o mesmo está íntegro, armazena-se o *hash* desse *firmware* no ambiente seguro. A cada processo de inicialização o *bootloader* deve calcular o *hash* do *firmware* presente na partição de execução e verificar a correspondência deste com o *hash* mantido no ambiente seguro.

A autenticidade do *firmware*, ou seja, a determinação que o mesmo foi gerado por uma parte autorizada (*e.g.* fabricante do dispositivo) é obtida na presente proposta por meio da criptografia de chave pública. Neste caso, a chave pública do fabricante deve ser inserida, em tempo de manufatura, no ambiente seguro do dispositivo. No processo de atualização, juntamente com o *firmware* candidato, deve-se encaminhar a assinatura digital gerada sobre o mesmo com a chave privada, par daquela mantida no dispositivo. A estrutura do binário a ser enviado no processo de atualização é apresentada na Subseção 3.3.

Uma vez que foi verificada a integridade e autenticidade do *firmware* candidato e o mesmo foi gravado na partição de execução, o processo de inicialização não precisará mais verificar a autenticidade do *firmware* a cada vez que o dispositivo for energizado.

Em alguns cenários, pode ser desejada a garantia de que somente novas versões de um *firmware* possam ser instaladas no dispositivo. Ou seja, não é desejado que o usuário possa instalar uma versão antiga, mesmo que essa tenha sido emitida e esteja assinada com a chave privada do fabricante do dispositivo. Nesta proposta está previsto o envio de um valor, juntamente com o binário do *firmware*, para representar a versão do *firmware*. As versões de cada *firmware* candidato devem ser crescentes e esta informação deve ser utilizada pelo *bootloader* no momento da atualização do dispositivo, impedindo que o *firmware* candidato seja instalado caso sua versão seja inferior ao *firmware* já armazenado no dispositivo. A versão do *firmware* candidato deve ser assinada pelo fabricante do dispositivo em conjunto com o binário do *firmware*. Dessa forma, o *bootloader* poderá ter certeza que a assinatura que vai ser verificada tem relação exclusivamente com o *firmware* candidato e o valor da versão que foram recebidos.

A presente proposta não restringe quais algoritmos de resumo criptográfico e de chave pública devem ser usados, ficando a critério do fabricante do dispositivo e de acordo com as capacidades de cada microcontrolador, tendo em vista que alguns microcontroladores possuem aceleradores criptográficos para algoritmos específicos.

3.3. Estrutura do arquivo binário com o *firmware* candidato

O *bootloader* precisa receber, além do *firmware* candidato, a versão deste *firmware* e a assinatura digital realizada sobre ambos. Ou seja, tem-se aqui três informações distintas e que poderiam ser enviadas em três transmissões sequenciais, e assim a necessidade da definição de um protocolo o qual indicaria a ordem das informações que seriam transmitidas e o que deveria ser feito caso não fosse recebida a informação esperada. Nesta proposta, optou-se pela simplicidade do processo de transmissão, e assim foi criado uma estrutura para que em um único binário estivessem a assinatura digital, a versão do *firmware* e o próprio *firmware* candidato. Ao *bootloader* cabe fazer a separação desse binário para conseguir obter cada informação.

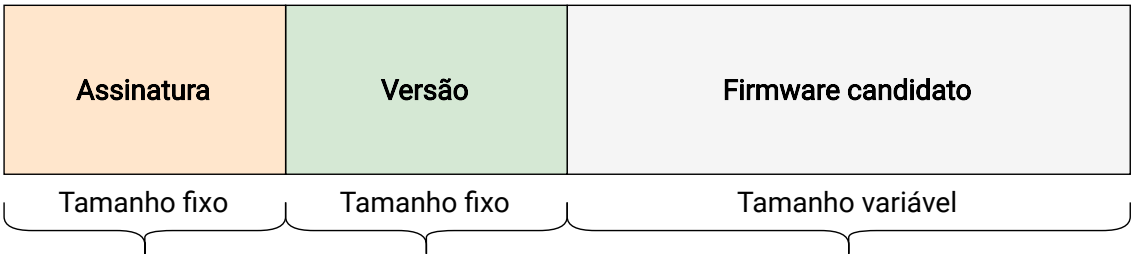


Figura 2. Estrutura do binário proposto.

Na Figura 2 é apresentada a estrutura de binário proposta neste trabalho. A versão do *firmware* e a assinatura digital possuem tamanhos fixos, sendo a última em função do algoritmo de assinatura digital escolhido. O tamanho fixo dessas informações permite ao *bootloader* separar facilmente o binário recebido e obter as informações necessárias para os processos de atualização e inicialização, enquanto que os últimos *bytes* do binário são referentes ao *firmware* candidato.

3.4. Organização e funcionalidades do *bootloader*

Na Figura 3 é apresentada uma das contribuições deste trabalho, um diagrama de classes conceitual UML [OMG 2017] para guiar fabricantes de dispositivos na implementação de seus *bootloader*. Ciente que a implementação do *bootloader* é dependente das ferramentas e da arquitetura de cada microcontrolador, todas as classes propostas são abstratas e assim devem ser especializadas e implementadas pelo fabricante do dispositivo. Um diagrama de classes UML na perspectiva de implementação, bem como a codificação dessas classes abstratas na linguagem C++ estão disponíveis no GitHub[1].

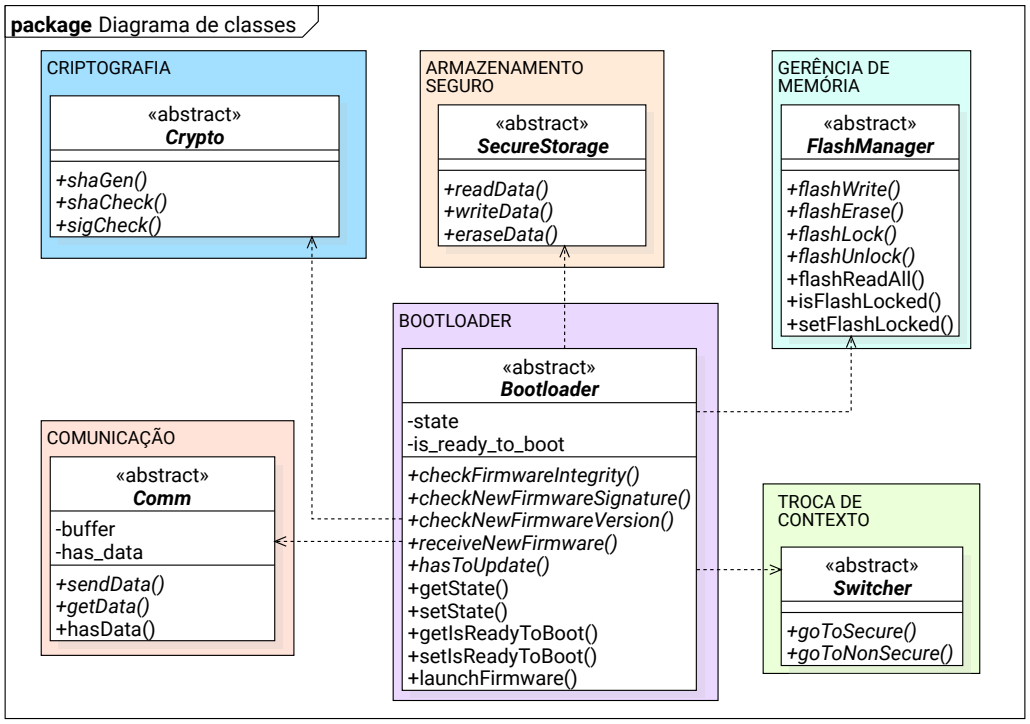


Figura 3. Diagrama de classes UML do *bootloader* proposto.

A classe `Bootloader` é responsável por implementar todo o comportamento do *bootloader* e, para melhor organização, depende das demais classes para executar as operações de criptografia (gerar e verificar assinaturas e resumos), armazenamento seguro (ler e escrever no ambiente seguro), gestão de memória (ler e escrever nas partições), comunicação (enviar e receber dados de acordo com a tecnologia usada, *e.g.* USB) e troca de contexto.

A classe para troca de contexto (`Switcher`) tem por objetivo permitir encerrar a execução do *bootloader* e colocar o *firmware* do dispositivo em execução, podendo também ser utilizada para a operação inversa. Assim, o *firmware* do dispositivo também deve apresentar uma implementação para esta classe. Apenas dois métodos são propostos nesta classe: um para a troca de contexto entre o *bootloader* e o *firmware* e outro para que o *firmware* avise ao *bootloader* que o processo de atualização de *firmware* deve ser iniciado.

[1] https://github.com/ifsc-saojose/secure-firmware-update/tree/master/artigo-wticg/classes-abstratas

O *bootloader* proposto foi modelado a partir de uma máquina de estados finita (i.e. *Finite State Machine* – FSM), apresentada na Figura 4, na qual três estados são definidos: verificação, confirmação e atualização. O estado atualização, por sua vez, é dividido em quatro subestados: preparação, recepção, validação e instalação.

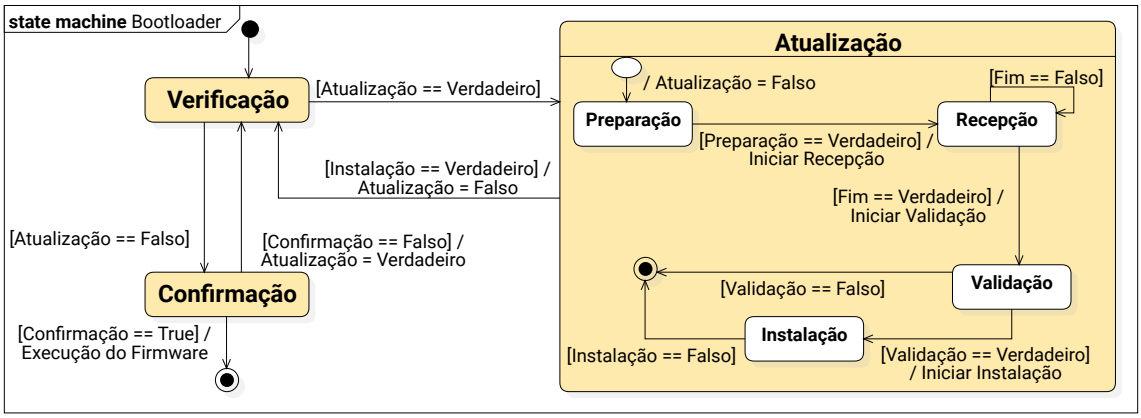


Figura 4. Máquina de estados finitos UML para o *bootloader*.

No estado **verificação** o *bootloader* verifica se deve iniciar o processo de atualização de *firmware*. Assim, consulta a variável `atualização` que está armazenada no ambiente seguro. Se estiver como verdadeiro, então irá para o estado **atualização**. Se estiver como falso, então irá para o estado **confirmação** onde é feita a verificação de integridade do *firmware* presente na partição de execução. Se estiver íntegro, é feito o chaveamento de contexto, colocando o *firmware* em execução. Se o *firmware* não estiver íntegro, então o *bootloader* deve atribuir o valor verdadeiro na variável `atualização`, indicando assim a necessidade de atualização, e retornar ao estado inicial (**verificação**).

O *firmware* em execução deve possuir uma funcionalidade (a ser invocada pelo usuário) que permita atribuir o valor verdadeiro na variável `atualização` para indicar ao *bootloader* que este deverá ir para o estado de **atualização** assim que for feito o chaveamento de contexto, ou seja, quando o *firmware* passar a execução para o *bootloader*. Somente o *bootloader* será capaz de alterar para falso o conteúdo da variável `atualização` e este sempre o fará a partir do momento que entrar no estado **atualização**.

No subestado **preparação**, o *bootloader* atribui o valor falso na variável `atualização` e prepara o dispositivo para o recebimento do binário contendo o *firmware* candidato. A principal ação nesse estado é a limpeza da região de memória que irá armazenar o *firmware* candidato contido no binário a ser recebido, bem como a limpeza das regiões de memória no ambiente de armazenamento seguro que irão armazenar a assinatura e versão do *firmware* candidato, que também estão contidos no binário. Uma vez que a limpeza foi finalizada, então o *bootloader* passa para o estado **recepção** onde o binário é recebido e armazenado na participação de atualização.

Assim que encerrar o recebimento e armazenamento do binário, o *bootloader* passa para o estado **validação**, onde acontecem as operações para verificar a integridade e autenticidade do *firmware* recebido, bem como se o número da versão recebido é superior ao atual número armazenado no ambiente seguro. Caso qualquer uma dessas verificações resulte em insucesso, então o *bootloader* aborta o processo de atualização e deve indicar ao usuário que a atualização não foi bem-sucedida, além de permanecer neste estado

até que o usuário reinicie o dispositivo. Por outro lado, se as validações resultarem em sucesso, então o *bootloader* passa para o estado **instalação**.

No estado **instalação**, o *bootloader* define que o *firmware* recém recebido é a partir de agora o *firmware* que deve ser executado no dispositivo, realizando o chaveamento entre as partições do dispositivo. Por fim, é gerado o resumo criptográfico sobre o binário do *firmware* recebido e persistido no ambiente seguro. Caso algum dos processos deste estado resulte em falha, o *bootloader* deve abortar o processo de atualização e indicar ao usuário que a atualização não foi bem-sucedida, devendo ficar neste estado até que o usuário reinicie o dispositivo. Por fim, caso o processo de geração do resumo criptográfico e a instalação do *firmware* tenha sido bem-sucedida, o *bootloader* deve retornar ao estado **verificação**.

4. Implementação da solução proposta

A fim de verificar se a organização de classes e o comportamento do *bootloader* propostos são factíveis, foi feita uma implementação utilizando o microcontrolador STM32L562QE, da fabricante *STMicroelectronics*, com o *kit* de desenvolvimento *STM32L562-DK Discovery*.

Este microcontrolador é composto pelo o *TrustZone*, a implementação de um TEE da empresa *ARM*, o qual foi utilizado como ambiente de armazenamento seguro para as informações essenciais aos processos de atualização e inicialização do dispositivo. Com o *TrustZone* é possível fazer uma divisão da memória de programa do microcontrolador em área segura e área não segura. Um código sendo executado na área não segura é totalmente isolado e não possui acesso à área segura, a não ser pela possibilidade de chamadas feitas para funções armazenadas em uma região chamada *Non-Secure Callable* (NSC). 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. Essas chamadas precisam ser pré-definidas e acordadas em tempo de compilação. Por outro lado, um código executado na área segura tem acesso a toda memória de programa do microcontrolador [STMicroelectronics 2020c].

O *TrustZone* provê um mecanismo de segurança chamado *Write Protection* (WRP), que permite que uma área específica da memória de programa seja imutável. Neste caso foi feito uso do WRP nas regiões que armazenam o *bootloader* e a chave pública do emissor do *firmware*. Com o *TrustZone* também foi possível definir um ponto único de início de execução, sendo este o endereço de memória o qual o *bootloader* foi armazenado.

4.1. Implementação do *bootloader*

Nesta seção é apresentado como cada classe abstrata presente na Figura 3 foi implementada. O microcontrolador usado conta com uma interface USB e essa foi usada como interface de comunicação para recebimento do binário contendo o *firmware* candidato. Foi implementada a classe USB *Communications Device Class* (CDC), subclasse *Abstract Control Model* (ACM) [USB-IF 2010], pois esta permite estabelecer uma comunicação serial entre os pares comunicantes de forma simples e o fabricante do microcontrolador disponibiliza uma biblioteca de fácil compreensão e utilização.

O microcontrolador escolhido possui acelerador criptográfico em *hardware* que implementa criptografia *Rivest-Shamir-Adleman* (RSA) e criptografia de curvas elípticas. Optou-se por usar chaves criptográficas RSA com tamanho de 2.048 *bits* e o SHA-256 [NIST 2015] como algoritmo para resumos criptográficos (*hash*), uma vez que as bibliotecas criptográficas do fabricante forneciam suporte a estes.

O fabricante do microcontrolador oferece biblioteca para manipulação da memória de programa do mesmo. Esta biblioteca foi utilizada para a especialização da classe de gerência de memória desta proposta e aqui foi escolhida a abordagem com duas partições: partição de execução – que contém o *firmware* que deverá ser executado após o término da execução do *bootloader*; partição temporária – para armazenar o *firmware* candidato durante o processo de atualização.

A utilização do TEE como ambiente de armazenamento seguro permitiu utilizar a classe de gerência de memória como especialização da classe de armazenamento seguro (veja Figura 3). Neste caso, o acesso à área de memória segura pelo *bootloader* ocorre da mesma maneira que o acesso à área não segura, não sendo necessário prover uma implementação específica para trabalhar com a memória segura. Se fosse feito uso de uma organização com elemento seguro (SE) e não TEE, então seria necessário prover uma implementação específica da classe de armazenamento seguro que interagisse com o *hardware* externo.

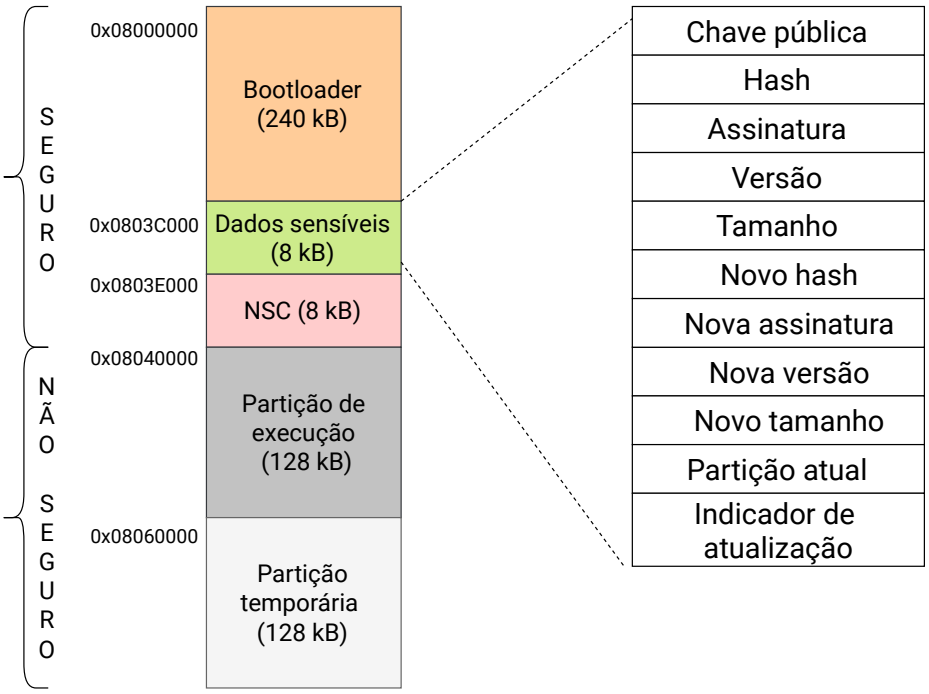


Figura 5. Segmentação da memória de programa do microcontrolador.

Na Figura 5 é apresentada a segmentação da memória de programa do microcontrolador para armazenar o código do *bootloader*, dados sensíveis, as rotinas *Non-Secure Callable* (NSC), bem como as áreas destinadas para as partições de execução e temporária. Na figura é possível observar em qual ambiente de execução, seguro e não seguro, cada informação é armazenada.

A implementação da classe *switcher* consistiu em uma função que faz o salto da memória de programa segura (*bootloader*) para a memória de programa não segura, colocando o *firmware* em execução como recomendado pelo fabricante do microcontrolador [STMicroelectronics 2020a].

Por fim, a classe *bootloader* foi especializada a partir das classes já discutidas, de modo que alguns métodos desta classe são dependentes das demais especializações, por exemplo, o método que verifica a integridade do *firmware* atual, que depende da classe de criptografia.

4.2. Implementação do *firmware*

A fim de garantir que o *firmware* possa avisar ao *bootloader* sobre uma tentativa de atualização, a implementação do firmware especializou a classe abstrata de troca de contexto (veja Figura 3). A especialização desta classe se deu a partir de uma chamada NSC, fazendo assim uso da *Application Programming Interface* (API) compartilhada entre as áreas segura (*bootloader*) e não segura (*firmware*. A implementação da chamada NSC foi feita na área segura e na área não segura foi feita a invocação dessa API compartilhada.

5. Experimentos e resultados

Foram realizados quatro experimentos para verificar se o modelo e organização propostos, bem como a implementação no microcontrolador STM32L562QE, gerariam os resultados esperados. Para verificar se a atualização de *firmware* acontecia com sucesso, foram desenvolvidas duas versões de *firmware*, `versão 01` e `versão 02`. Cada *firmware* tinha um comportamento distinto de forma que fosse fácil identificar qual *firmware* estava em execução após uma tentativa de atualização. A `versão 01` faz com que o *led* do *kit* de desenvolvimento pisque a cada 100 milissegundos, enquanto que a `versão 02` faz com que o *led* pisque a cada 1.000 milissegundos. Os experimentos conduzidos foram:

1. Transmissão completa e íntegra de todos os dados necessários para que a atualização ocorresse com sucesso;
2. Tentativa de atualizar um *firmware* com versão inferior à versão instalada no dispositivo;
3. Tentativa de atualizar com uma assinatura digital inválida sobre o binário;
4. Tentativa de atualizar com a transmissão sendo interrompida antes da transferência completa do binário.

Para o experimentos 1, 2 e 3 foi feito o seguinte procedimento: com o dispositivo energizado e com o *firmware* em execução, pressionou-se o botão do kit de desenvolvimento para iniciar o processo de atualização. Uma vez que o *bootloader* entrou em execução, então foi iniciada a transmissão do binário (veja Figura 2) pela interface USB do kit de desenvolvimento e aguardou-se que essa fosse executada por completo. Para o experimento 4, o dispositivo era reiniciado antes que a transferência completa do binário pudesse ser concluída com sucesso. Na Tabela 1 são apresentados os resultados esperados e obtidos com cada experimento.

No experimento 1, o dispositivo estava com o *firmware* `versão 01` instalado e após o recebimento do binário o *bootloader* do dispositivo validou a versão e assinatura e instalou o *firmware* com sucesso. Por fim, o *bootloader* encerrou sua execução, validando a integridade do *firmware* e colocando o *firmware* `versão 02` em execução, finalizando o experimento com sucesso.

Tabela 1. Resumo dos experimentos realizados

Experimento	Assinatura	Versão	Transmissão	Resultado	
				Esperado	Obtido
1	válida	superior	completa	sucesso	sucesso
2	válida	inferior	completa	insucesso	insucesso
3	inválida	superior	completo	insucesso	insucesso
4	válida	superior	incompleta	insucesso	insucesso

No experimento 2, o dispositivo estava com o *firmware* `versão 02` instalado e recebeu o binário com o *firmware* `versão 01`. Ao final da transmissão, o *bootloader* do dispositivo verificou que a versão recebida era inferior à versão do *firmware* já instalado e assim interrompeu a tentativa de atualização. O *bootloader* indicou ao usuário que o processo de atualização não foi bem-sucedido e, após a reinicialização do dispositivo, o *firmware* `versão 02` voltou a ser executado, conforme o esperado.

Para o experimento 3, o dispositivo estava com o *firmware* `versão 01` instalado e recebeu o binário com o *firmware* `versão 02`, porém contendo uma assinatura que não fora gerada com a versão e *firmware* candidato contidos no binário. O *bootloader*, ao constatar que a assinatura era inválida, interrompeu a tentativa de atualização, indicou ao usuário que o processo de atualização não foi bem-sucedido e, após a reinicialização do dispositivo, o *firmware* `versão 01` voltou a ser executado, conforme o esperado.

Por fim, no experimento 4, com o *firmware* `versão 01` sendo executado no dispositivo, o processo de atualização foi iniciado, sendo transmitido ao dispositivo binário com o *firmware* `versão 02` e com assinatura gerada pela chave par da chave pública contida no dispositivo. Antes que o binário fosse recebido por completo pelo dispositivo, foi retirada a alimentação do dispositivo e quando este voltou a ser energizado, o *bootloader* colocou o *firmware* `versão 01` em execução e ignorou os dados corrompidos na partição temporária, como o esperado.

6. Conclusões

Este trabalho apresentou uma proposta de atualização de *firmware* voltada para dispositivos utilizados em sistemas embarcados, de modo que essa atualização permita que apenas *firmware* autênticos possam ser instalados, além de garantir que o dispositivo não fique inutilizável após o processo de atualização caso a atualização não seja finalizada com sucesso. Também foi proposta uma organização de classes para guiar a implementação do *bootloader* em microcontroladores com diferentes arquiteturas, sendo necessário apenas especializar as classes que interagem com o *hardware* de cada microcontrolador.

Foi feita uma implementação da organização proposta no microcontrolador STM32L562QE que possui um TEE, sendo este o ambiente seguro no qual o *bootloader* e as informações sensíveis aos processos de atualização e inicialização do dispositivo foram armazenados. Na implementação, optou-se pela abordagem com duas partições (execução e temporária) para permitir que o dispositivo possa se recuperar caso alguma tentativa de atualização do *firmware* do dispositivo termine sem sucesso.

Quatro experimentos foram conduzidos a fim de validar o modelo proposto, bem

como validar a implementação específica para o microcontrolador STM32L562QE. Os experimentos permitiram demonstrar que a solução de atualização de *firmware* e *bootloader* propostos, bem como a respectiva implementação, permitem que a atualização do *firmware* do dispositivo ocorra somente quando um *firmware* autêntico é transferido ao mesmo, além de não inutilizá-lo quando uma tentativa de atualização não é finalizada com sucesso.

Como proposta de trabalhos futuros, sugere-se implementar a solução apresentada neste trabalho utilizando um microcontrolador não-seguro acoplado com um elemento seguro, de modo a validar a solução de atualização de *firmware* para esta organização de *hardware*. Sugere-se também implementar a solução utilizando apenas um partição de armazenamento de *firmware*.

Referências

[Beningo 2015] Beningo, J. (2015). Bootloader design for microcontrollers in embedded systems.

[Bouazzouni et al. 2018] Bouazzouni, M. A., Conchon, E., and Peyrard, F. (2018). Trusted mobile computing: An overview of existing solutions. *Future Generation Computer Systems*, 80:596 – 612.

[Dhobi et al. 2019] Dhobi, R., Gajjar, S., Parmar, D., and Vaghela, T. (2019). Secure firmware update over the air using trustzone. In *2019 Innovations in Power and Advanced Computing Technologies (i-PACT)*, volume 1, pages 1–4.

[Jain et al. 2016] Jain, N., Mali, S. G., and Kulkarni, S. (2016). Infield firmware update: Challenges and solutions. In *2016 International Conference on Communication and Signal Processing (ICCSP)*, pages 1232–1236.

[Landwehr 2001] Landwehr, C. (2001). Computer security. *International Journal of Information Security*, 1:3–13.

[Nikolov 2018] Nikolov, N. (2018). Research firmware update over the air from the cloud. In *2018 IEEE XXVII International Scientific Conference Electronics - ET*, pages 1–4.

[NIST 2015] NIST (2015). *Secure Hash Standards*. National Institute of Standards and Technology.

[OMG 2017] OMG (2017). Omg unified modeling language (omg uml).

[Russel and Gangemi 1991] Russel, D. and Gangemi, G. T. (1991). *Computer Security Basics*. O'Reilly & Associates.

[STMicroelectronics 2020a] STMicroelectronics (2020a). Getting started with projects base on the stm32l5 series in stm32cubeide.

[STMicroelectronics 2020b] STMicroelectronics (2020b). Overview of secure boot and secure firmware update solution on arm trustzone stm32l5 series microcontrollers.

[STMicroelectronics 2020c] STMicroelectronics (2020c). Reference manual - stm32l552xx and stm32l562xx advanced arm-based 32-bit mcus.

[USB-IF 2010] USB-IF (2010). *Universal Serial Bus Class Definitions for Communications Devices*. USB-IF.