LEONARDO RODRIGUES MURAD CASSIANO
LUCAS DE MENEZES CAVALCANTE
MARCIO SUGUIYAMA DE ABREU

SEGURANÇA DE SISTEMAS A NÍVEL DE FIRMWARE: APLICAÇÃO COM TIME-BASED ONE TIME PASSWORD ALIADO AO SECURITY PROTOCOL AND DATA MODEL

São Paulo
2022

LEONARDO RODRIGUES MURAD CASSIANO
LUCAS DE MENEZES CAVALCANTE
MARCIO SUGUIYAMA DE ABREU

SEGURANÇA DE SISTEMAS A NÍVEL DE FIRMWARE: APLICAÇÃO COM TIME-BASED ONE TIME PASSWORD ALIADO AO SECURITY PROTOCOL AND DATA MODEL

Trabalho apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro de Computação.

São Paulo
2022

LEONARDO RODRIGUES MURAD CASSIANO
LUCAS DE MENEZES CAVALCANTE
MARCIO SUGUIYAMA DE ABREU

SEGURANÇA DE SISTEMAS A NÍVEL DE FIRMWARE: APLICAÇÃO COM TIME-BASED ONE TIME PASSWORD ALIADO AO SECURITY PROTOCOL AND DATA MODEL

Trabalho apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro de Computação.

Orientador:
Prof. Dr. Renan Alves

Co-orientador:
Prof. Dr. Marcos Simplicio
Prof. Dr. Bruno Albertini

São Paulo
2022

Prof. Dr. Renan Cerqueira Afonso Alves

AGRADECIMENTOS

Ao nosso orientador, Prof. Dr. Renan Cerqueira Afonso Alves. Aos nossos co-orientadores, Prof. Dr. Marcos Antonio Simplicio Junior e Prof. Dr. Bruno de Carvalho Albertini. Aos nossos professores, que nos forneceram a base do conhecimento para que chegássemos até aqui. Aos nossos amigos e familiares, que nos apoiaram e nos motivaram durante todo o decorrer do projeto.

RESUMO

Atualmente, sistemas computacionais são construídos utilizando componentes de múltiplos vendedores. Nesse cenário, é fundamental prevenir ataques à *supply chain*, como manipulação de componentes através de modificação do firmware. Ainda, temos que ações maliciosas a nível de firmware são, de maneira geral, de difícil detecção quando comparadas a ações semelhantes a nível da aplicação ou do sistema operacional. Dessa forma, para segurança a nível de firmware, protocolos como o Security Protocol and Data Model (SPDM) têm se mostrado eficientes, sendo utilizados para a autenticação de identidade de firmware de servidores. Em outro ramo da segurança, voltado à autenticação a partir de um intervalo de tempo, nota-se a forte presença do Time-based One Time Password (TOTP), utilizado em sistemas de autenticação em dois fatores. Com o intuito de contribuir para o desenvolvimento e para a aplicação do SPDM na indústria, o projeto busca integrar as duas tecnologias em uma única solução de segurança, por meio do desenvolvimento de um sistema composto por um driver USB e um dispositivo emulado pelo QEMU, que realizam uma troca periódica de mensagens SPDM e TOTP de tal forma que qualquer inconsistência no firmware do sistema host ou falha na comunicação SPDM inativa o sistema como um todo, garantindo, assim, a segurança do sistema a nível de firmware. Foi desenvolvido também um dispositivo físico que substitui o dispositivo emulado. O resultado final deste trabalho é um conjunto capaz de proteger o sistema a nível de firmware de maneira periódica e constante, com um ganho de desempenho até 99% comparado ao uso apenas do SPDM.

Palavras-Chave – Segurança de Computadores. C (Linguagem de Programação). Linux. Engenharia de Computação. Hardware.

ABSTRACT

In modern times, computer systems are built using parts from multiple vendors. In this scenario, it is critical to prevent supply chain attacks, such as component manipulation through firmware modification. Furthermore, malicious actions at the firmware level are often difficult to detect when compared to similar actions at the application or operating system level. Thus, for firmware level security, protocols such as the Security Protocol and Data Model (SPDM) have proven to be efficient, and have been used for server firmware identity authentication. In another branch of security, focused on time-based authentication, the Time-based One Time Password (TOTP) proves itself as the leading technology, used in two-factor authentication systems. In order to contribute to the development and application of SPDM in the industry, this project seeks to integrate the two technologies into a single security solution by developing a system composed of a USB driver and a device emulated by QEMU, which perform a periodic exchange of SPDM and TOTP messages in such a way that any inconsistency in the host system firmware or failure in SPDM communication inactivates the system as a whole, thus ensuring system security at the firmware level. A physical device that replaces the emulated device was also developed. The result is an assembly capable of securing the system at the firmware level periodically and constantly, with a performance improvement of up to 99% compared to using only SPDM.

Keywords – Computer Security. C (Programming Language). Linux. Computer Engineering. Hardware.

LISTA DE FIGURAS

1 Tríade de requisitos de segurança. 16

2 Fluxo de mensagens do SPDM. 18

3 Fluxo de mensagens do SPDM com Chave Pré-Compartilhada. 19

4 Estrutura do HMAC. 21

5 Esquematização do processo de funcionamento do TOTP. 23

6 Esquematização do kernel como núcleo de um sistema operacional. 23

7 Esquematização do funcionamento do sistema. 34

8 Ilustração do fluxo de execução do sistema desenvolvido. 36

9 Caminho para ativação de opção do Buildroot para depuração do kernel. . 42

10 Caminho para ativação de opção do Buildroot para permitir comunicação serial. 45

11 Diagrama de Sequência do fluxo de geração de chave TOTP. 49

12 Arquitetura do sistema físico. 94

13 Organização do sistema físico. 95

14 Logs indicando a adição de um novo dispositivo USB. 97

15 Logs do início da execução do fluxo normal do sistema. 107

16 Logs da inicialização da comunicação SPDM. 107

17 Logs da remoção do dispositivo USB. 107

18 Logs da inicialização do sistema sem o dispositivo. 108

19 Logs da adição do dispositivo USB após *boot*. 108

20 Logs da inicialização do SPDM após adição do dispositivo USB. 109

21 Logs da dessincronização da máquina *host* e do dispositivo USB. 109

22 Logs das verificações periódicas com SPDM. 111

23 Teste de desempenho do driver desenvolvido por meio do htop. 111

24 Teste de desempenho do driver desenvolvido por meio do htop. 112

25 Teste de desempenho do driver desenvolvido por meio do htop. 112

26 Erros encontrados devido à diferença nas versões do kernel utilizado. 114

SUMÁRIO

1 Introdução **11**

1.1 Objetivos 12

1.2 Justificativa 12

1.3 Organização do Trabalho 13

2 Aspectos Conceituais **15**

2.1 Confidencialidade, Integridade, Autenticidade e Irretratabilidade 15

2.2 Security Protocol and Data Model (SPDM) 17

2.3 HMAC: Código de Autenticação de Mensagens com base em Hash 19

2.4 Time-based One-Time Password (TOTP) 22

2.5 Kernel do Linux 23

3 Especificação de Requisitos **25**

4 Tecnologias Utilizadas **29**

4.1 QEMU 29

4.2 Buildroot 29

4.3 libspdm 30

4.4 GDB 30

5 Projeto e Implementação **32**

5.1 Metodologia do Trabalho 32

5.2 Visão geral do sistema 33

5.3 Configuração inicial do ambiente 36

5.4 Adição de driver ao Buildroot como módulo 38

5.4.1 Adição de módulo externo ao Buildroot 39

5.4.2 Adição de módulo interno ao Buildroot 40

5.5 Configuração de ambiente de depuração com GDB 41

5.6 Desenvolvimento do dispositivo USB emulado pelo QEMU 43

5.6.1 Entendimento de dispositivo USB básico 44

5.6.2 Criação de um novo dispositivo USB emulado pelo QEMU 46

5.6.3 Implementação do algoritmo TOTP no dispositivo 48

5.6.3.1 Fluxo de geração de novas chaves TOTP 51

5.6.3.2 Fluxo de envio de código TOTP 53

5.6.4 Implementação do protocolo SPDM no dispositivo 54

5.6.4.1 Configurações iniciais do contexto SPDM 54

5.6.4.2 Recepção de mensagens SPDM 59

5.6.4.3 Transmissão de mensagens SPDM 63

5.7 Desenvolvimento do driver do dispositivo 69

5.7.1 Comunicação entre driver e dispositivo USB 70

5.7.2 Criação de rotina periódica constante no driver 73

5.7.3 Desenvolvimento de método de desligamento do sistema em caso de falha 74

5.7.4 Implementação do protocolo SPDM no driver 75

5.7.4.1 Operação de envio de mensagens SPDM por USB 77

5.7.4.2 Operação de recebimento de mensagens SPDM por USB . 78

5.7.4.3 Inicialização da comunicação SPDM 81

5.7.5 Implementação do algoritmo TOTP no driver 84

5.7.5.1 Estrutura geral dos fluxos TOTP 84

5.7.5.2 Fluxo de geração de chave secreta para o TOTP 85

5.7.5.3 Fluxo de verificação de consistência do código TOTP . . . 88

5.8 Implementação do dispositivo físico 92

5.8.1 Uso da placa Intel Galileo e da tecnologia USB OTG 94
5.8.2 Configuração de ambiente para uso do dispositivo físico 96
5.8.3 Criação de imagem com uso de Buildroot e Yocto aliado a Docker . 97
5.8.4 Compilação cruzada da libspdm para a versão do Linux customizada 100
5.8.5 Implementação do código em C do dispositivo SPDM 101
5.9 Finalização da implementação 104

6 Testes e Avaliação **106**

6.1 Testes de Segurança do dispositivo emulado 106
6.2 Testes de desempenho do dispositivo emulado 109
6.3 Testes e avaliação do dispositivo físico 112

7 Conclusão **115**

7.1 Contribuições 115
7.2 Trabalhos Futuros 116

Referências **117**

Apêndice A - Exemplo de script de *daemon* de um módulo externo **120**

1 INTRODUÇÃO

Sistemas computacionais modernos podem ser construídos para otimizar diferentes métricas, incluindo o custo, tamanho, e desempenho. Para este fim, o processo de construção envolve normalmente peças de múltiplos vendedores especializados. Em tal cenário, com muitos grupos envolvidos, é fundamental prevenir explorações que envolvam ataques à cadeia de suprimentos (*supply chain*) de dispositivos computacionais, incluindo a manipulação de componentes através da modificação do firmware [1]. Afinal, como esse tipo de elemento monitora e controla todas as operações do componente associado, sua modificação permite ataques contra qualquer outro componente que dependa do seu correto funcionamento [2]. Ao mesmo tempo, ações maliciosas realizadas no nível do firmware são geralmente muito mais difíceis de detectar do que ações semelhantes no nível da aplicação ou do sistema operacional, uma vez que são invisíveis a softwares de proteção comum, como firewalls e antivírus [3].

Com isso em mente, é de suma importância que haja uma proteção contra tais ataques de modificação de firmware. Entre as principais iniciativas que abordam esta faceta da segurança de um sistema, destaca-se o Protocolo de Segurança e Modelo de Dados (SPDM, sigla em inglês para Security Protocol and Data Model). Em essência, a especificação SPDM define um conjunto de mecanismos e formatos para autenticação de hardware e firmware desenvolvido como um padrão industrial aberto, definindo mensagens, objetos de dados e sequências para a realização de trocas de mensagens entre dispositivos de maneira segura.

Outro mecanismo que pode ser usado para aumentar a segurança de sistemas é o TOTP (Time-Based One-Time Password), que é um algoritmo gerador de senhas de uso único. Esta ferramenta é uma expansão do algoritmo de senhas únicas HOTP, porém gerando senhas com base no tempo atual, utilizando este como métrica para garantia de sua singularidade [4]. Assim, dois sistemas que partilham dos mesmos parâmetros para o algoritmo (método de hash e chave secreta) podem computar o valor do TOTP baseado nestes parâmetros e no tempo atual, e o autenticador confirma se o valor fornecido pelo

dispositivo que fez a requisição corresponde ao gerado localmente. Dessa forma, o uso de TOTP garante que a autenticação seja realizada com sucesso apenas mediante o uso do mesmo conjunto de chaves secretas, com uma requisição feita em tempo real.

Este algoritmo tem sido utilizado largamente nos últimos anos, com a recente demanda por sistemas mais seguros em todos os âmbitos de sistemas computacionais. Em particular, nota-se sua presença em sistemas de autenticação de dois fatores, conhecido como 2FA, um sistema popularmente utilizado em logins de web serviços em geral, que aumentam a segurança da conta de usuários. Seu funcionamento baseia-se na requisição de um código de 6 a 8 dígitos, que é enviado por meio de SMS, e-mail ou método similar. Assim, tem-se uma garantia altíssima de segurança, salvo possíveis ataques do tipo MITM (Man-In-The-Middle, que nesse caso seria equivalente à interceptação do código gerado e enviado ao usuário).

1.1 Objetivos

O projeto tem como objetivo integrar o SPDM e o TOTP a uma aplicação real, gerando uma solução de segurança mais robusta a nível de firmware contra quaisquer modificações de componentes do sistema.

Dessa forma, esse trabalho se insere no contexto de Segurança da Informação, propondo uma discussão da aplicabilidade em um protótipo de um produto real do protocolo SPDM aliado a TOTP, duas tecnologias que têm ganhado relevância na indústria recentemente, visando mitigar ataques a nível de hardware.

1.2 Justificativa

O SPDM como solução de segurança a nível de *firmware* ainda está em sua infância, tendo como fronteira o desenvolvimento de bibliotecas como a libspdm e integração com sistemas de propósito genérico, como o Linux [5]. Sendo assim, a integração do SPDM com outras tecnologias de segurança é algo inédito, e que tem potencial de melhorar ainda mais a segurança do sistema adicionando uma nova camada de defesa.

A utilização de TOTP como segurança adicional é atraente por adicionar uma camada temporal à segurança do sistema, possibilitando verificações periódicas entre dois dispositivos. Com isso, dificulta-se a execução de ataques como de *spoofing*, já que o sistema rejeitaria quaisquer mensagens geradas com período ou chave incorreta.

Adicionalmente, a encriptação proporcionada pelo SPDM impede também a execução de ataques do tipo Man-In-The-Middle, que poderiam ser realizados por meio da interceptação de códigos TOTP enviados do dispositivo à máquina *host*. A confidencialidade e a integridade garantida pelo SPDM neste quesito garante que um possível atacante não poderá visualizar as informações trafegadas na rede ou alterá-las sem que o sistema saiba.

Em questão de desempenho, a adição do TOTP a um sistema protegido por SPDM é leve, não interferindo com o funcionamento do sistema ou do SPDM.

1.3 Organização do Trabalho

Este documento se propõe como um estudo sobre ataques à *supply chain* a nível de firmware, não apenas colocando o SPDM como uma das melhores maneiras de preveni-los, mas também considerando a efetividade de uma aplicação deste protocolo unificado ao TOTP, incluindo toda a documentação do processo tomado para a criação da ferramenta em questão, garantindo a possibilidade de reprodução e continuação do projeto a partir dos resultados aqui expressos.

Assim, o Capítulo 2 contém os aspectos conceituais relevantes ao entendimento e desenvolvimento do projeto, contendo breves explicações sobre os paradigmas de segurança que devem ser considerados na criação de uma ferramenta voltada para esta faceta de sistemas computacionais. Em seguida, são apresentados o protocolo SPDM e o algoritmo TOTP, fundamentais para a proteção do firmware de sistemas, além da plataforma na qual esses sistemas serão integrados – o Kernel do sistema operacional Linux.

Então, no Capítulo 3, é documentada a metodologia de trabalho, especificando-se as diferentes etapas tomadas no desenvolvimento do projeto, enquanto que o Capítulo 4 concentra os requisitos funcionais e não-funcionais definidos para o sistema.

A partir do Capítulo 5, apresenta-se o desenvolvimento do trabalho como um todo, permitindo a visualização dos produtos finais obtidos com o projeto. Inicialmente, aborda-se as principais tecnologias utilizadas durante o desenvolvimento, servindo como suporte à integração entre as tecnologias e as plataformas necessárias ao projeto ou como ferramenta para testes, emulação e simulação.

Em seguida, no Capítulo 6, é descrita a implementação como um todo, tendo-se uma documentação de todos os processos realizados e decisões tomadas, permitindo assim a reprodução dos resultados obtidos. Já no Capítulo 7, apresenta-se os testes realizados quanto à segurança e ao desempenho dos sistemas que utilizam a ferramenta desenvolvida,

avaliando-a como possível ferramenta no mercado.

Por fim, o Capítulo 8 traz conclusões finais do projeto, apresentando e justificando os resultados atingidos durante o desenvolvimento do trabalho. Nesse capítulo, também são descritas as contribuições geradas pelo trabalho empenhado, além de perspectivas de continuidade para o projeto.

2 ASPECTOS CONCEITUAIS

Neste capítulo, são apresentados conceitos fundamentais para o entendimento do desenvolvimento do projeto. Inicialmente, aborda-se conceitos relevantes acerca da segurança de sistemas, por meio da descrição dos conceitos de Confidencialidade, Integridade, Autenticidade e Irretratabilidade. Em seguida, as duas bases de segurança do projeto, o protocolo SPDM e o algoritmo TOTP, são descritos e explicados de maneira sucinta, apresentando seu funcionamento, junto com uma descrição do algoritmo HMAC, fundamental para o entendimento do TOTP. Após isso, também é descrita a plataforma do Kernel do sistema operacional Linux, na qual as tecnologias serão implementadas, focando-se em como sua relevância para o projeto é dada.

2.1 Confidencialidade, Integridade, Autenticidade e Irretratabilidade

A segurança de computadores pode ser definida como "a proteção proporcionada a um sistema de informação de maneira que se atinja o objetivo de preservar a integridade, disponibilidade e confidencialidade dos recursos do sistema", como definido em [6].

Considerando o escopo deste projeto, é de suma importância citar e explicar os três grandes pilares da segurança de informação, conhecidos pela sigla CIA – Confidencialidade, Integridade e Disponibilidade (Availability, em inglês), que pode ser vista na Figura 1.

Como citado por Samonas et. al. em [8], a tríade oferece aos desenvolvedores no âmbito da segurança de informação uma maneira simples de entender e resolver problemas relacionados à área, servindo como guia para modelar e orientar as práticas com as quais os sistemas voltados à segurança são desenvolvidos há mais de 40 anos.

Vale ressaltar, no entanto, que esta tríade não é completa, com a literatura focando na expansão desta, adicionando e alterando elementos de tal maneira que os novos pa-

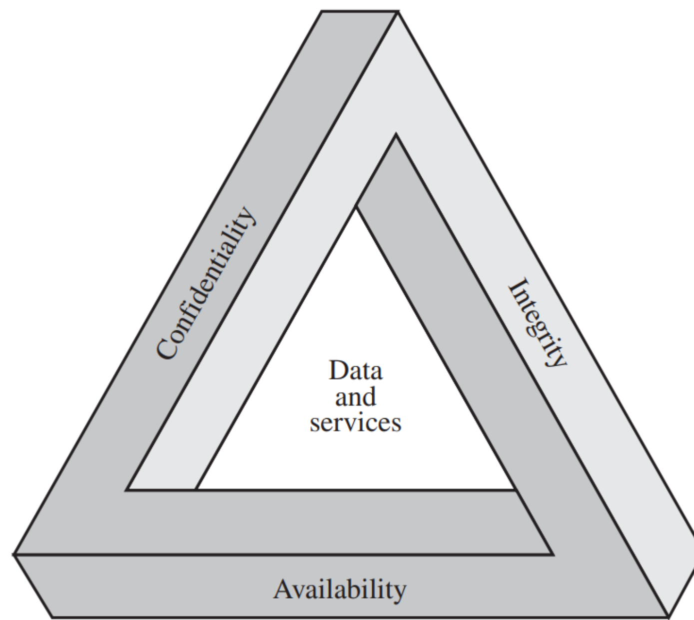


Figura 1: Tríade de requisitos de segurança.

Fonte: [7].

radigmas e aperfeiçoamentos no pensamento voltado à segurança possam ser inclusos na teoria básica da área.

Em particular, na solução proposta, nota-se uma necessidade de se cobrir não apenas os elementos de Confidencialidade e Integridade, presentes na tríade de segurança, além dos conceitos de Autenticidade e Irretratabilidade, que também se provam necessários para a segurança do dispositivo a ser desenvolvido neste trabalho.

Nesse contexto, Confidencialidade pode ser descrita como a preservação das restrições autorizadas de acesso e divulgação de informações [7]. Assim, como um exemplo aplicado à segurança de firmware, a garantia de confidencialidade implica na segurança de que acessos aos dados importantes do sistema, incluindo informações internas, segredos e canais de comunicação entre dispositivos ou componentes, sejam realizados unicamente por usuários confiáveis.

Em seguida, por Integridade, entende-se a proteção contra a modificação imprópria ou destruição de informações [7]. No campo deste projeto, uma violação de integridade se daria pela modificação dos dados internos do firmware de um sistema por um atacante, possibilitando que sejam alterados dados internos do sistema.

Então, como Autenticidade, tem-se a propriedade de ser confiável, podendo ser verificado. Também pode ser considerada como a garantia de confiança na validade de uma

transmissão ou mensagem [7]. Neste trabalho, é fundamental garantir-se a autenticidade, já que um dos principais objetivos do sistema a ser desenvolvido é justamente a garantia de que os dispositivos dialogando entre si por meio do SPDM não foram falsificados, por exemplo.

Por fim, define-se Irretratabilidade como a exigência de que as ações de uma entidade sejam rastreadas exclusivamente a ela [7]. No âmbito deste projeto, a irretratabilidade se dá na medida em que quaisquer falhas de segurança devem ser rastreáveis, provando a responsabilidade de alguém para terceiros.

2.2 Security Protocol and Data Model (SPDM)

O Protocolo de Segurança e Modelo de Dados (SPDM, sigla em inglês para Security Protocol and Data Model) é uma especificação que define mensagens, objetos de dados e sequências de ações para a realização de trocas de mensagens entre dispositivos [9], sendo desenvolvido pela DMTF, uma organização de padrões de indústria.

Os mecanismos especificados são utilizados para autenticação de hardware e firmware, e desde a concepção do protocolo, em 2019, o protocolo já era visto como uma das principais maneiras de se garantir comunicação e medições seguras entre diferentes componentes de hardware e firmware de sistemas [10].

O protocolo SPDM define um modelo de mensagens do tipo requisição-resposta entre dois *endpoints*, realizando uma troca de mensagens específicas que tem como objetivos a descoberta de recursos de segurança, autenticação de identidade e a medição do firmware de um dos sistemas.

Cada um dos *endpoints* toma a função de Requester (que realiza as solicitações) ou Responder (que responde às solicitações), mas um determinado *endpoint* pode implementar recursos para tomar ambas as funções em diferentes trocas de mensagens. Todas as mensagens são iniciadas pelo Requester, enquanto que todas estas devem ser respondidas pelo Responder.

A especificação de descoberta e negociação de recursos de segurança se dá no início da troca de mensagens, com o Requester buscando descobrir, por exemplo, quais algoritmos de criptografia são aceitos pelo Responder. Nesta etapa, o Requester também pode definir um conjunto de algoritmos em comum entre os dois *endpoints* para transações futuras.

Quanto à autenticação de identidade do Responder, o processo se dá por meio de

assinaturas digitais geradas pelo *endpoint* a partir de uma chave privada. Estas assinaturas podem ser verificadas criptograficamente pelo Requester, utilizando a chave pública associada àquela chave privada.

Por fim, as medições de firmware se dão pelo processo de cálculo do valor de hash criptográfico de um firmware ou de dados de configuração desse, relacionando-o com a identidade do *endpoint* através do uso de assinaturas digitais.

O fluxo completo de mensagens do SPDM em alto-nível, no qual um *endpoint* que age como Requester manda mensagens SPDM para outro *endpoint*, que age como Responder e responde a essas com suas próprias mensagens SPDM, pode ser visto na Figura 2.

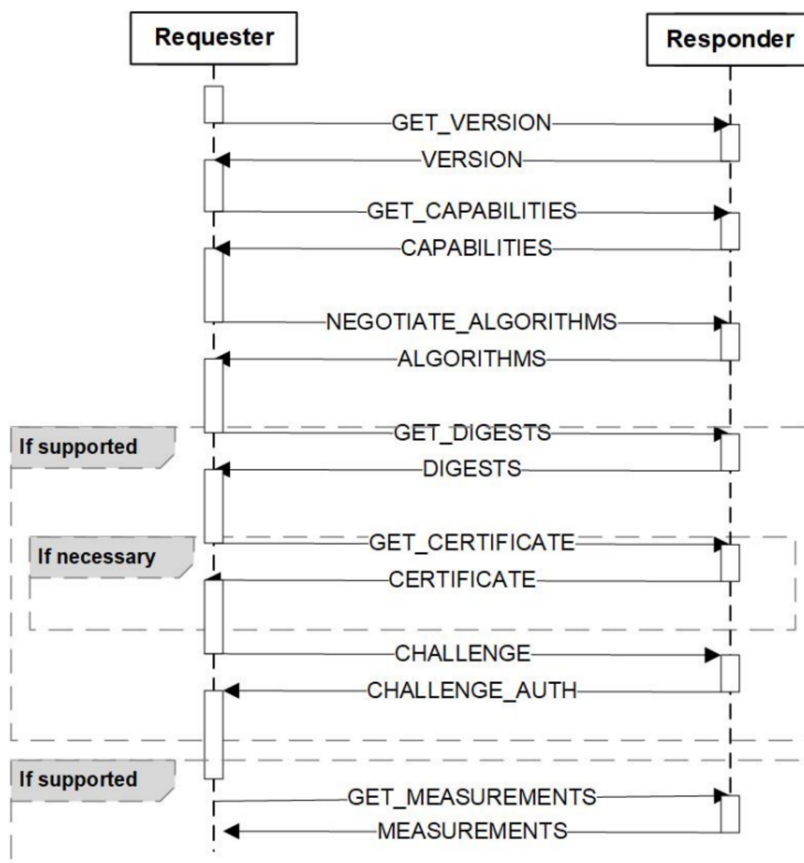


Figura 2: Fluxo de mensagens do SPDM.

Fonte: [9].

Esta especificação também inclui um método de autenticação e definição de sessão por meio de criptografia de chave simétrica, denominada Chave Pré-Compartilhada (PSK, sigla em inglês para Pre-Shared Key).

Neste método de troca de mensagens, tanto o Requester quanto o Responder devem conhecer uma chave secreta previamente antes da operação de *handshake*, que garante a

confiabilidade de ambos os lados da troca de mensagens quando efetuada com sucesso. Assim, deve-se levar em consideração a proteção desta chave secreta, idealmente sendo conhecida apenas pelos dois *endpoints* e, potencialmente, por terceiro confiável que fornece a chave aos *endpoints*.

Para a execução desse método, é utilizado um outro par de mensagens, `PSK_EXCHANGE`, cuja função se resume na verificação da consistência da chave secreta por ambos os lados da requisição.

O fluxo de mensagens padrão para esta forma de autenticação é descrito pela Figura 3.

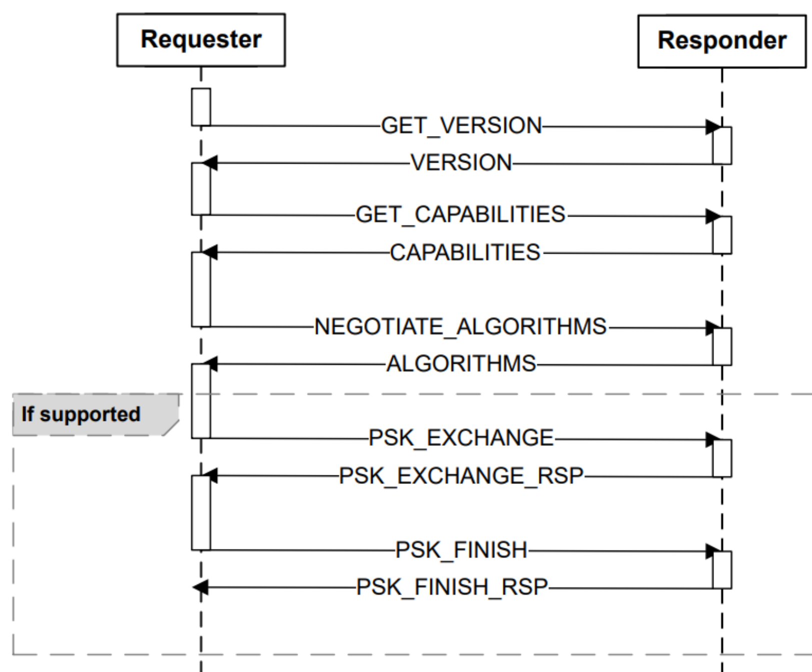


Figura 3: Fluxo de mensagens do SPDM com Chave Pré-Compartilhada.

Fonte: [9].

2.3 HMAC: Código de Autenticação de Mensagens com base em Hash

O Código de Autenticação de Mensagens com base em Hash (HMAC, sigla em ingês para Hash-based Message Authentication Code) é um mecanismo para autenticação de mensagens que utiliza funções hash criptográficas, como MD5 ou SHA-1, aliadas a uma chave secreta compartilhada [11].

Este algoritmo tem como objetivos a determinação de um método de autenticação de mensagens que consegue utilizar funções de hash já estabelecidas, sem causar perdas

significativas de desempenho, enquanto é utilizada uma maneira simples de gerenciar as chaves secretas envolvidas no processo de criptografia.

O processo de criptografia do algoritmo pode ser descrito pela seguinte equação [11]:

$$H(K \oplus opad, H(K \oplus ipad, text)) \tag{2.1}$$

No qual H é a função hash criptográfica utilizada, K é a chave secreta, $text$ é a mensagem a ser enviada, e $ipad$ e $opad$ são valores específicos fixos, referentes aos bytes $0x36$ e $0x5C$ repetidos um número de vezes igual ao tamanho B de blocos da função H em bytes, respectivamente.

Para realizar esta operação, deve-se, primeiramente, acrescentar zeros à direita da chave K, criando uma *string* de B bytes. Então, realiza-se a operação lógica XOR (operação de OU exclusivo) entre a nova chave e $ipad$, e utiliza-se este novo valor junto à mensagem $text$ como parâmetro para a operação de H.

Em seguida, é realizada a operação lógica XOR novamente, agora entre a mesma chave K de tamanho B e o valor de $opad$; o resultado desta operação é utilizado como parâmetro de H, junto ao valor obtido no passo anterior.

Um diagrama desta operação pode ser visto na Figura 4.

A segurança de qualquer mecanismo de autenticação de mensagens baseado em uma função hash criptográfica é diretamente ligada à segurança da função H utilizada. No caso do HMAC, no entanto, foi provada a relação exata entre a força de segurança da função hash utilizada e a força do HMAC como um todo [7].

Em particular, nota-se que a construção do algoritmo é voltada para o futuro, já que é independente da definição da função hash criptográfica H escolhida, bastando alterar apenas o tamanho do bloco B e a própria função H para se obter um novo HMAC [11]. No caso de surgir uma nova tecnologia de encriptação mais segura ou mais eficiente que as alternativas atuais, a substituição é relativamente simples.

Como será abordado a seguir, o HMAC é de suma importância para o funcionamento do TOTP, que utiliza este algoritmo com a função SHA-1 para o processamento de sua encriptação.

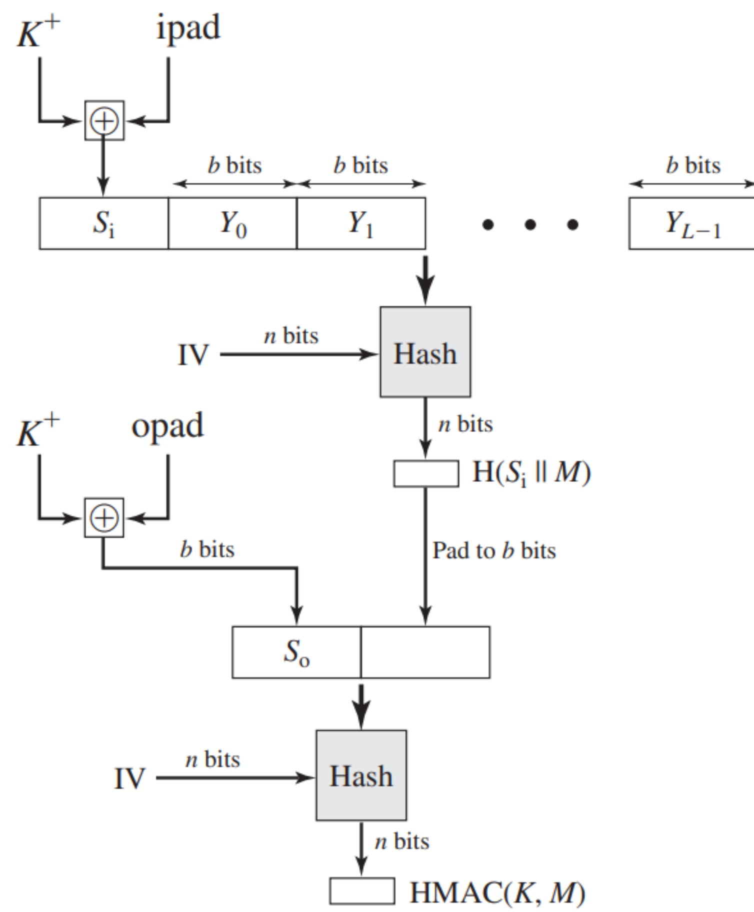


Figura 4: Estrutura do HMAC.

Fonte: [7].

2.4 Time-based One-Time Password (TOTP)

Time-Based One-Time Password (TOTP) é um algoritmo gerador de senhas de uso único. Esta aplicação é uma expansão do algoritmo de senhas únicas HMAC-Based One-Time Password (HOTP), porém gerando senhas com base no tempo atual, utilizando esta como métrica para garantia de sua singularidade [4].

O algoritmo HOTP é baseado no algoritmo HMAC-SHA-1, e é aplicado ao valor de um contador crescente, representando a mensagem da computação do HMAC. O valor da função é truncado, de tal maneira que o resultado seja facilmente escrito por um usuário [12]. A função de obtenção do HOTP a partir de uma chave secreta compartilhada K e um valor do contador C é dada pela seguinte equação:

$$HOTP(K, C) = Truncate(HMAC - SHA - 1(K, C)) \tag{2.2}$$

Na qual $Truncate$ é uma função responsável pela conversão do valor obtido com o algoritmo HMAC.

O TOTP configura-se como a versão baseada em tempo deste algoritmo, no qual um valor T referente ao tempo substitui o contador C da equação de obtenção do HOTP. Este valor depende do tempo Unix do momento da geração do valor, e a sincronização de ambos os sistemas é realizada por meio de *time-steps*, obtendo-se assim o valor T pela divisão do tempo Unix pelo *time-step* escolhido. Como exemplo, um *time-step* de 30 segundos implica em T sendo igual a 1 para qualquer valor de tempo Unix entre 30 e 59.

Ademais, é possível realizar uma operação de ressincronização para tratar os casos nos quais pode haver alguma diferenciação entre os relógios dos dois sistemas, na qual define-se um limite de *time-steps* aos quais o valor do TOTP é verificado. Assim, o validador da operação deve verificar também os valores $T - 1$ e $T + 1$, por exemplo, realizando um total de três validações.

O processo simplificado de funcionamento do TOTP pode ser visualizado na Figura 5.

Com isso, o algoritmo TOTP pode ser aplicado a dois sistemas distintos, possibilitando a operação de autenticação temporal de um deles face ao outro. O compartilhamento de chaves criptografadas garante a segurança do processo, requirindo que um possível atacante obtenha a chave privada do autenticante, além de garantir que a geração de senhas seja realizada em tempo real, dificultando ataques de *phishing* [14].

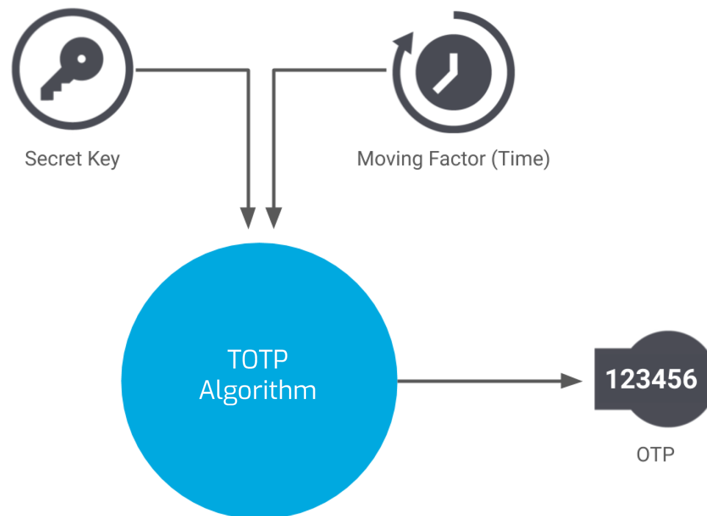


Figura 5: Esquematização do processo de funcionamento do TOTP.

Fonte: Adaptado de [13].

2.5 Kernel do Linux

Em qualquer sistema operacional moderno, pode-se encontrar uma camada do sistema referente ao kernel, responsável por providenciar serviços básicos a todos os outros componentes do sistema, além de gerenciar o hardware e distribuir os recursos do sistema. Assim, o kernel se configura como o núcleo do sistema operacional [15], como esquematizado na Figura 6.

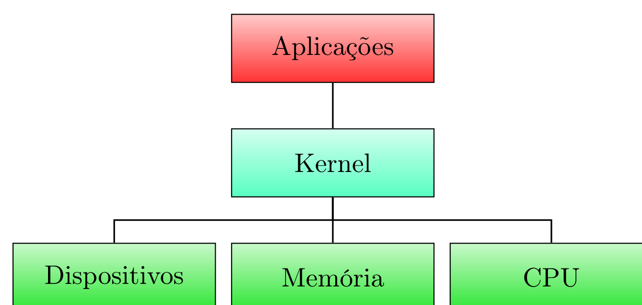


Figura 6: Esquematização do kernel como núcleo de um sistema operacional.

Fonte: do autor.

Em geral, este kernel é executado em um estado do sistema diferenciado de outras aplicações, incluindo uma seção de memória protegida e acesso completo ao hardware, que é conhecido como *kernel-space*, ou modo de kernel. Em contrapartida, aplicações comuns são executadas no *user-space*, ou modo de usuário. Estas aplicações podem, em geral, realizar chamadas a funções específicas do kernel, como `write()` ou `open()` por meio

de *system calls* [15].

Dessa forma, para os desenvolvimentos planejados neste trabalho, nota-se o kernel cumpre um papel importante, servindo como base para as implementações relevantes do SPDM e do TOTP.

3 ESPECIFICAÇÃO DE REQUISITOS

Neste capítulo, são apresentados os requisitos funcionais e não-funcionais especificados para o sistema, explorando todas as possíveis ideias que podem ser implementadas no escopo deste projeto. Vale ressaltar, no entanto, que nem todas estas serão implementadas no decorrer do trabalho; assim, algumas das ideias aqui apresentadas servirão apenas como possibilidades de expansões e continuidades do projeto no futuro.

Primeiramente, levanta-se o desenvolvimento de um dispositivo que, ao se conectar com um sistema principal, pode realizar as verificações de segurança do SPDM e do TOTP de maneira eficaz, dialogando com o kernel deste sistema. Para isso, este *dongle* possuirá um sistema operacional Linux modificado por meio da ferramenta Buildroot, e as tecnologias citadas serão implementadas em seu kernel.

O dispositivo também poderá conferir um método anti-furto ao sistema, impedindo que o computador seja utilizado enquanto o *dongle* não está conectado.

Por fim, para fins de experiência do usuário, deve haver uma forma de se verificar o *status* da conexão ao sistema principal, além de métricas de segurança do sistema conectado, como verificações realizadas com sucesso desde a inicialização. Para tal, podem ser utilizados LEDs ou *displays* no dispositivo, ou até uma conexão a um outro dispositivo do usuário que possa exibir as informações importantes (e.g. um celular recebendo as informações por meio de Wi-Fi).

Para esta aplicação, define-se os seguintes requisitos:

Conexão inicial ao dispositivo Para que as verificações de segurança sejam realizadas com sucesso, deve haver um sistema para a conexão do *dongle* ao dispositivo principal por USB.

Verificações periódicas de segurança O dispositivo, assim que conectado, deve realizar verificações constantes de segurança com o sistema principal por meio de SPDM e TOTP.

Impedimento de funcionamento em caso de desconexão abrupta Nos casos em que o *dongle* é desconectado abruptamente do sistema principal, deve-se impedir o funcionamento do sistema principal.

Interface para exportação de informações Deve haver um método para que o usuário possa obter informações sobre a conexão e sobre a segurança do sistema.

Confidencialidade Deve-se garantir que o acesso às informações e métricas importantes do sistema seja realizada apenas pelo *dongle* conectado.

Integridade Deve-se garantir que seja percebida qualquer modificação dos dados internos do firmware do sistema por um atacante, identificando-se, assim, quaisquer tentativas de alteração de seus dados internos.

Autenticidade Deve-se garantir que não houve falsificação dos dispositivos dialogando entre si por meio do SPDM.

Irretratabilidade Deve-se garantir também que quaisquer falhas de segurança sejam rastreáveis, provando a responsabilidade de algum elemento do sistema ou grupo participante para terceiros.

Análise dos dados pelo usuário O usuário deve poder conferir informações sobre a conexão e a segurança do sistema principal.

Vale notar, também, que esta aplicação poderia ser utilizada em conjunto com o dongle que realiza verificações a nível de kernel, fornecendo uma proteção ainda maior para um sistema, agindo a nível de kernel.

Por fim, define-se os seguintes requisitos funcionais e não funcionais para esta aplicação:

Conexão inicial ao dispositivo Para que as verificações de segurança sejam realizadas com sucesso, deve haver um sistema para a conexão do *dongle* ao dispositivo principal, possivelmente por USB.

Verificações de segurança no momento de boot O dispositivo, assim que conectado, deve realizar verificações constantes de segurança com o sistema principal por meio de SPDM e TOTP.

Confidencialidade Deve-se garantir que o acesso às informações e métricas importantes do sistema seja realizado apenas pelo *dongle* conectado.

Integridade Deve-se garantir que seja percebida qualquer modificação dos dados internos do firmware do sistema por um atacante, identificando-se, assim, quaisquer tentativas de alteração de seus dados internos antes mesmo do processo de inicialização do computador.

Autenticidade Deve-se garantir que não houve falsificação dos dispositivos dialogando entre si por meio do SPDM desde o momento de boot do sistema.

Irretratabilidade Deve-se garantir também que quaisquer falhas de segurança sejam rastreáveis, provando a responsabilidade de algum elemento do sistema ou grupo participante para terceiros.

Segurança dos componentes após a verificação inicial Depois das verificações no momento de boot, deve-se garantir que todos os componentes estejam seguros em todos os momentos seguintes.

Então, é apresentada também a possibilidade do uso do SPDM para a criação de uma Virtual Personal Area Network (VPAN). Com isso, buscar-se-ia permitir que dispositivos verifiquem uns aos outros numa mesma rede por meio do protocolo SPDM.

Esta aplicação possui grande potencial como facilitador de segurança em dispositivos de Internet das Coisas (IoT), as quais comumente necessitam de uma grande quantidade de dispositivos. Com esta aplicação, a segurança destes seria garantida de maneira mais fácil, já que todos os dispositivos teriam como se autenticar e se comunicar entre si com confidencialidade garantida. Desta forma, um ataque a um único membro da rede de dispositivos seria facilmente identificado e evitado.

Para esta aplicação, pode-se citar os seguintes requisitos funcionais e não funcionais:

Conexão entre dispositivos Para que as verificações de segurança sejam realizadas com sucesso, deve haver um sistema para a conexão dos dispositivos entre si.

Verificações periódicas de segurança Os dispositivos devem realizar verificações constantes de segurança entre si por meio de SPDM.

Confidencialidade Deve-se garantir que o acesso às informações e métricas importantes de todos os sistemas envolvidos seja realizada apenas por outros dispositivos autenticados e conectados à mesma rede.

Integridade Deve-se garantir que seja percebida qualquer modificação dos dados internos do firmware do sistema por um atacante, identificando-se, assim, quaisquer tentativas de alteração de seus dados internos de qualquer um dos dispositivos presentes na VPAN.

Autenticidade Deve-se garantir que não houve falsificação dos dispositivos dialogando entre si por meio do SPDM.

Irretratabilidade Deve-se garantir também que quaisquer falhas de segurança sejam rastreáveis, provando a responsabilidade de algum elemento do sistema ou grupo participante para terceiros.

4 TECNOLOGIAS UTILIZADAS

Para o melhor entendimento do projeto, é importante compreender as tecnologias que foram usadas no decorrer do desenvolvimento e dos testes realizados. Assim, neste capítulo, são citadas as ferramentas que foram utilizadas tanto para a codificação necessária quanto para as emulações e simulações realizadas para testes.

4.1 QEMU

Os sistemas desenvolvidos durante o projeto serão emulados com o auxílio do QEMU, uma aplicação open-source capaz de emular uma grande variedade de sistemas operacionais e processadores, possuindo, inclusive, suporte a KVM (Kernel-based Virtual Machine) [16].

A aplicação também confere maior facilidade para a emulação do kernel do Linux gerado a partir do Buildroot, que será utilizado no projeto para a simulação do sistema com SPDM e TOTP a nível de kernel.

O QEMU é publicado com licença do tipo GNU GPL [17], possibilitando a alteração de seu código fonte, o que será necessário para a simulação das trocas de mensagens SPDM entre o Requester e o Responder no ciclo de mensagens padrão da especificação.

Este foi utilizado no projeto não apenas como ferramenta para emulação do sistema *host* nos testes da aplicação, mas também foi fundamental como *framework* para o desenvolvimento do dispositivo emulado, que utilizou funções específicas a esta aplicação.

4.2 Buildroot

Buildroot é uma ferramenta de geração de sistemas Linux simples, que podem ser facilmente emulados [18].

Para tal, o Buildroot é capaz de gerar uma *toolchain* de compilação cruzada, um

sistema de arquivos, uma imagem de kernel do Linux e um *bootloader*. Com isso, permite-se que os usuários criem uma distribuição do Linux customizada, facilitando possíveis alterações a serem feitas no código fonte do sistema operacional e possibilitando a remoção de *features* desnecessárias ao projeto.

A aplicação também permite a fácil adição de bibliotecas e de opções para depuração, facilitando a criação de um kernel Linux customizado para as necessidades do grupo.

O Buildroot foi utilizado pelo grupo para a criação de duas imagens distintas do Linux. Uma delas foi criada com o intuito de servir como sistema *host* na emulação do QEMU, executando o driver desenvolvido neste trabalho. Por outro lado, a outra imagem foi criada visando ser utilizada com a placa Intel Galileo, no dispositivo físico que responderia às requisições do driver.

4.3 libspdm

libspdm [19] é uma biblioteca em C que implementa a troca de mensagens determinada pela especificação SPDM. Ela inclui não apenas a implementação das funções necessárias, como descrito na especificação do protocolo, como também uma implementação base dos dois *endpoints* necessários para a comunicação.

A biblioteca é disponibilizada em código aberto no GitHub da DMTF, criadores da especificação SPDM, sob a licença BSD 3-Clause.

A versão utilizada da biblioteca neste projeto conta tanto com alterações realizadas previamente pelos professores do LARC, quanto com eventuais alterações realizadas pelos integrantes do grupo para inseri-la no contexto de módulos do kernel.

A libspdm foi utilizada em todos os códigos desenvolvidos para este trabalho – em particular, no driver e nos códigos do dispositivo USB. Para tal, esta também precisou ser inserida nos kernels do Linux que o grupo gerou com o Buildroot.

4.4 GDB

GNU Debugger, ou GDB, é um depurador feito originalmente para o sistema GNU, mas ainda é usado comumente para depurar códigos em diversas linguagens, em particular em C [20].

O projeto possui facilidades em seu uso com o Buildroot, permitindo a depuração

cruzada da imagem do Buildroot sendo executada em uma máquina virtual, enquanto a máquina *host* conecta-se à VM por TCP e permite não apenas o uso de funções padrão do GDB, como inserção de *breakpoints*, como também possui um conjunto de scripts em python que permitem a extração de mais informações específicas da VM, como o `lx-dmesg`, por exemplo, que imprime o log completo do comando `dmesg` no terminal do GDB.

Neste projeto, o GDB foi de suma importância para a depuração e para a visualização de informações do sistema *host*, sendo utilizado tanto nos testes quanto na demonstração do protótipo.

5 PROJETO E IMPLEMENTAÇÃO

Neste capítulo, descreve-se a metodologia de trabalho e o projeto de maneira resumida, fornecendo uma visão geral do sistema desenvolvido, e então se exibe o processo completo de implementação do projeto como um todo. Este foi dividido pelo grupo em etapas, de maneira a facilitar a identificação de áreas que necessitariam um estudo mais aprofundado por parte dos alunos, além do desenvolvimento dos códigos necessários para a execução do projeto.

5.1 Metodologia do Trabalho

Sobre a metodologia de trabalho, vale ressaltar, primeiramente, que a especificação SPDM é relativamente recente, não tendo, assim, muitas implementações base. Visto isso, o grupo identificou a necessidade de se realizar um estudo a fundo da tecnologia como precedente para o projeto.

Realizados os estudos necessários, foi priorizada uma organização da especificação de requisitos do projeto, feita em conjunto com os orientadores, de forma a determinar quais são as necessidades do sistema para que seja possível suprir os problemas de segurança que o projeto planeja atacar.

Em seguida, foram desenvolvidos os códigos necessários, referentes à implementação do SPDM e do TOTP a nível de kernel, com esse sendo executado no ambiente emulado pelo QEMU. A implementação foi feita em linguagem C, com o uso de bibliotecas como a libspdm [19] para a implementação das tecnologias em questão.

Também foi desenvolvido um *dongle* que pode se comunicar com o sistema *host* via USB, de forma que a verificação é feita de maneira periódica desde o momento do boot do equipamento. Este código foi criado inicialmente como um dispositivo emulado pelo QEMU, utilizando funções específicas deste sistema, além das mesmas bibliotecas utilizadas pelo código do driver.

Então, utilizando este código como base, foi desenvolvido também um dispositivo físico, baseado no dispositivo emulado pelo QEMU. Para este dispositivo, foi utilizada a placa Intel Galileo. Depois da configuração da placa como dispositivo USB OTG , foi necessária a criação de uma nova imagem do Linux com o Buildroot, para então se desenvolver o código em C, que deve ser executado no dispositivo com a mesma funcionalidade do dispositivo emulado pelo QEMU.

Por fim, foram realizados testes com os produtos finais gerados pelo projeto, comprovando-se a eficácia da solução de segurança capaz de resolver problemas em nível de firmware e ataques à *supply chain*, atentando-se também ao desempenho dos dispositivos.

5.2 Visão geral do sistema

O sistema como um todo é composto principalmente por dois elementos: o driver USB e o dispositivo emulado pelo QEMU. O driver possui a função de interagir com o dispositivo criado durante o desenvolvimento do projeto. Este enviará e receberá mensagens USB por meio de blocos de mensagens utilizados nesse protocolo, denominados URBs. Enquanto isso, o dispositivo é responsável por responder às requisições por meio do protocolo SPDM e do algoritmo TOTP. A Figura 7 ilustra o funcionamento do sistema como um todo.

O fluxo de execução do driver começa com a busca pelo dispositivo USB em questão. A busca é realizada de maneira simples, por um Vendor ID e Product ID especificado previamente.

Em seguida, tem-se a inicialização da biblioteca responsável pelas verificações do protocolo SPDM, definindo as mensagens e as ferramentas disponíveis para o processamento da libspdm.

Feitas as configurações iniciais, inicia-se o processo de comunicação entre driver e dispositivo, no qual a comunicação SPDM é estabelecida.

Com isso, pode-se também iniciar as verificações periódicas, que utilizarão o SPDM em conjunto com o TOTP. Neste momento da execução, o driver e o dispositivo deverão, inicialmente, chegar a um acordo quanto à chave secreta a ser utilizada na geração dos códigos TOTP. Este processo é realizado por meio da geração de números aleatórios em ambos os lados da comunicação, com tamanho de 64 bits. Os dois números são concatenados pelo dispositivo, que realiza uma operação de *hashing* utilizando o algoritmo SHA384, e gera a chave que será utilizada. Esta chave é, então, enviada de volta ao driver,

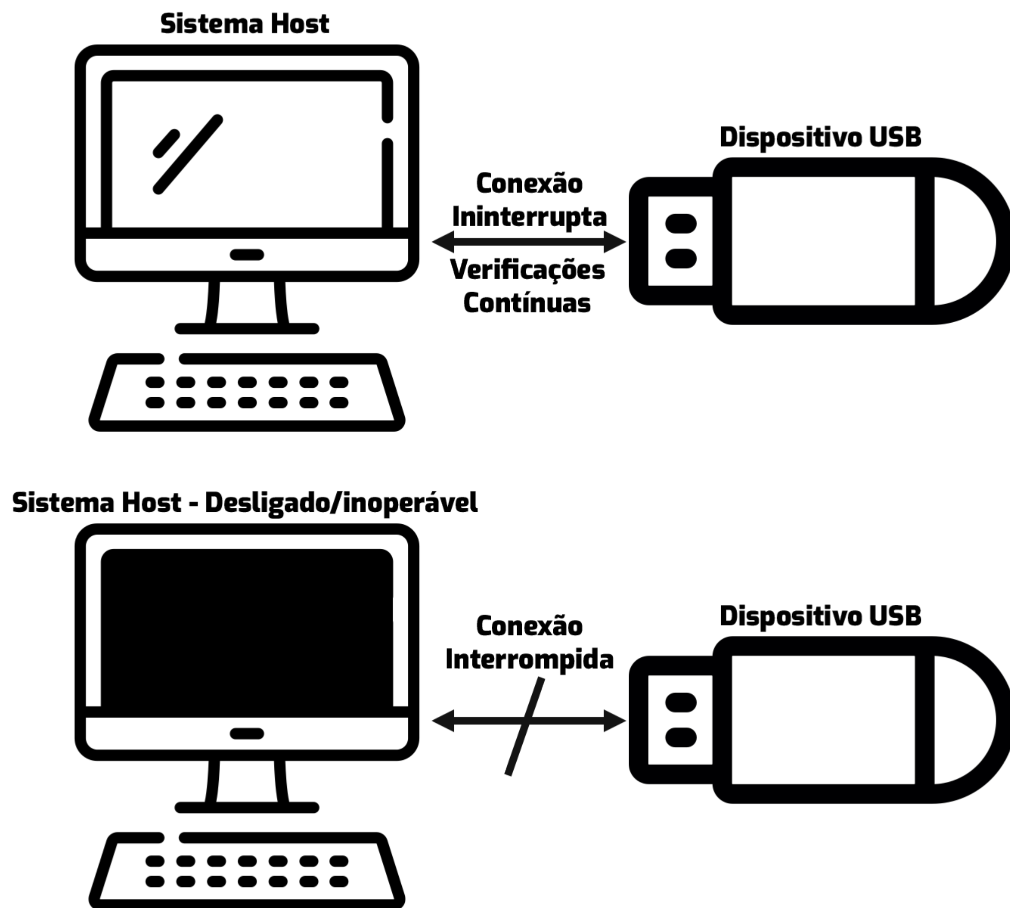


Figura 7: Esquematização do funcionamento do sistema.

Fonte: do autor.

permitindo que este também gere códigos TOTP localmente.

Com ambos os lados da comunicação tendo a chave disponível, o dispositivo gera um código TOTP e o envia ao driver. Ao fazer o mesmo, o driver consegue confirmar se a comunicação ocorreu como esperado ao se comparar o valor recebido e o valor gerado localmente.

Esta troca de mensagens acontece por meio de mensagens criptografadas às regras do SPDM, de modo que o estabelecimento prévio da comunicação SPDM é essencial para a execução deste fluxo.

O driver também conterá uma função que desliga o sistema caso algum problema ocorra durante a execução, como, por exemplo, uma alteração a nível de firmware da máquina *host* detectada pelo SPDM, uma falha na conexão USB, ou um desacordo na troca de códigos TOTP entre driver e dispositivo.

Por outro lado, o dispositivo USB deve poder responder apropriadamente a todos estes fluxos – desde a criação de um contexto SPDM próprio, no qual as configurações iniciais quanto às ferramentas disponíveis para a execução da libspdm são definidas, até a geração de uma chave secreta para o TOTP, a geração do código TOTP e seu envio.

A execução correta destes ciclos de maneira periódica permite ao sistema que os requisitos de segurança especificados na seção 3 sejam atingidos.

O esquema da Figura 8 ilustra o fluxo geral da execução do sistema.

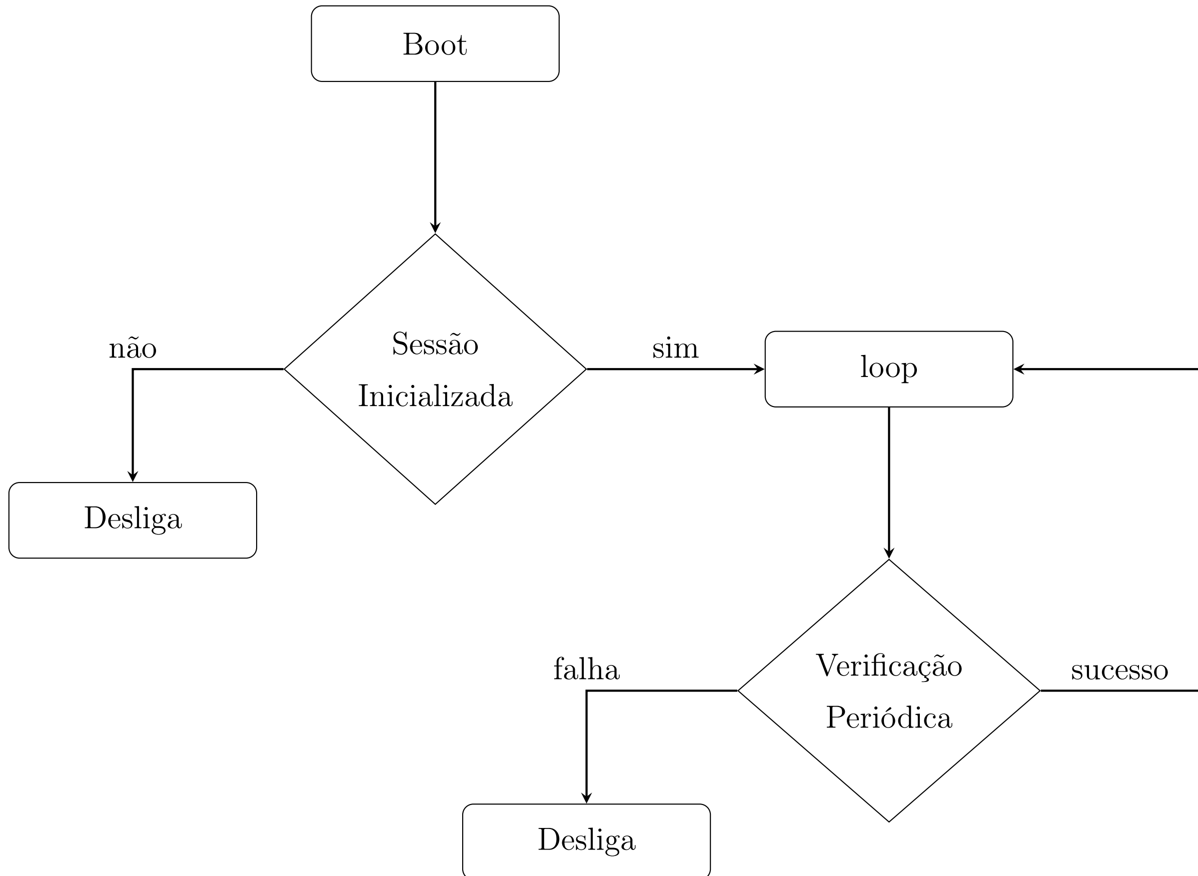


Figura 8: Ilustração do fluxo de execução do sistema desenvolvido.

Fonte: do autor.

5.3 Configuração inicial do ambiente

O primeiro passo para o desenvolvimento e teste dos códigos relevantes consiste na configuração de um ambiente de desenvolvimento na plataforma Linux. Para tal, foi escolhido o Ubuntu 20.04, que conta com várias ferramentas essenciais para as operações a serem realizadas, como o `make`, `cmake`, e `GCC`. Vale ressaltar, no entanto, que outras distribuições do Linux também podem ser utilizadas, feitas as devidas alterações nos comandos utilizados.

Em primeiro lugar, foi necessária a compilação do libspdm. Para tal, basta seguir as instruções contidas no arquivo `README.md` do próprio repositório da DMTF [19], utilizando-se os seguintes argumentos para a compilação no diretório base do repositório da libspdm:

```
$ mkdir build
$ cd build
```

```
$ cmake -DARCH=x64 -DTOOLCHAIN=GCC -DTARGET=Debug \
   -DCRYPTO=mbedtls ..
$ make copy_sample_key
$ make
```

Trecho de Código 5.1: Comandos para compilação da libspdm

Como pode-se notar, para esta ferramenta, estão sendo utilizados o GCC como *toolchain* e o Mbed TLS como biblioteca para a criptografia.

Em seguida, é necessário realizar a compilação do QEMU, já que a versão utilizada para o desenvolvimento deste projeto já possui o libspdm integrado. Esta versão foi desenvolvida por pesquisadores do LARC (Laboratório de Arquitetura e Redes de Computadores), estando disponível no GitLab do laboratório, e foi baseada na versão 4.1 do QEMU original.

A compilação do QEMU é realizada da seguinte maneira, do diretório base do repositório do aplicativo:

```
$ mkdir build
$ cd build
$ ../configure --enable-libspdm --libspdm-crypto=mbedtls \
   --libspdm-srcdir=<libspdm-srcdir> \
   --libspdm-builddir=<libspdm-builddir> \
   --target-list=x86_64-softmmu
$ make
```

Trecho de Código 5.2: Comandos para compilação do QEMU.

Nestes comandos, `<libspdm-srcdir>` é o diretório base do libspdm, e `<libspdm-builddir>` é o diretório específico no qual a `build` foi realizada.

Em seguida, deve ser realizada a compilação e geração da imagem do kernel do Linux com o auxílio da ferramenta Buildroot. A versão utilizada da ferramenta foi a 2020.02.9, com o kernel compilado sendo a versão 4.19.

Antes do desenvolvimento de códigos adicionais, é importante criar uma versão inicial com as configurações padrão do Buildroot, com todos os requisitos mínimos para o funcionamento do kernel. Para tal, deve-se executar os seguintes comandos na pasta principal do Buildroot:

```
$ make qemu_x86_64_defconfig
```

```
$ make
```

Trecho de Código 5.3: Comandos para compilação do Buildroot.

Para se modificar o sistema compilado, pode-se utilizar o comando `make menuconfig` para se ativar ou desativar manualmente itens para a compilação inicial, como adicionar ou remover programas instalados por padrão. Além disso, para se configurar especificamente o kernel depois da build inicial, pode-se utilizar também a instrução `make linux-menuconfig`.

Com isso, é possível executar o comando base para os testes que serão realizados por meio da emulação com o QEMU da seguinte maneira:

```
$ ./<qemu-executable-path> -kernel <buildroot-bzImage-path> \
   -drive file=<buildroot-rootfs.ext2-path>, \
   if=ide,format=raw \
   -append "console=ttyS0 rootwait root=/dev/sda nokaslr" \
   -m 1024
```

Trecho de Código 5.4: Comando para execução do QEMU com o kernel compilado.

No qual `qemu-executable-path` corresponde ao caminho para o executável do QEMU de 64 bits (`qemu-system-x86_64`), enquanto que `buildroot-bzImage-path` é o caminho para o arquivo `bzImage`, referente à imagem do kernel gerado pelo Buildroot, e `buildroot-rootfs.ext2-path` é o caminho para o arquivo `rootfs.ext2`, referente ao sistema de arquivos do sistema. Ambos os dois sendo obtidos a partir da compilação do Buildroot, em geral no diretório `output/images` do Buildroot.

Este comando, se executado corretamente, abrirá uma janela com uma máquina virtual emulada pelo QEMU com base no kernel compilado previamente com o Buildroot.

5.4 Adição de driver ao Buildroot como módulo

Para a inserção de um driver ao kernel compilado com o Buildroot, torna-se necessária a adição de um módulo ao Buildroot. Este processo pode ser realizado de diversas formas, ressaltando-se a criação de um módulo externo e a recompilação do kernel com os novos arquivos e modificações aos `Makefiles` como módulo interno.

O grupo inicialmente buscou criar um módulo externo, em particular devido à facili-

dade de organização que este método confere ao projeto. No entanto, surgiram problemas no momento de adição do biblioteca libspdm, que necessitaria de uma grande quantidade de alterações em todo o código, de tal forma que fosse adaptado ao formato de módulo do kernel. Por exemplo, seria necessário alterar todos os *includes* do código da libspdm para conformidade com os *headers* disponíveis a nível de kernel, exportando-se as funções necessárias para o espaço de kernel por meio de funções como `EXPORT_SYMBOL`.

Por outro lado, as alterações necessárias para que a libspdm fosse adicionada a módulos internos já foram realizadas pelos professores do LARC. Visto isso, o grupo decidiu seguir pelo caminho de adição do driver como módulo interno ao Buildroot. Ainda, assim, por completude, decidiu-se documentar também a seção sobre a adição de módulo externo, como visto na Seção 5.4.1.

5.4.1 Adição de módulo externo ao Buildroot

Para utilizar este método, o módulo a ser desenvolvido deverá estar localizado em uma pasta exterior ao Buildroot em si, sendo então adicionado e compilado como módulo externo por meio da interface gráfica do `make linux-menuconfig`.

Além disso, o driver, quando inserido no kernel, deverá ser inicializado no momento de *boot* do dispositivo. Este passo é realizado por um script *daemon*, que é inicializado pelo BusyBox, contido no Buildroot, de maneira automática. O script contém chamadas à função `modprobe`, ativando, assim, os drivers necessários.

Para utilizá-lo, deve-se definir também uma *overlay* para os arquivos do sistema de inicialização do kernel, adicionando, assim, o arquivo necessário para a execução automática do driver. Esta operação é realizada alterando-se a variável `BR2_ROOTFS_OVERLAY` com o valor `BR2_ROOTFS_OVERLAY = $(BR2_EXTERNAL_TOTP_SPDM_PATH)/rootfs-overlay`.

Com isso em mente, ressalta-se que a estrutura do módulo deve ser condizente com o seguinte modelo:

```
diretório base
├─ Config.in
├─ driver.c
├─ Makefile
├─ external.desc
├─ external.mk
└─ rootfs-overlay
   └─ etc
      └─ init.d
         └─ SNNscript
```

Para realizar a compilação do Buildroot com o driver adicionado, deve-se definir algumas variáveis próprias da aplicação para garantir que os arquivos sejam adicionados corretamente. Para tal, deve-se inicialmente definir a variável `BR2_EXTERNAL` para o caminho do diretório no qual se encontram os arquivos necessários. Isso pode ser realizado no momento em que se chama o comando `make menuconfig`, da seguinte maneira:

```
$ make BR2_EXTERNAL=/home/user/external-module-location \
    menuconfig
```

Trecho de Código 5.5: Comando `menuconfig` com especificação do módulo externo.

Este comando também possibilitará que outras variáveis do Buildroot utilizem o caminho para o diretório com a variável `BR2_EXTERNAL_[DIRECTORY-NAME]_PATH`, com o nome em questão sendo definido em um dos arquivos do diretório.

Na interface visual, deve-se ativar os drivers adicionados na opção *totp-spdm*, dentro do menu *External Options*.

5.4.2 Adição de módulo interno ao Buildroot

A criação de um módulo interno ao Buildroot é mais simples. Para tal, dado que o kernel do Linux já foi compilado pelo Buildroot, basta copiar os arquivos relevantes para o diretório ao qual eles pertencem, tendo em mente que os `Makefiles` relevantes também devem ser modificados para incluir a compilação dos novos arquivos. Em seguida, deve-se executar o comando `make linux-rebuild`, que recompila o kernel com os novos arquivos.

Esta simplicidade na compilação também se mostra como uma redução do tempo necessário para se criar uma nova versão do sistema operacional após alguma mudança. Um módulo externo precisa ser compilado por si só pelo Buildroot, com uma recompilação do próprio kernel não sendo o bastante para obter-se as novas mudanças, já que seus arquivos relevantes não são exclusivos ao kernel. Embora um módulo externo possa ser rapidamente recompilado, no caso de mudanças serem poucas e contidas apenas nele, qualquer mudança maior pode forçar a recompilação de muitos outros módulos, gerando uma demora muito maior que a esperada.

Adicionalmente, este método dispensa a criação de um script *daemon* como o anterior, justamente pelo conjunto de arquivos já estar inserido diretamente nos drivers do kernel. Dessa forma, sua execução com a inicialização do sistema se dá de maneira automática.

Assim, para o driver em questão, bastaria modificar o `Makefile` já existente no diretório `output/build/linux-4.19.91/drivers/usb`, contendo o caminho para o novo diretório, e criar o novo driver e `Makefile` para sua compilação. Estes arquivos podem ser modificados diretamente do código fonte do kernel já compilado pelo Buildroot, ou pode ser colocado em um diretório à parte, sendo apenas copiado ao kernel em si quando se deseja recompilá-lo.

Com isso, a estrutura de arquivos necessária para o uso deste método resume-se em:

```
diretório base
└─ drivers
   └─ usb
      ├─ Makefile
      └─ totp+spdm
         ├─ Makefile
         └─ driver.c
```

5.5 Configuração de ambiente de depuração com GDB

Durante o desenvolvimento de um driver para o kernel do Linux, é comum que apareçam diversos erros em diferentes etapas da escrita do código, e o processo de depuração pode se mostrar como um grande empecilho e disperdiçar grandes quantidades de tempo dos desenvolvedores.

Para evitar tal cenário, é de suma importância que se possibilite alguma forma mais

eficiente de depuração que simples *prints*, em particular devido à possível ocorrência de pânicos no kernel, que impossibilita a verificação do valor de variáveis ou do conteúdo de posições de memória do sistema.

Para tal, optou-se por utilizar o GDB como ferramenta de depuração.

Antes de qualquer uso do GDB como ferramenta, deve-se garantir que ele está instalado na máquina *host*. Então, pode-se habilitar o GDB na imagem gerada pelo Buildroot ativando a opção *Build cross gdb for the host*, dentro do menu *Toolchain*, na interface gráfica do `menuconfig`.

Depois do fim da execução, deve-se alterar também as opções referentes à disponibilização de informações de depuração no momento da compilação e de scripts para uso no GDB por meio interface gráfica do `linux-menuconfig`, como indicado pela Figura 9.

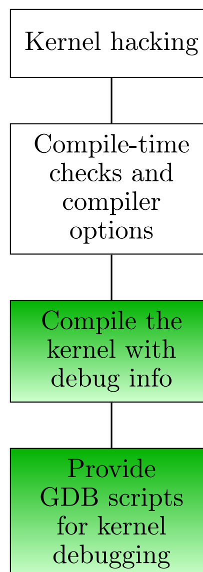


Figura 9: Caminho para ativação de opção do Buildroot para depuração do kernel.

Fonte: do autor.

Feito isso, o kernel compilado já contém as informações necessárias para o uso do GDB. O próximo passo se dá pela adição do *script* de comandos para o GDB do Buildroot ao arquivo `.gdbinit` do GDB da máquina *host*. Isso pode ser realizado por meio da execução do seguinte comando em um terminal:

```
$ echo "add-auto-load-safe-path \
   <buildroot-linux-path>/vmlinux-gdb.py" \
   >> ~/.gdbinit
```

Trecho de Código 5.6: Comando para adição de *scripts* do GDB.

No qual `buildroot-linux-path` corresponde ao caminho para a base do kernel compilado, geralmente disponível em `output/build/linux-4.19.91`.

Isso permite que sejam utilizados os comandos especiais do Buildroot, como o `lx-dmesg`, que imprime o log completo do comando `dmesg` no terminal do GDB.

A execução do GDB em si se dá, primeiramente, pela adição das flags `-S -s` ao final do comando do QEMU. A opção `-s` fará com que QEMU aguarde uma conexão de entrada do GDB na porta TCP 1234, enquanto que `-S` fará com que o QEMU não inicie a máquina virtual até que o usuário o faça pelo GDB [16].

Então, em outro terminal, pode-se executar o comando `gdb`, especificando-se o caminho para o arquivo `vmlinux`, também contido no diretório `output/build/linux-4.19.91`. Dentro do terminal próprio da aplicação, deve-se executar o comando `target remote :1234`, que realiza a conexão com o processo em execução do QEMU.

Com isso, já é possível realizar operações comuns de depuração com o GDB, além de se utilizar o *script* específico do Buildroot para ações mais específicas do kernel do Linux, como listar os módulos atualmente carregados com `lx-lsmod` ou exibir o *buffer* de logs do kernel com `lx-dmesg`.

Além disso, também é possível utilizar o GDB como ferramenta de depuração do QEMU em si, caso haja a necessidade de se depurar alguma alteração realizada no próprio QEMU. Para tal, pode-se utilizar o comando `gdb -args`, especificando-se, então, o comando completo para a execução do QEMU. Como exemplo, pode-se utilizar o seguinte comando:

```
$ ./gdb --args <qemu-executable-path> -kernel \
   <buildroot-bzImage-path> -drive \
   file=<buildroot-rootfs.ext2-path>,if=ide,format=raw \
   -append "console=ttyS0 rootwait root=/dev/sda nokaslr" \
   -m 1024
```

Trecho de Código 5.7: Comando para depuração do QEMU.

5.6 Desenvolvimento do dispositivo USB emulado pelo QEMU

O desenvolvimento de um dispositivo USB a ser emulado pelo QEMU configura-se como uma das principais tarefas para a execução deste projeto. Este dispositivo deverá

conter uma implementação tanto do protocolo SPDM quanto do algoritmo TOTP, mantendo um constante diálogo com um driver próprio para este dispositivo. Em intervalos de tempo pré-definidos, mensagens serão trocadas entre driver e dispositivo, que atuarão como Requester e Responder no fluxo padrão de mensagens do protocolo SPDM.

Para tal, é importante notar primeiramente a falta de documentação disponível para esta atividade. Dentro do código-fonte do QEMU, encontram-se alguns códigos prontos para dispositivos USB, cada qual com um propósito específico, como dispositivos de *mass storage* (e.g. pen-drives), de leitura serial, ou de áudio. Estes, no entanto, são pobres em comentários, dificultando a compreensão dos códigos já existentes, além de não haver uma documentação extensa o bastante a ponto abordar o desenvolvimento dos dispositivos em si, apenas seu uso.

Com isso, o entendimento básico de um dos dispositivos mostrou-se como uma das atividades iniciais necessárias para o início do desenvolvimento.

5.6.1 Entendimento de dispositivo USB básico

Como exemplo, o dispositivo básico do QEMU referente à comunicação serial, `dev-serial`, deve ser chamado da seguinte maneira:

```
$ ./<qemu-executable-path> -kernel <buildroot-bzImage-path> \
    -drive file=<buildroot-rootfs.ext2-path>, \
    if=ide,format=raw \
    -append "console=ttyS0 rootwait root=/dev/sda nokaslr"
    -m 1024 -chardev socket,id=channel0,path=/tmp/port0 \
    -usb -device usb-serial,bus=usb-bus.0,chardev=channel0
```

Trecho de Código 5.8: Comando para execução do QEMU com dispositivo serial.

A inicialização de qualquer dispositivo USB no QEMU dá-se, inicialmente, pelo argumento `-usb`, que inicializa um *USB hub* do Linux, necessário para a adição de outros dispositivos. Então, o comando `-device usb-serial,bus=usb-bus.0,chardev=channel0` é responsável por criar uma instância do dispositivo base definido no QEMU como `usb-serial`, que emula a interface USB para UART serial FT232B, da FTDI.

Neste caso específico, também é necessária a passagem do argumento `-chardev socket,id=channel0,path=/tmp/port0`, que define um dispositivo para transporte e exibição de caracteres no caminho `/tmp/port0` da máquina *host*.

Para a execução da simulação, no entanto, vale ressaltar que o Linux compilado pelo Buildroot deverá conter algum método de transmissão serial, não contido na *build* padrão. Como exemplo, para se utilizar a ferramenta *minicom*, pode-se habilitar a opção *enable wchar support*, dentro do menu *Toolchain*, e o próprio pacote do *minicom*, dentro de *Target Packages -> Hardware Handling*. Além disso, outras opções de suporte ao aplicativo devem ser ativadas, como indicado na Figura 10.

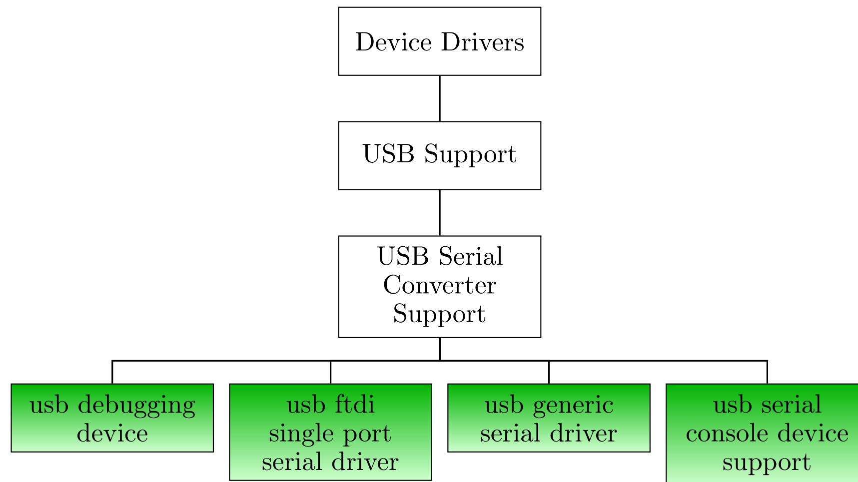


Figura 10: Caminho para ativação de opção do Buildroot para permitir comunicação serial.

Fonte: do autor.

Realizada a recompilação do kernel, pode-se executar o comando `socat UNIX-LISTEN:/tmp/port0 -` em um outro terminal do *host*, enquanto que, na máquina virtual em si, executa-se o comando `minicom -s`, definindo-se uma configuração para porta serial no dispositivo `ttyUSB0`. Com isso, qualquer dado digitado no terminal do `minicom` aparecerá no terminal do *host* no qual o servidor (`socat`) está sendo executado.

O código deste dispositivo pode ser visto no GitHub do QEMU [17], sob o caminho `hw/usb/dev-serial.c`. Nele, nota-se alguns detalhes importantes que serão úteis para o desenvolvimento de um novo dispositivo emulado pelo QEMU.

Em primeiro lugar, é importante definir um *struct* com todas as variáveis relevantes ao código. No dispositivo serial, este é denominado `USBSerialState`, guardando, por exemplo, a latência permitida e o *buffer* de transmissão para transferência de dados.

Em seguida, vale notar como é realizada a inicialização das funções relevantes, que é notada nas últimas linhas do arquivo em questão. Aqui, encontram-se as funções `usb_serial_register_types`, responsável por registrar funções de descrição e de inicialização; `usb_serial_class_initfn`,

usb_braille_class_initfn e usb_serial_dev_class_init, responsáveis por inicializar as classes de dispositivos e definir funções para gerenciar ações específicas do dispositivo (usb_serial_realize, usb_serial_handle_reset, usb_serial_handle_control e usb_serial_handle_data); e as constantes vmstate_usb_serial, serial_info e serial_properties, que definem valores globais para encontrar outros dispositivos ou permitir o descobrimento dele mesmo por um driver.

5.6.2 Criação de um novo dispositivo USB emulado pelo QEMU

Com isso tudo em mente, pode-se iniciar o desenvolvimento do novo dispositivo, denominado *usb-totp-spdm*, que se encontrará no arquivo dev-totp-spdm.c.

Criado o arquivo, este deverá ser adicionado ao Makefile relevante. Por estarmos trabalhando com dispositivos USB, altera-se o arquivo hw/usb/Makefile.objs, adicionando-se a seguinte linha:

```
common-obj-$(CONFIG_USB_TOTP_SPDM) += dev-totp-spdm.o
```

Trecho de Código 5.9: Adição de novo dispositivo ao Makefile.

Com isso, o Makefile compilará o arquivo criado se passada a *flag* específica. Então, no arquivo configure, localizado no diretório raiz do repositório, adiciona-se a seguinte instrução:

```
echo "CONFIG_USB_TOTP_SPDM=y" >> $config_target_mak
```

Trecho de Código 5.10: Instrução para o arquivo configure.

Com isso, a execução do comando configure adicionará a compilação do arquivo às instruções realizadas pelo comando make. Com isso, deve-se executar novamente os comandos referentes à compilação do QEMU, como indicado a seguir.

```
$ cd build
$ ../configure --enable-libspdm --libspdm-crypto=mbedtls \
   --libspdm-srcdir=<libspdm-srcdir> \
   --libspdm-builddir=<libspdm-builddir> \
   --target-list=x86_64-softmmu
$ make
```

Trecho de Código 5.11: Comandos para compilação do QEMU.

Nestes comandos, `<libspdm-srcdir>` é o diretório base do libspdm, e `<libspdm-builddir>` é o diretório específico no qual a `build` foi realizada.

A partir deste momento, o dispositivo pode ser adicionado à execução da máquina virtual por meio do seguinte comando:

```
$ ./<qemu-executable-path> -kernel <buildroot-bzImage-path> \
    -drive file=<buildroot-rootfs.ext2-path>, \
    if=ide,format=raw \
    -append "console=ttyS0 rootwait root=/dev/sda nokaslr" \
    -m 1024 -usb -device usb-totp-spdm,bus=usb-bus.0
```

Trecho de Código 5.12: Execução do QEMU com o novo dispositivo USB.

No qual `qemu-executable-path` corresponde ao caminho para o executável do QEMU de 64 bits (`qemu-system-x86_64`), enquanto que `buildroot-bzImage-path` é o caminho para o arquivo `bzImage` e `buildroot-rootfs.ext2-path` é o caminho para o arquivo `rootfs.ext2` do Buildroot.

Definidos os elementos básicos do dispositivo, como citado anteriormente, pode-se definir elementos específicos do dispositivo, que serão relevantes no processo de comunicação com o driver.

Em particular, nota-se a função `usb_totp_spdm_handle_data`, responsável pela transmissão de dados entre driver e dispositivo. Esta função é chamada automaticamente toda vez que uma mensagem chega ao dispositivo por meio de um USB Request Block (URB).

O detalhe mais importante desta função é o pacote recebido – no QEMU, este é chamado de `USBPacket`. Esta estrutura de dados possui uma grande quantidade de variáveis, incluindo o *endpoint* de destino do dispositivo, o *buffer* referente à transmissão, seu tamanho, e um identificador do tipo de transmissão.

Dentro da função, que não diferencia chegada ou saída de dados por si só, é necessária uma avaliação deste identificador, denominado `pid`, para descobrir o tipo de comunicação que está sendo efetuada. Para tal, o QEMU disponibiliza as flags `USB_TOKEN_OUT` e `USB_TOKEN_IN`, que simbolizam a saída e a chegada de dados no driver, respectivamente.

Em ambos os casos, o QEMU também disponibiliza a função `usb_packet_copy`, que recebe como parâmetros o pacote referente à comunicação atual, um *buffer* do qual os dados serão lidos (ou escritos), e o tamanho da requisição. Esta função consegue extrair

o tipo de transmissão a partir do pacote, realizando a operação de escrita do *buffer* no pacote no caso de ser uma operação de requisição de dados feita pelo driver, ou de cópia dos dados contidos no pacote para o *buffer* especificado no caso contrário.

5.6.3 Implementação do algoritmo TOTP no dispositivo

Nas transferências de dados entre dispositivo e driver, um dos valores importantes a serem transferidos é o código obtido pela execução do algoritmo TOTP.

Para tal, o grupo decidiu utilizar a biblioteca TOTP-MCU, desenvolvida por Netthaw, disponível em seu GitHub [21] e disponibilizada sob licença da MIT, permitindo que sejam realizadas as alterações necessárias para a execução do código durante o processo do dispositivo.

Esta biblioteca foi escolhida em particular por não precisar de nenhuma biblioteca externa, facilitando seu uso tanto dentro do dispositivo quanto no driver do kernel, cujo desenvolvimento será descrito nas seções seguintes.

Aqui, vale notar também que a função de criptografia utilizada nesta biblioteca é o SHA-1. Esta função de hash, utilizada em larga escala na internet há mais de duas décadas, não é mais considerada criptograficamente segura para aplicações que necessitem de algum grau razoável de resistência a colisões [22].

No entanto, aplicações que utilizam o SHA-1 como ferramenta criptográfica para a geração de um OTP, ou para a utilização de um Código de Autenticação de Mensagens (MAC), como o HMAC (e o TOTP, por consequência), as falhas identificadas na função em questão não são vistas como problemáticas [23].

Isso se dá principalmente porque a geração do código de uso único é realizado por meio de uma chave secreta. Sendo assim, colisões na função de *hash* utilizada pelo algoritmo HMAC não aumentam a chance de falsificação de um OTP [24], desde que a chave secreta não possa ser descoberta de maneira trivial por atacantes.

Ainda assim, é importante ressaltar que o uso de uma função mais segura, como o SHA-512, é encorajado para aplicações desse tipo, e poderia ser implementado junto ao TOTP em uma versão futura deste projeto.

Os códigos da biblioteca em questão foram adaptados para este projeto, utilizando-se as funções contidas nos arquivos `sha1.c` e `TOTP.c`, referentes à implementação da função de hash SHA-1 e à implementação do OTP em si.

Uma definição importante a ser feita quanto ao uso do TOTP é quanto à chave secreta utilizada na geração de códigos de uso único, que deve ser igual em ambos os lados da comunicação, de modo que os códigos gerados sejam compatíveis entre si.

Nos testes iniciais, foi utilizada uma chave pré-definida no código. Esta solução, no entanto, não serve para uma aplicação real. Portanto, foi necessário desenvolver um método que não depende de valores pré-definidos. O método escolhido pelo grupo pode ser resumido nos seguintes passos:

- Driver gera um número aleatório e o envia para o dispositivo;
- Dispositivo recebe o número aleatório, e gera um número aleatório próprio;
- Dispositivo concatena os dois números aleatórios gerados;
- Dispositivo gera o *hash* deste novo valor, e o utiliza de chave secreta para o TOTP;
- Dispositivo envia este *hash* de maneira criptografada para o driver também poder utilizá-lo como chave secreta.

A Figura 11 ilustra essa operação por meio de um diagrama de sequência.

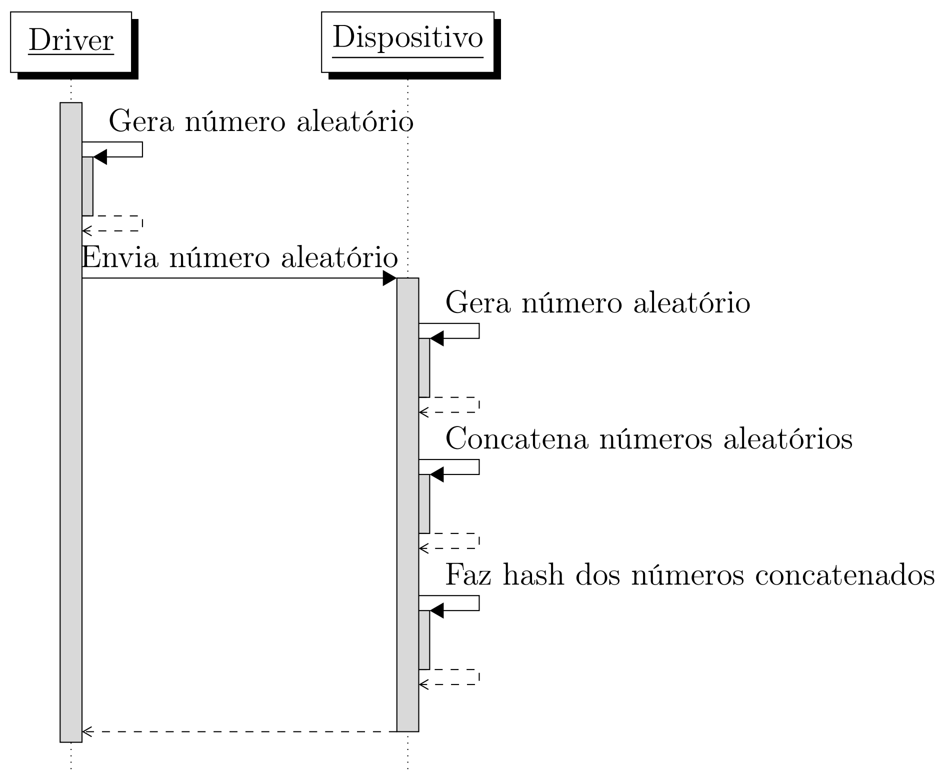


Figura 11: Diagrama de Sequência do fluxo de geração de chave TOTP.

Fonte: do autor.

Com isso, gera-se uma chave secreta que depende de ambos os lados da comunicação, garantindo um bom nível de segurança mesmo que alguma vulnerabilidade seja encontrada na geração de números aleatórios em algum deles.

A função de *hash* escolhida para esta etapa foi o SHA-384. Esta já está presente no código, sendo utilizada pela libspdm, e a API da biblioteca fornece uma função para realizar a operação de *hash* de maneira independente – a `spdm_hash_all`. Para utilizá-la, basta que a troca de mensagens SPDM já tenha atingido a etapa de `NEGOTIATE_ALGORITHMS`, como indicado no fluxo SPDM da Figura 2. A função pode ser chamada da seguinte maneira:

```
// Hash through SPDM
spdm_hash_all(
   ((spdm_context_t *)spdm_context)
         ->connection_info.algorithm.base_hash_algo, // SPDM
            base_hash_algo
   temp_key_array,    // data to be hashed
   TOTP_RANDOM_NUM_SIZE, // data size
   temp_totp_key      // buffer to receive the hash
);
```

Trecho de Código 5.13: Chamada da função de `hashing` do SPDM.

Neste exemplo, o hash é realizado sobre o vetor `temp_key_array`, com tamanho `TOTP_RANDOM_NUM_SIZE`, com o resultado sendo definido no *buffer* de saída, `temp_totp_key`. O primeiro termo da chamada informa para a biblioteca do SPDM o tipo de *hash* a ser utilizado no processamento, de acordo com o algoritmo descoberto nas etapas iniciais da comunicação SPDM.

Com isso tudo em mente, pode-se desenvolver os dois fluxos referentes ao TOTP do dispositivo USB, desenvolvidas dentro da função `spdm_device_get_response`, que é chamada como resposta para mensagens customizadas oriundas do driver.

Esta função é organizada de tal maneira que, dependendo do tamanho da mensagem recebida pelo dispositivo, ações diferentes são tomadas. Em particular, com requisições do tamanho do número aleatório definido pelo driver, a função entra no fluxo de definição de chave para a geração de códigos TOTP. Caso a mensagem seja menor, entende-se que a requisição é de verificação periódica de códigos TOTP. A estrutura da função é indicada pelo código abaixo.

```
static return_status spdm_device_get_response(
          IN void *spdm_context,
          IN uint32 *session_id,
          IN boolean is_app_message,
          IN uintn request_size,
          IN void *request,
          IN OUT uintn *response_size,
          OUT void *response) {
   // Check the request size
   // If zero, this is a periodic TOTP check
   // Also check if a TOTP key has been set previously
   if (request_size <= TOTP_RANDOM_NUM_SIZE &&
      !arr_is_zero(totp_key, TOTP_RANDOM_NUM_SIZE)){
      // [...]
   }

   // Otherwise, this is a TOTP key defining message
   else {
      // [...]
   return RETURN_SUCCESS;
}
```

Trecho de Código 5.14: Definição da função `spdm_device_get_response`.

5.6.3.1 Fluxo de geração de novas chaves TOTP

Primeiramente, o fluxo de geração de uma nova chave deve ser iniciado pelo driver com o envio de um número aleatório, seguindo o processo descrito anteriormente.

Do lado do dispositivo, a geração de um número aleatório foi realizada com o auxílio de uma biblioteca externa, denominada CSPRNG, desenvolvida por Duthomhas e disponível em seu GitHub [25]. Esta biblioteca facilita o uso de um gerador de números aleatórios do sistema; dessa maneira, o QEMU consegue extrair números pseudo-aleatórios criptograficamente seguros utilizando, no caso deste projeto, o `/dev/urandom`, extraindo os valores aleatórios da própria máquina *host* na execução do QEMU.

```
// Generate a random number locally
do{
```

```
    rng = csprng_create();
} while (!rng);
csprng_get (rng, &random_local_num, sizeof(random_local_num));
rng = csprng_destroy(rng);
```

Trecho de Código 5.15: Função de geração de número pseudo-aleatório.

Feita a geração desse número, o *hash* é realizado por meio da função `spdm_hash_all`, e o valor gerado é salvo em uma variável global do dispositivo, sendo, então, enviada ao driver.

```
// Fetch the original value as a uint64_t
uint64_t temp = *((uint64_t*)(request + 1));

// Concatenate both random numbers and hash them
memcpy(temp_key_array, &random_local_num, TOTP_RANDOM_NUM_SIZE);
memcpy(temp_key_array + TOTP_RANDOM_NUM_SIZE, &temp,
    TOTP_RANDOM_NUM_SIZE);

// Hash through SPDM
spdm_hash_all(
    ((spdm_context_t *)spdm_context)
          ->connection_info.algorithm.base_hash_algo, // SPDM
             base_hash_algo
    temp_key_array,    // data to be hashed
    TOTP_RANDOM_NUM_SIZE, // data size
    temp_totp_key      // buffer to receive the hash
);

// Saving generated key on global array
memcpy(totp_key, &temp_totp_key, SPDM_SHA_SIZE);
```

Trecho de Código 5.16: Concatenação das variáveis, seguido do hash desse valor.

Para realizar o envio, o primeiro byte do *buffer* é definido com a constante `MCTP_MESSAGE_TYPE_VENDOR_DEFINED_IANA`, que auxilia a libspdm no processo de identificar o envio e recebimento das mensagens customizadas criptografadas necessárias para esta geração e troca de chaves, e a função libera os dados para que o driver possa extraí-los e definir a nova chave do TOTP.

```
// Set first byte as this flag
*((uint8_t*)response) = MCTP_MESSAGE_TYPE_VENDOR_DEFINED_IANA;

// copy TOTP key to response after the first byte
memcpy(response + 1, &temp_totp_key, SPDM_SHA_SIZE);
*response_size = SPDM_SHA_SIZE + 1;
```

Trecho de Código 5.17: Definição do começo do *buffer*.

5.6.3.2 Fluxo de envio de código TOTP

Por outro lado, o outro fluxo, referente ao envio do código TOTP gerado localmente, também é iniciado com o envio de uma mensagem do driver. O processamento do dispositivo em si, no entanto, é relativamente simples: após receber uma mensagem de tamanho menor que o número aleatório envolvido no fluxo descrito anteriormente, a biblioteca do TOTP é inicializada com a última chave secreta definida, e um código é gerado com a *timestamp* atual.

```
/*
* Gets a TOTP code with the current timestamp.
*/
static uint32_t get_totp(void){
   TOTP(totp_key, SPDM_SHA_SIZE, 60); // key, key size, timestep in s

   return getCodeFromTimestamp((unsigned)time(NULL));
}
```

Trecho de Código 5.18: Chamada da função para obter o código TOTP com a *timestamp* atual.

A função `get_totp` expressa acima é chamada de dentro da própria função `spdm_device_get_response`, atribuindo o valor recebido para a variável `totp_code`.

Então, para facilitar o processo de codificação e decodificação do TOTP gerado, é utilizada a função `sprintf`, que codifica o código em hexadecimal, como no seguinte exemplo:

```
sprintf(totp_str, "%x", totp_code);
```

Trecho de Código 5.19: Exemplo de chamada da função `sprintf`.

Este valor será decodificado no driver por meio de uma função que converte o valor hexadecimal em decimal.

Neste fluxo, a constante `MCTP_MESSAGE_TYPE_VENDOR_DEFINED_IANA` também é adicionada ao *buffer* de saída.

```
// Set first byte as this flag
*((uint8_t*)response) = MCTP_MESSAGE_TYPE_VENDOR_DEFINED_IANA;

// Copy TOTP value to response after the first byte
memcpy(response + 1, &totp_str, TOTP_HEX_SIZE*sizeof(char));
```

Trecho de Código 5.20: Preparação do *buffer* de saída.

Por fim, o driver realizará uma verificação do valor gerado, podendo causar o desligamento da máquina caso inconsistências sejam encontradas.

5.6.4 Implementação do protocolo SPDM no dispositivo

A implementação do protocolo SPDM no dispositivo USB por meio da biblioteca libspdm foi, em partes, facilitada pelo uso da versão do QEMU já alterada previamente pelos professores do LARC, como citado anteriormente. Com esta versão, o processo de adição das funções da libspdm não foi muito complexo, necessitando apenas da alteração do Makefile do dispositivo para adicionar os *headers* da biblioteca e da inclusão destes arquivos no início do código do dispositivo em si.

Com isso, a maior dificuldade desta atividade passou a ser a sincronização das diferentes funções chamadas tanto pela biblioteca libspdm, aguardando mensagens enviadas pelo driver e preparando suas respostas, quanto pelas funções chamadas pelo QEMU em si, tratando, por exemplo, da transmissão e união de pacotes USB recebidos e enviados pelo dispositivo.

5.6.4.1 Configurações iniciais do contexto SPDM

Antes de qualquer ação mais específica, deve-se inicializar tanto as variáveis da libspdm, quanto a própria conexão SPDM com o driver.

Levando-se em consideração que o driver inicializará todas as comunicações, o dispositivo deverá agir como Responder na comunicação, possuindo, assim, informações como os algoritmos que ele disponibiliza, versão do SPDM utilizada, entre outras. Este processo ocorre com a definição de variáveis intrínsecas à variável de contexto do SPDM, além do uso da função `spdm_set_data` como maneira de definir estes valores a partir de constantes definidas previamente. Por exemplo:

```
static uint32_t totp_spdm_m_support_hash_algo =
   SPDM_ALGORITHMS_BASE_HASH_ALGO_TPM_ALG_SHA_384 |
   SPDM_ALGORITHMS_BASE_HASH_ALGO_TPM_ALG_SHA_256;

// [...]

data32 = totp_spdm_m_support_hash_algo;
spdm_set_data(s->spdm_context, SPDM_DATA_BASE_HASH_ALGO, &parameter,
       &data32, sizeof(data32));
```

Trecho de Código 5.21: Função que negocia os tipos de algoritmos suportados.

Antes disso, no entanto, deve-se inicializar o contexto SPDM:

```
// Initialize context variable
spdm_init_context(s->spdm_context);
```

Trecho de Código 5.22: Chamada da função que inicializa o contexto do SPDM.

Nesta etapa, também são registradas as funções que se comunicam diretamente com a biblioteca da seguinte maneira:

```
spdm_register_device_io_func(
      s->spdm_context,
      spdm_device_send_message,
      spdm_device_receive_message);
```

Trecho de Código 5.23: Registro das funções de envio e recebimento de mensagens.

Com isso, as funções `spdm_device_send_message` e `spdm_device_receive_message` tornam-se os pontos de contato entre os dados do dispositivo e o processamento da biblioteca do SPDM.

Outras funções que são definidas incluem as funções já inclusas na própria

libspdm para codificação e decodificação de mensagens, a definição de uma função `spdm_device_get_response`, que é ativada no recebimento de funções customizadas do SPDM, e a criação de uma *thread* do SPDM no dispositivo, que continuamente busca requisições de saída ou chegada de dados.

```
// Set functions for encoding and decoding SPDM messages
spdm_register_transport_layer_func(
      s->spdm_context,
      spdm_transport_mctp_encode_message,
      spdm_transport_mctp_decode_message);

// Set callback function for custom SPDM messages
spdm_register_get_response_func(
      s->spdm_context,
      spdm_device_get_response);

// Set QEMU thread function
qemu_thread_create(&s->spdm_io_thread,
      "spdm_io_usb_device",
      usb_totp_spdm_io_thread,
      s,
      QEMU_THREAD_JOINABLE);
```

Trecho de Código 5.24: Definição de diversas funções usados pelo SPDM.

A função responsável pela execução do *thread* do QEMU é composta por um `while (true)`, dentro do qual espera-se que o envio de dados seja liberado por outras funções, que serão vistas em detalhe a seguir. Este processo ocorre por meio do uso de um *mutex*, pela verificação da variável global `spdm_receive_is_ready`, e a espera desta variável ter o valor desejado.

```
/*
* QEMU thread function. Will execute continually and look for
   incoming
* or outgoing SPDM requests.
*/
static void *usb_totp_spdm_io_thread(void *opaque)
{
   USBTotpSpdmState *s = opaque;
```

```
    return_status status;

    while (true) {
        qemu_mutex_lock(&spdm_io_mutex);
        if (!spdm_receive_is_ready) {
            qemu_cond_wait(&spdm_io_cond, &spdm_io_mutex);
        }
        spdm_receive_is_ready = 0;

        qemu_mutex_unlock(&spdm_io_mutex);

        // [...]
    }

    return NULL;
}
```

Trecho de Código 5.25: Função de execução da *thread*.

Por fim, este código chama a função `spdm_responder_dispatch_message`, que avalia o conteúdo da mensagem, e chama a função de callback quando a mensagem for recebida ou enviada com sucesso.

```
    while (true) {
        // [...]

        status = spdm_responder_dispatch_message (s->spdm_context);

        if (status == RETURN_SUCCESS) {
            // Load certificates
            usb_totp_spdm_server_callback (s->spdm_context);
        } else {
            printf("SpdmResponderDispatchMessage error: %llX\n",
                status);
        }
    }
```

Trecho de Código 5.26: Avaliação da mensagem.

Esta função de callback, em particular, define respostas relacionadas ao recebimento

de certificados, processo que deve ser feito do lado do driver para garantir a autenticidade da comunicação.

De maneira similar à definição de dados essenciais da comunicação na função de inicialização, a função `spdm_set_data` é utilizada como maneira de definir valores e informações relevantes a partir de constantes definidas previamente. Estes incluem informações sobre os certificados enviados pelo driver, além de tipos de autenticação que podem ser utilizados durante a comunicação, por exemplo.

Além disso, é utilizada também uma função de leitura de certificado já disponível na própria biblioteca.

```
Res = read_responder_public_certificate_chain (m_use_hash_algo,
   m_use_asym_algo, &Data, &DataSize, NULL, NULL);
if (Res) {
   zero_mem (&Parameter, sizeof(Parameter));
   Parameter.location = SPDM_DATA_LOCATION_LOCAL;
   Data8 = m_use_slot_count;
   spdm_set_data (SpdmContext, SPDM_DATA_LOCAL_SLOT_COUNT,
      &Parameter, &Data8, sizeof(Data8));

   for (Index = 0; Index < m_use_slot_count; Index++) {
      Parameter.additional_data[0] = Index;
      spdm_set_data (SpdmContext, SPDM_DATA_LOCAL_PUBLIC_CERT_CHAIN,
         &Parameter, Data, DataSize);
   }
}
```

Trecho de Código 5.27: Função de leitura de certificado.

Logo depois, constantes que sinalizam a presença do certificado são adicionadas ao contexto SPDM.

Estas funções vistas até agora, no entanto, não manipulam os dados diretamente, tratando apenas de definir o contexto SPDM apropriadamente para o processo de recebimento de dados.

5.6.4.2 Recepção de mensagens SPDM

Como visto anteriormente, todos os dados recebidos ou enviados pelo dispositivo emulado pelo QEMU são passados pela função `usb_totp_spdm_handle_data` (equivalente à função `usb_serial_handle_data` do dispositivo serial). No entanto, os dados devem ser também repassados para as funções de leitura e envio de dados da própria libspdm, `spdm_device_send_message` e `spdm_device_receive_message`, definidas anteriormente.

Primeiramente, foram definidas duas variáveis globais, `spdm_buf` e `spdm_buf_size`, que guardam tanto o vetor de chegada quanto de recebimento.

Também vale ressaltar que as transferências USB realizadas entre este dispositivo e o driver têm tamanho máximo de 64 bytes; sendo assim, as funções de transferência precisaram ser adaptadas para receber e enviar pacotes grandes fragmentados. Em particular, tanto no driver quanto no dispositivo, as funções de envio adicionam 4 bytes de cabeçalho ao começo do pacote, indicando o tamanho total da transferência.

No caso do dispositivo estar recebendo dados do driver, o QEMU inicializa a função `usb_totp_spdm_handle_data` com a flag `USB_TOKEN_OUT`. Inicialmente, trava-se um *mutex* específico do QEMU, que fora inicializado em outra seção do código, que apenas será liberado no fim da execução da função.

Em seguida, o código diferencia as transferências de acordo com a ordem dos pacotes que chegaram.

Durante o processamento do primeiro pacote da transferência, o código sempre extrai os primeiros 4 bytes do pacote, definindo este valor como o tamanho total da transferência. A execução apenas para quando a quantidade de bytes lidos for igual a este tamanho.

```
// First transfer
if (s->full_transfer_size_out == 0){
   first = true;

   // Copy size data to size array
   usb_packet_copy(p, size_out, LEN_HEX_SIZE);

   // Get actual size
   s->full_transfer_size_out = get_size_from_request(size_out);
   spdm_buf_size = s->full_transfer_size_out;
   // [...]
```

```
}
```

Trecho de Código 5.28: Processamento do cabeçalho do primeiro pacote da transferência.

A função `get_size_from_request`, de maneira simplificada, extrai o valor contido no cabeçalho, que havia sido encodado em hexadecimal, por meio da função `hexdec`, que converte o valor extraído em decimal. Esta utiliza uma tabela de valores para realizar a conversão de maneira rápida e computacionalmente eficiente.

```
static long hexdec(unsigned const char *hex) {
    long ret = 0;
    while (*hex && ret >= 0) {
        ret = (ret << 4) | hextable[*hex++];
    }
    return ret;
}

/*
* Fetches SPDM request size from specific byte from the request
    itself
*/
static size_t get_size_from_request(void* data) {
    uint8_t size_hex[LEN_HEX_SIZE];
    unsigned long size_dec;

    memcpy(size_hex, data, LEN_HEX_SIZE*sizeof(uint8_t));
    size_dec = hexdec(size_hex);
    return size_dec;
}
```

Trecho de Código 5.29: Funções `get_size_from_request` e `hexdec` .

Também é realizada uma verificação do tamanho máximo do pacote, definindo se a transferência precisará de mais pacotes.

```
// Length for next usb_packet_copy will be either the full
// remaining buffer (single packet transfer), or
// MAX_PACKET_SIZE - LEN_HEX_SIZE (multiple packet transfer)
if (s->full_transfer_size_out <= MAX_PACKET_SIZE - LEN_HEX_SIZE){
    // Single packet transfer
```

```
    last = true;
    len = s->full_transfer_size_out;
}
else {
    // First packet of multiple packet transfer
    last = false;
    len = MAX_PACKET_SIZE - LEN_HEX_SIZE;
}
```

Trecho de Código 5.30: Verificação do tamanho do pacote.

Então, no caso de ser um outro pacote dentro de uma transferência maior, esta verificação é ignorada, definindo apenas o tamanho real da leitura atual na variável `len`.

```
// First transfer
if (s->full_transfer_size_out == 0){
    // [...]
}
// Not the first transfer
else {
    first = false;

    // Last packet in multiple packet transfer
    if (s->current_transfer_size_out + MAX_PACKET_SIZE >=
        s->full_transfer_size_out){
        last = true;
        len = s->full_transfer_size_out - s->current_transfer_size_out;
    }
    else {
        last = false;
        len = MAX_PACKET_SIZE;
    }
}
```

Trecho de Código 5.31: Definição do tamanho real da leitura.

Em seguida, o pacote em si é copiado para a variável `spdm_buf` por meio da função `usb_packet_copy`.

```
// Copy packet data to spdm_buf with current_transfer_size_out as
```

```
    offset
usb_packet_copy(p, spdm_buf + s->current_transfer_size_out, len);
```

Trecho de Código 5.32: Chamada da função usb_packet_copy.

Por fim, finaliza-se a execução desta função. Se o pacote atual é o último, as variáveis referentes ao tamanho da transferência são zeradas. Além disso, a variável global spdm_receive_is_ready é definida como 1, sinalizando que a operação de recebimento de dados acabou, e a função da libspdm de recebimento de dados já pode executar. Este processo é simbolizado também pela ativação do condicional do QEMU, qemu_cond_signal(&spdm_io_cond);.

No caso em que ainda há pacotes nesta mesma transferência, o tamanho atual é simplesmente ajustado. Se ainda houver dados a serem recebidos, o próprio QEMU chamará a função novamente, permitindo que a execução continue normalmente até não haver mais dados disponíveis.

Neste momento, o *mutex* do QEMU também é liberado.

```
// Last transfer
if (last){
    // Reset all variables specific to this transfer
    s->current_transfer_size_out = 0;
    s->full_transfer_size_out = 0;

    // Signal SPDM that the message is ready
    spdm_receive_is_ready = 1;
    qemu_cond_signal(&spdm_io_cond);
}
// Not the last transfer
else {
    // Add the size that was written to spdm_buf
    // to current_transfer_size_out
    s->current_transfer_size_out += len;
}

qemu_mutex_unlock(&spdm_io_mutex);
break;
```

Trecho de Código 5.33: Final da função de transferência.

Terminada a execução da função `usb_totp_spdm_handle_data`, a função `spdm_device_receive_message` é chamada. Esta extrai o *buffer* salvo em `spdm_buf`, salvando-o no vetor de saída, `*response`.

```
static return_status spdm_device_receive_message(
        IN void *spdm_context,
        IN OUT uintn *response_size,
        IN OUT void *response,
        IN uint64 timeout){
    // [...]

    qemu_mutex_lock(&spdm_io_mutex);
    *response_size = spdm_buf_size;
    memcpy(response, spdm_buf, *response_size);
    qemu_mutex_unlock(&spdm_io_mutex);

    // [...]
    return RETURN_SUCCESS;
}
```

Trecho de Código 5.34: Chamada da função que realiza o recebimento do dados.

Feito este processo todo, os dados recebidos são enviados com sucesso para a libspdm, na qual os dados serão processados, e a resposta, enviada de volta ao driver.

5.6.4.3 Transmissão de mensagens SPDM

Para enviar dados, duas ações acontecem de maneira quase simultânea: o driver realiza uma requisição de recebimento de dados, especificando o tamanho máximo da transferência, e a libspdm, depois de finalizar o processamento dos dados e a geração da resposta SPDM, chama a função `spdm_device_send_message`.

Os dois processos devem seguir uma ordem específica; no caso, a função `spdm_device_send_message` deve ser realizada antes, definindo o vetor `spdm_buf` com o conteúdo correto a ser enviado de volta para o driver.

Então, a função `usb_totp_spdm_handle_data` será chamada, esperando o fim da execução da função anterior, para então enviar os dados para o driver. Este processo, assim como o de recebimento de dados, necessita da codificação de um cabeçalho de 4 bytes com o tamanho total da transferência. Com isso, o driver poderá realizar o processamento

necessário dos dados por ele recebidos.

De maneira mais específica, a função `spdm_device_send_message`, de maneira similar à de recebimento de dados do SPDM, precisa travar o *mutex* para não correr o risco de ocasionar problemas com condições de corrida. Então, os conteúdos da requisição, contidas em `*request` e `request_size`, são passados para as variáveis `spdm_buf` e `spdm_buf_size`.

Por fim, a condicional do QEMU `spdm_io_cond` é ativada por meio da função `qemu_cond_signal`, sinalizando que o processamento dos pacotes por meio da função `usb_totp_spdm_handle_data` já pode ser realizado. A variável global `spdm_send_is_ready` também é ativada com esse mesmo propósito.

```
/*
* Send SPDM messages
*/
static return_status spdm_device_send_message(
         IN void *spdm_context,
         IN uintn request_size,
         IN void *request,
         IN uint64 timeout) {
   // [...]

   qemu_mutex_lock(&spdm_io_mutex);
   spdm_buf_size = request_size;
   memcpy(spdm_buf, request, request_size);
   spdm_send_is_ready = 1;
   qemu_cond_signal(&spdm_io_cond);
   qemu_mutex_unlock(&spdm_io_mutex);

   return RETURN_SUCCESS;
}
```

Trecho de Código 5.35: Função `spdm_device_send_message`.

Então, entra-se na função `usb_totp_spdm_handle_data`, no caso `USB_TOKEN_IN`. Antes que qualquer coisa, o *mutex* é trancado, e o código aguarda a execução da função anterior.

```
qemu_mutex_lock(&spdm_io_mutex);
```

```
if (!spdm_send_is_ready) {
   qemu_cond_wait(&spdm_io_cond, &spdm_io_mutex);
}
```

Trecho de Código 5.36: Travamento do *mutex* e espera do término da função anteriror.

Em seguida, a função inicia o processamento dos dados e a conversão destes em pacotes. Inicialmente, o código lida com o caso em que o pacote atual é o primeiro da transferência atual. Neste caso, deve-se encodar o tamanho com a função `sprintf`, assim como foi realizado no envio do TOTP. O cabeçalho é copiado para o pacote por meio da função `usb_packet_copy`.

```
// First transfer
if (s->current_transfer_size_in == 0){
   first = true;

   // add length to first positions of spdm_buf
   char len_str[LEN_HEX_SIZE];
   sprintf(len_str, "%x", spdm_buf_size);

   // If it's 64 bytes, it breaks, so do this
   int headersize = (spdm_buf_size == 60) ? (LEN_HEX_SIZE -
      1)*sizeof(char) : LEN_HEX_SIZE*sizeof(char);
   memcpy(&(new_header)[0], len_str, headersize);

   // Copy header to packet
   usb_packet_copy(p, new_header, headersize);
}
```

Trecho de Código 5.37: Tratamento do pacote atual como o primeiro da transferência.

Nota-se, em particular, que há um caso especial quando o tamanho da transferência é de exatamente 60 bytes. Ao se adicionar os 4 bytes de cabeçalho, o pacote atinge 64 bytes, que é o tamanho máximo exato de um único pacote da transferência USB por URBs. O problema surge dentro do QEMU: ao ver um pacote com o tamanho máximo, o aplicativo assume que haverá mais pacotes a serem enviados, quando isso não é verdade, e o grupo não encontrou uma maneira razoável de informar o QEMU que a transferência chegou a seu fim sem alterar arquivos específicos do próprio programa, o que está fora do escopo da aplicação desenvolvida neste projeto.

Com isso em mente, a decisão tomada pelo grupo foi de criar um cabeçalho específico para o caso em que a mensagem a ser enviada possui exatamente 60 bytes. Neste caso, o cabeçalho possui apenas 3 bytes, fazendo com que a transferência tenha um total de 63 bytes.

Isso gerou também uma necessidade de mudança do lado do driver, que precisou aceitar também este caso especial no qual o cabeçalho é reduzido.

Também é importante notar que esta adição abre a possibilidade de problemas no caso em que o tamanho do cabeçalho é de exatamente 960 bytes, já que seus valores em hexadecimal podem ser facilmente confundidos ($60_{10} = 0x3C_{16}$, enquanto que $960_{10} = 0x3C0_{16}$). No entanto, depois de centenas de testes, os dados enviados para o driver provaram ter tamanhos inferiores a 500 bytes ou superiores a 1200 bytes, levando o grupo à decisão de que este problema não possui alta prioridade.

Toda vez que o cabeçalho for adicionado, 4 bytes dos que seriam enviados ao driver seriam normalmente perdidos nesta etapa. Para evitar tal problema, estes bytes são salvos em um vetor, `s->extra_buffer_in`. A criação deste vetor se dará sempre que um pacote não for o último de uma transferência fragmentada. No entanto, sempre que o pacote não for o primeiro, este valor será acessado, para que o *buffer* de saída tenha todos os dados necessários.

Além disso, também são realizadas verificações para extrair o tamanho total deste vetor de dados faltantes: seu valor será igual a `extra_header_len` sempre que tiver o tamanho completo, mas se faltar um valor menor de bytes, este valor estará expresso na equação `spdm_buf_size - s->current_transfer_size_in`. Após estes processos, os dados são copiados ao pacote por meio da função `usb_packet_copy`.

```
// First transfer
if (s->current_transfer_size_in == 0){
   // [...]
}
// Not the first transfer
else {
   first = false;

   int extra_header_len = (spdm_buf_size == 60) ? (SPDM_SEND_OFFSET
      - 1) : SPDM_SEND_OFFSET;
   extra_len = MIN(spdm_buf_size - s->current_transfer_size_in,
      extra_header_len);
```

```
    // Copy spare data from extra_buffer_in
    usb_packet_copy(p, s->extra_buffer_in, extra_len);
    s->current_transfer_size_in += extra_len;
}
```

Trecho de Código 5.38: Tratamento do pacote no caso de não ser o primeiro da transferência.

Em seguida, o conteúdo em si deve ser copiado ao pacote. Seu tamanho é definido ao levar em consideração o tamanho máximo do pacote, `p->iov.size`, e o tamanho do cabeçalho, `SPDM_SEND_OFFSET`. Os dados são novamente copiados com `usb_packet_copy`, e o tamanho da transferência atual é atualizado.

```
// len is the minimum value between the remainig data to be
    transfered
// (spdm_buf_size - s->current_transfer_size_in) and the
// max size subtracted by the original header, SPDM_SEND_OFFSET
int iov_with_header = (spdm_buf_size == 60) ? (p->iov.size -
    (SPDM_SEND_OFFSET - 1)) : p->iov.size - SPDM_SEND_OFFSET;
len = MIN(spdm_buf_size - s->current_transfer_size_in,
    iov_with_header);

// Copy spdm_buf to packet, right after new_header
usb_packet_copy(p, spdm_buf + s->current_transfer_size_in, len);
s->current_transfer_size_in += len;
```

Trecho de Código 5.39: Definição do tamanho do pacote e cópia da mensagem.

No caso do pacote atual não ser o último da transferência, os bytes que sobraram após a inserção do cabeçalho são salvos para serem utilizados na iteração seguinte, como citado anteriormente. Esta operação também leva em consideração os valores `spdm_buf_size`, `s->current_transfer_size_in` e o tamanho do cabeçalho.

```
// Not the last transfer
if (s->current_transfer_size_in < spdm_buf_size) {
    last = false;

    // Copy spare data to extra_buffer_in
    memcpy(s->extra_buffer_in,
```

```
        spdm_buf + s->current_transfer_size_in,
        MIN(spdm_buf_size - s->current_transfer_size_in,
            SPDM_SEND_OFFSET));
}
```

Trecho de Código 5.40: Bytes restantes são salvos para a próxima iteração.

Por fim, se o pacote atual é o último, adiciona-se os bytes restantes que haviam sido salvos (no caso em que o pacote também não é o primeiro), e as variáveis globais relevantes são redefinidas como 0. Em particular, `spdm_send_is_ready` ser definida como 0 prepara a função para a próxima transferência, na qual `spdm_device_send_message` deverá definir o valor desta variável como 1 novamente para continuar a execução. O *mutex* que havia sido trancado no início da execução também é liberado.

```
// Not the last transfer
if (s->current_transfer_size_in < spdm_buf_size) {
    // [...]
}
// Final transfer
else {
    last = true;
    // If it is the last transfer of a combined one,
    // add back the original header size
    if (!first){
        usb_packet_copy(p, spdm_buf + s->current_transfer_size_in -
            SPDM_SEND_OFFSET, SPDM_SEND_OFFSET);
    }

    s->current_transfer_size_in = 0;
    spdm_send_is_ready = 0;
}

qemu_mutex_unlock(&spdm_io_mutex);
```

Trecho de Código 5.41: Tratamento do pacote para o caso de ser o último da transferência.

Feito todo este processo, finaliza-se o código necessário para a transferência de dados do lado do dispositivo.

5.7 Desenvolvimento do driver do dispositivo

De maneira similar ao dispositivo emulado pelo QEMU, o driver desenvolvido teve como base drivers USB já existentes no kernel do Linux, incluindo o driver esqueleto presente nos arquivos do kernel como `usb-skeleton.c`.

Antes de qualquer funcionalidade específica, é importante definir as funções necessárias para a inicialização do driver em si.

Para tal, é importante primeiramente definir um *struct* do tipo `usb_driver`, que contém, por exemplo, a definição do nome do driver e chamadas às funções de *probe* (chamada quando um dispositivo relacionado ao driver é encontrado) e *disconnect* (chamada quando o dispositivo é desconectado).

Aqui também é definida uma variável para a tabela de identificadores de dispositivos, que define um conjunto de IDs de vendedor e de produto que, quando inseridos no sistema, são relacionados a este driver.

```
static struct usb_driver usb_totp_spdm_driver = {
   .name     = "TOTP + SPDM USB Driver",
   .probe    = usb_totp_spdm_probe,
   .disconnect = usb_totp_spdm_disconnect,
   .id_table = usb_totp_spdm_table,
};
```

Trecho de Código 5.42: Struct para a definição do driver.

Em seguida, vale notar as chamadas às funções `module_init` e `module_exit`, que definem quais funções realizam as operações de inicialização e remoção do driver, respectivamente. Os nomes destas funções para o driver deste projeto são `usb_totp_spdm_init` e `usb_totp_spdm_exit`.

Dentro destas, é essencial realizar as operações de registro do driver, `usb_register`, e de cancelamento, `usb_deregister`. Estas funções permitem a identificação e inserção do driver pelo sistema.

Além destas funções, é importante também notar a função de *probe* do driver, que neste projeto foi denominada de `usb_totp_spdm_probe`. Esta função é chamada pelo sistema toda vez que um dispositivo USB contido na tabela de identificadores de dispositivos é inserido no sistema, e realiza operações relevantes a este momento inicial de ativação e inicialização de um dispositivo.

Nesta etapa do funcionamento do driver, pode-se também analisar os *endpoints* do dispositivo USB. Um *endpoint* representa um caminho de saída ou entrada de dados no dispositivo, sendo que cada dispositivo pode ter dois ou mais destes. O *endpoint* apropriado deve ser especificado em qualquer operação de transmissão de dados entre driver e dispositivo USB.

Por fim, vale lembrar também que, assim como no caso do dispositivo USB, o driver também deve possuir um `struct` com todas as informações e estruturas de dados relevantes para os processamentos a serem feitos. Em particular, é importante conter um campo para um `struct usb_device`. A definição deste dispositivo pode ser realizado na função de *probe*, da seguinte maneira:

```
static int usb_totp_spdm_probe(
      struct usb_interface *interface,
      const struct usb_device_id *id)
{
   // [...]
   totp_spdm_usb_struct->dev = interface_to_usbdev(interface);
   // [...]
}
```

Trecho de Código 5.43: Definição do dispositivo na função *probe*.

Com isso tudo definido, já é possível inicializar o driver na máquina virtual, dado que a inserção do módulo com o driver foi realizada com sucesso, como visto na Seção 5.4.

Neste estágio do desenvolvimento, é possível começar a criar as funções específicas do driver, de tal maneira que os requisitos definidos pelo projeto sejam atendidos.

5.7.1 Comunicação entre driver e dispositivo USB

Para que dados possam ser trocados livremente entre o driver e o dispositivo, era fundamental que o grupo compreendesse o funcionamento interno das transferências USB. Estas se dão por meio de URBs, ou USB Request Blocks – blocos de dados que carregam não apenas as informações a serem transferidas, mas também o tipo de transferência, *pipe* utilizado e *endpoint* de destino, por exemplo.

Primeiramente, é importante saber que todas as transferências são iniciadas pelo driver. Assim sendo, independentemente da requisição ser de envio ou de recebimento de dados, ela deverá ser feita por ele, sendo, então, respondida pelo dispositivo.

Uma típica transferência de dados possui a seguinte aparência, como visto em [26]:

```
usb_fill_bulk_urb(
      skel->write_urb,
      skel->dev,
      usb_sndbulkpipe(skel->dev, skel->bulk_out_endpointAddr),
      skel->write_urb->transfer_buffer,
      bytes_written,
      skel_write_bulk_callback,
      skel);
```

Trecho de Código 5.44: Exemplo de uma tranferência de dados.

Nessa transferência, vale ressaltar a especificação das seguintes variáveis:

`skel->write_urb` é um ponteiro para o bloco de requisição como um todo, que deve ser inicializado de antemão com uma chamada à função `usb_alloc_urb`;

`skel->dev` especifica o dispositivo USB, apontando para a estrutura específica que deve estar contida no *struct* global da classe;

`usb_sndbulkpipe` especificando-se o dispositivo e o *endpoint* de destino, cria um *pipe* (em outras palavras, uma conexão configurada entre o driver e o *endpoint* do dispositivo), podendo ser de envio (`usb_sndbulkpipe`) ou de recebimento de dados (`usb_rvcbulkpipe`);

`skel->write_urb->transfer_buffer` é o *buffer* em si, que será transportado para o dispositivo;

`bytes_written` é tamanho do *buffer* especificado anteriormente;

`skel_write_bulk_callback` especifica uma função que será chamada após o envio do URB;

`skel` é uma referência à estrutura de dados do driver, utilizada como contexto do URB.

Feito o preenchimento do URB, este deve ser enviado por meio de uma chamada à função `usb_submit_urb`.

Em particular, vale notar também que o fluxo de chamada da função de *callback* não é síncrono. Dessa forma, é fundamental que haja uma maneira de controlar a execução

destas funções (por exemplo, por um `mutex`), de tal maneira que a execução das funções seguintes necessite da confirmação da finalização dos passos anteriores.

No caso das funções do driver desenvolvido, a principal condição que precisou ser verificada foi a conclusão do processo de envio (ou recebimento) de dados, que ocorre assim que a chamada da função de *callback* é realizada. Para isso, foi utilizada a estrutura de `completion`, disponível no kernel do Linux.

Tanto na função de envio quanto na de recebimento de dados, o *struct* em questão é inicializado:

```
// initializing completion struct
init_completion(&totp_spdm_usb_struct->spdm_response_done);
```

Trecho de Código 5.45: Estrutura `completion`.

Então, depois do envio do URB em si, o código aguarda pela conclusão do processo de envio:

```
// Code must (not) continue until the response is completed.
// Sending and receiving URBs is not a synchronous process, so
// this is necessary to make sure we have the right data here.
wait_for_completion(&totp_spdm_usb_struct->spdm_response_done);
```

Trecho de Código 5.46: Espera pelo `completion`.

Dentro da função de *callback*, depois de se garantir que os dados foram recebidos ou enviados com sucesso, pode-se marcar esta estrutura de dados como completa:

```
if (!completion_done (&totp_spdm_usb_struct->spdm_response_done)){
    complete (&totp_spdm_usb_struct->spdm_response_done);
}
```

Trecho de Código 5.47: Marcação da estrutura como completa.

Isso faz com que a posição de memória alocada pelo URB em si possa ser limpada com a função `usb_free_urb`, e que o código que utiliza os dados possa fazê-lo sem maiores preocupações.

Como testes iniciais, o grupo criou transferências simples de vetores de dados, que permitiram a certeza do funcionamento adequado das funções relacionadas à submissão e recebimento de URBs, o que facilitou o processo de transferência de dados com a libspdm.

5.7.2 Criação de rotina periódica constante no driver

Como especificado previamente, este projeto garante verificações periódicas utilizando o protocolo SPDM e o algoritmo TOTP. Para tal, é de suma importância que o driver possua um método simples e eficiente de executar uma tarefa de maneira contínua.

Com este propósito, o grupo utilizou *workqueues* – uma estrutura do kernel do Linux capaz de ser executada em um único *kernel thread*, de maneira assíncrona de outras funções do próprio driver no qual ela está inserida. Dentro desta *workqueue*, as funções relevantes do SPDM e do TOTP serão chamadas, conferindo uma inicialização rápida da comunicação SPDM e trocas de códigos TOTP eficientes.

Para utilizar a *workqueue* no driver, esta deve ser inicializada com as seguintes definições globais:

```
static void totp_spdm_work_handler(struct work_struct *w);
static struct workqueue_struct *wq = 0;
static DECLARE_WORK(totp_spdm_work, totp_spdm_work_handler);
```

Trecho de Código 5.48: Definições globais da *workqueue*.

Então, a *workqueue* em si deve ser inicializada. Decidiu-se que este processo seria realizado dentro da função `usb_totp_spdm_init`, chamada no momento de inicialização do driver. Assim, esta função é chamada independentemente do que acontecer depois do *boot* do sistema, e a verificação da presença (ou não) do dispositivo SPDM USB conectado ao computador deve ser realizada dentro da própria *workqueue*. Para isso, as seguintes linhas foram adicionadas a esta função:

```
    wq = create_singlethread_workqueue("totp_spdm");
    if (wq){
        queue_work(wq, &totp_spdm_work);
    }
```

Trecho de Código 5.49: Inicialização da *workqueue*.

Com isso, é inicializada a *workqueue* `totp_spdm_work`, cujo trabalho em si é definida na função `totp_spdm_work_handler`. Esta função possui a seguinte aparência:

```
static void totp_spdm_work_handler(struct work_struct *w) {
    // Processamento realizado no inicio da execucao do work handler
    // [...]
```

```
    while(true){
        // Processamento periodico
        // [...]
        msleep(VERIFICATION_PERIOD_MS);
    }
}
```

Trecho de Código 5.50: Função `totp_spdm_work_handler`.

No qual `VERIFICATION_PERIOD_MS` define o intervalo em milissegundos entre duas execuções da porção periódica da *workqueue*.

5.7.3 Desenvolvimento de método de desligamento do sistema em caso de falha

Para servir como medida de segurança caso algum erro ocorra na execução dos testes do sistema, foi desenvolvida uma função simples que inviabiliza o uso da máquina nestes casos.

Para realizar o desligamento rápido da máquina, foi utilizada a seguinte função:

```
char * shutdown_argv[] = { "/sbin/poweroff", NULL };
call_usermodehelper(shutdown_argv[0], shutdown_argv, NULL,
    UMH_NO_WAIT);
```

Trecho de Código 5.51: Chamada da função para o desligamento da máquina.

Esta função chama o binário `/sbin/poweroff`, disponível na máquina virtual devido à presença do BusyBox como *init system*, ou o primeiro programa em *userspace*, responsável por inicializar os serviços e programas que executam em espaço de usuário, e é a solução padrão do Buildroot para a inicialização da máquina.

Com isso, o sistema é desligado instantaneamente, salvo exceções que serão abordadas em uma seção futura deste documento. Para evitar este tipo de problema, também foi adicionada uma chamada à função `kernel_halt()`, que desliga a máquina e realiza um *system halt*, parando a execução de todos os processos atuais do sistema.

5.7.4 Implementação do protocolo SPDM no driver

A porção do código referente à implementação do protoclo SPDM no driver foi desenvolvida em conjunto com o dispositivo mencionado previamente; dessa maneira, muitas das definições realizadas naquele código possuem um equivalente neste.

O processamento referente ao SPDM começa na função `totp_spdm_work_handler`, definida previamente, responsável pelo processamento dentro da *workqueue* do driver. Nesta função, após a definição inicial de variáveis, é importante garantir que o dispositivo foi encontrado. Para tal, é utilizado um *loop*, executado até que o valor da variável `totp_spdm_usb_struct->endpoints_count` seja diferente de zero, como indicado pelo código abaixo.

```
for (tries = 0; tries < MAX_TRIES; tries++){
   if (totp_spdm_usb_struct->endpoints_count != 0){
      pr_info("SPDM device found on attempt %d\n", tries);
      device_found = true;
      break;
   }
   msleep(TIMEOUT_MS);
}

if (!device_found){
   pr_alert("SPDM device not found!\n");
   fail();
}
```

Trecho de Código 5.52: *Loop* até encontrar um dispositivo SPDM.

A função sai do loop assim que o dispositivo for encontrado, e falha se o mesmo não for encontrado. O *timeout* existe para fornecer ao usuário um período no qual ele pode inserir o dispositivo caso o sistema *host* seja inicializado sem ele.

A definição dos valores para o intervalo entre tentativas e para o máximo de buscas foi realizada de maneira empírica. Primeiramente, para não sobrecarregar o sistema com a busca pelo dispositivo no momento de boot, definiu-se o intervalo (`TIMEOUT_MS` como 5000 milissegundos. Então, para o número de tentativas (`MAX_TRIES`, especificou-se inicialmente o valor de 2 tentativas, fornecendo ao usuário 10 segundos para inserir o dispositivo.

Vale ressaltar, no entanto, que este valor foi alterado para 9 tentativas durante os testes com o QEMU. Isso se deu porque o processo de *hot-plugging*, ou a adição de um dispositivo após o boot do sistema, é bem demorado no QEMU, necessitando da escrita de um longo comando no temrinal do monitor do aplicativo.

A variável `totp_spdm_usb_struct->endpoints_count` é definida em outra função, `usb_totp_spdm_probe`, também já citada neste documento, tendo a função de manter um valor atualizado de dispositivos inseridos.

Terminado esse processo, o contexto SPDM é inicializado. Assim como no dispositivo, funções separadas realizam um processo completo de inicialização da variável de contexto, definindo, por exemplo, a versão do SPDM utilizada, os algoritmos de *hashing* suportados, as funções de medição de firmware que podem ser utilizadas, e as funções de codificação e decodificação no protocolo MCTP. Estas definições são, em essência, idênticas às do dispositivo, em particular pela necessidade de se manter uma consistência entre os dois lados da comunicação.

Uma definição mais específica ao driver nesta função, no entanto, se dá pela definição de funções que enviam e recebem dados durante a comunicação com o dispositivo, realizada da seguinte maneira:

```
// Set functions for sending and receiving SPDM messages
spdm_register_device_io_func(
      spdm_context,
      spdm_usb_send_message,
      spdm_usb_receive_message);
```

Trecho de Código 5.53: Definição das funções de envio e recebimento de dados.

Feitas estas definições iniciais, é importante levar em consideração como ocorre o envio e recebimento de mensagens USB no driver. As duas funções chamadas pelo SPDM possuem o seguinte formato:

```
/*
* Send SPDM messages
*/
static return_status spdm_usb_send_message(
         IN void *spdm_context,
         IN uintn request_size,
         IN void *request,
```

```
          IN uint64 timeout) {
   // [...]
}

/*
* Receive SPDM message
*/
static return_status spdm_usb_receive_message(
          IN void *spdm_context,
          IN OUT uintn *response_size,
          IN OUT void *response,
          IN uint64 timeout){
   // [...]
}
```

Trecho de Código 5.54: Funções de envio e recebimento de dados.

Como pode-se notar, as duas funções são chamadas com o conhecimento do tamanho dos dados que serão enviados ou recebidos. Isso se mostra como um problema para a função de recebimento de dados, em particular, já que este dado não pode ser enviado de maneira separada do bloco de URB durante a transmissão. Com isso em mente, como já foi comentado na Seção 5.6.4.3, foi necessária a adição de um cabeçalho de quatro bytes, presente nas duas operações.

5.7.4.1 Operação de envio de mensagens SPDM por USB

A função de envio de dados deve, em primeiro lugar, inicializar o *struct* `completion`, de maneira que a função apenas seja finalizada com o término do envio com sucesso.

Em seguida, deve-se preparar o buffer de saída com o cabeçalho mencionado previamente. Para tal, são adicionados os 4 bytes de dados adicionais, por meio da conversão do tamanho da transferência, `request_size`, para hexadecimal com a função `sprintf`. Em seguida, o restante do buffer é copiado, e a requisição pode ser feita com este novo buffer, como indicado pelo código abaixo.

```
void *request_with_length_header;
char len_str[LEN_HEX_SIZE];

// Add 4 bytes of length to first positions of request buffer
```

```
request_with_length_header = kmalloc(request_size + LEN_HEX_SIZE,
    GFP_KERNEL);
sprintf(len_str, "%llx", request_size);
memcpy(request_with_length_header, len_str,
    LEN_HEX_SIZE*sizeof(char));

// Copy rest of request buffer
memcpy(request_with_length_header + LEN_HEX_SIZE, request,
    request_size);
```

Trecho de Código 5.55: Processamento do buffer.

Por fim, basta enviar os dados. Este processo é realizado por meio de uma chamada à função `send_arbitrary_data`, como visto no seguinte trecho:

```
send_arbitrary_data(request_with_length_header, request_size +
    LEN_HEX_SIZE);
```

Trecho de Código 5.56: Chamada da função para o envio de dados.

Esta função segue o processo descrito na Seção 5.7.1.

5.7.4.2 Operação de recebimento de mensagens SPDM por USB

A função de recebimento de dados funciona de maneira similar. Um detalhe importante a se levar em consideração, no entanto, é que este processo parte de uma requisição por parte do driver, e a requisição precisa conter o tamanho esperado, conforme especificado anteriormente.

Com isso em mente, é necessário o uso do cabeçalho de quatro bytes com o tamanho da requisição. No entanto, o funcionamento dele será diferente do caso anterior: a requisição inicial será grande, especificando o tamanho máximo possível da transmissão, e o cabeçalho será criado do lado do dispositivo. O driver, então, extrai a mensagem e seu tamanho oficial na função de *callback*.

Olhando o código mais a fundo, nota-se, primeiramente, a definição do vetor de resposta:

```
// Setting the response array to the maximum possible size initially
totp_spdm_usb_struct->response_data =
    kmalloc(MAX_SPDM_MESSAGE_BUFFER_SIZE, GFP_DMA);
```

```
totp_spdm_usb_struct->response_size = *response_size;
```

Trecho de Código 5.57: Definição do vetor de resposta.

O tamanho utilizado, `MAX_SPDM_MESSAGE_BUFFER_SIZE`, é oriundo de dados da própria libspdm, sendo igual a 4608 bytes. Este é o tamanho máximo das mensagens trocadas pela biblioteca.

Outro detalhe relevante é a definição da *flag* `GFP_DMA`. Esta é responsável por realizar a alocação de memória na região de DMA (Direct Memory Access) do computador, e a ausência desta causa o erro `transfer buffer not dma capable`, que impossibilitava a transferência.

Feito isso, o fluxo do código é similar ao descrito na Seção 5.7.1. Um struct de `completion` é inicializado no começo da execução da função. Então, é chamada a função `recv_arbitrary_data`, que realiza o processo de envio do URB. O resultado é retornado e processado na função de *callback*.

Por fim, quando a finalização do struct `completion` é sinalizada, é realizada a cópia do buffer original para o buffer de saída do SPDM, como indicado no trecho abaixo.

```
wait_for_completion(&totp_spdm_usb_struct->spdm_response_done);

*response_size = totp_spdm_usb_struct->response_size;
memcpy(response, totp_spdm_usb_struct->response_data,
   totp_spdm_usb_struct->response_size);
// [...]
return RETURN_SUCCESS;
```

Trecho de Código 5.58: Cópia do buffer original o buffer de saída do SPDM.

Olhando para a função de *callback*, nota-se primeiramente que é chamada uma outra função, `get_size_from_response`, como visto no bloco de código abaixo.

```
/*
* Callback funciton for receiving messages
*/
static void recv_arbitrary_data_callback(struct urb *urb){
   // Work with received data
   totp_spdm_usb_struct->response_size = get_size_from_response();
```

```
    //[...]
}
```

Trecho de Código 5.59: Parte da função de *callback*.

A responsabilidade desta é essencialmente extrair o valor do cabeçalho do buffer recebido. As quatro primeiras posições de memória do buffer são salvas em um buffer local, a partir do qual a função `hexdec`, já mencionada previamente, é chamada para realizar a conversão da string hexadecimal. O valor final, já convertido para decimal, é retornado da função.

Seu código pode ser visto no trecho abaixo.

```
/*
* Fetches SPDM response size from specific byte from the response
    itself
*/
static size_t get_size_from_response(void) {
    uint8_t size_hex[LEN_HEX_SIZE];
    unsigned long size_dec;

    memcpy(size_hex, totp_spdm_usb_struct->response_data,
        LEN_HEX_SIZE*sizeof(uint8_t));
    size_dec = hexdec(size_hex);
    return size_dec;
}
```

Trecho de Código 5.60: Retorno do valor do cabeçalho do buffer em decimal.

Vale ressaltar apenas que os dados do buffer estão contidos na variável `totp_spdm_usb_struct->response_data`, que é global por estar contida no struct de contexto do driver. O valor retornado pela função é salvo na variável `totp_spdm_usb_struct->response_size`.

Então, levando-se em conta o caso específico da transmissão de dados de 60 bytes com cabeçalho de 4 bytes adicionais, citado na Seção 5.6.4, é realizada mais uma manipulação de dados. No caso, o tamanho do cabeçalho deve ser definido, para que os dados recebidos sejam movidos, removendo, assim, o cabeçalho, que já foi processado.

Este processamento é realizado como indicado no seguinte trecho de código:

```
header_size = (totp_spdm_usb_struct->response_size == 60) ?
    (SPDM_RECEIVE_OFFSET - 1) : SPDM_RECEIVE_OFFSET;

memmove (totp_spdm_usb_struct->response_data,
       totp_spdm_usb_struct->response_data + header_size,
       (totp_spdm_usb_struct->response_size) * sizeof(uint8_t));
```

Trecho de Código 5.61: Remoção do cabeçalho processado.

Realizado este processo, pode-se sinalizar a finalização do struct `completion`, permitindo a continuação do processamento da função `spdm_usb_receive_message`.

Dessa forma, finaliza-se o procedimento necessário para o envio e recebimento de dados por meio de URBs pelo driver.

5.7.4.3 Inicialização da comunicação SPDM

Com isso tudo em mente, a comunicação SPDM pode ser inicializada. Para tal, é realizada inicialmente a operação de autenticação do dispositivo por SPDM, por meio do envio das mensagens `GET_VERSION`, `GET_CAPABILITIES`, e `NEGOTIATE_ALGORITHMS`.

Esta porção da execução é executada com o auxílio da função `spdm_init_connection`, disponível por padrão na libspdm, que já envia todos os valores inicializados previamente ao dispositivo, utilizando as funções definidas para envio e recebimento de dados. A função é chamada da seguinte maneira:

```
// Send get_version, get_capabilities, and negotiate_algorithms
totp_spdm_usb_struct->spdm_status = spdm_init_connection(
       totp_spdm_usb_struct->spdm_context, false);
if (RETURN_ERROR(totp_spdm_usb_struct->spdm_status)) {
   pr_alert("Error on spdm_init_connection: %llX.",
       totp_spdm_usb_struct->spdm_status);
   err_free_spdm();
   fail();
} else {
   pr_info("SPDM Context initialized.");
}
```

Trecho de Código 5.62: Chamada da função para envio dos valores inicializados.

Feito este processo, é inicializada a verificação mais específica do dispositivo, utilizando certificados.

De maneira similar à função de estabelecimento de configurações iniciais do SPDM, define-se a função `init_spdm_certificates`, que inicializa os dados relacionados aos certificados.

Primeiro, nota-se que a conexão neste momento deve estar no estado `SPDM_CONNECTION_STATE_NEGOTIATED`. Isso significa que as operações prévias (`GET_VERSION`, `GET_CAPABILITIES`, e `NEGOTIATE_ALGORITHMS`), devem ter sido realizadas com sucesso.

Dados específicos aos certificados foram guardados em um outro arquivo, denominado `certs.h`, por motivos de organização. A função `init_spdm_certificates` extrai estes valores, guardando-os no contexto SPDM.

Em seguida, a etapa de autenticação SPDM pode ser, de fato, inicializada, por meio do envio das operações `GET_DIGEST`, `GET_CERTIFICATE`, e `CHALLENGE`. Estas são realizadas por meio de uma função auxiliadora similar à `init_spdm_certificates`, mas desenvolvida pelos professores do LARC, denominada `spdm_do_authentication`.

Nesta função, muitos buffers são definidos e alocados – entre eles, nota-se o `cert_chain`, que guarda a cadeia de certificados quando esta estiver disponível, e o `total_digest_buffer`, que guarda o resultado da operação de *digest*.

Para cada uma das três operações realizadas, é utilizado um código definido como descrito a seguir:

```
if ((m_exe_connection & EXE_CONNECTION_DIGEST) != 0) {
    status = spdm_get_digest(context, slot_mask,
                total_digest_buffer);
    if (RETURN_ERROR(status)) {
        return status;
    }
}
```

Trecho de Código 5.63: Código para a realização das operações.

As outras duas operações utilizam os seguintes parâmetros:

```
if (slot_id != 0xFF) {
    status = spdm_get_certificate(
```

```
      context, slot_id, cert_chain_size, cert_chain);
// [...]
}
// [...]
status = spdm_challenge(context, slot_id, measurement_hash_type,
               measurement_hash);
```

Trecho de Código 5.64: Código para a realização do restante das operações.

Feito este processo, a comunicação SPDM é estabelecida. Nesta etapa, é fundamental inicializar também uma sessão SPDM, a qual será utilizada para os envios periódicos realizados em seguida. Este processo é realizado com a seguinte chamada à função `spdm_start_session`:

```
// Start SPDM session
totp_spdm_usb_struct->spdm_status =
   spdm_start_session(totp_spdm_usb_struct->spdm_context,
      use_psk,
      m_use_measurement_summary_hash_type,
      m_use_slot_id,
      &totp_spdm_usb_struct->session_id,
      &heartbeat_period,
      measurement_hash);
if (RETURN_ERROR(totp_spdm_usb_struct->spdm_status)) {
   // [...]
   fail();
}
```

Trecho de Código 5.65: Inicialização das sessão SPDM.

Nesta chamada, `use_psk` é definida como `FALSE`, já que não são utilizadas chaves pré-compartilhadas. Além disso, é relevante notar que `&totp_spdm_usb_struct->session_id` começa como uma variável vazia, sendo que o ID da sessão será adicionado a ela quando estiver disponível.

A partir deste momento, termina-se a verificação inicial e estabelecimento da comunicação SPDM, permitindo o começo das verificações periódicas entre driver e dispositivo.

5.7.5 Implementação do algoritmo TOTP no driver

Para a realização da geração de códigos TOTP de maneira idêntica ao dispositivo, o driver utilizou também a biblioteca TOTP-MCU, desenvolvida por Netthaw, disponível em seu GitHub [21] e disponibilizada sob licença da MIT.

Os códigos da biblioteca foram adaptados para o uso dentro do kernel. Isso exigiu, em particular, a mudança dos `includes` do código padrão, para o uso de funções e tipos específicos do kernel. Além disso, a estrutura `TimeStruct`, utilizada no código original, não está disponível na versão de kernel utilizada, sendo alterada para o struct `tm`, de uso similar.

5.7.5.1 Estrutura geral dos fluxos TOTP

Com isso em mente, vale ressaltar que o fluxo do TOTP é separado em duas etapas: a geração e envio da chave secreta e a geração do código TOTP em si.

Depois do estabelecimento da comunicação SPDM, o *work handler* responsável pelas funções específicas do driver entra em um loop infinito, o qual é executado constantemente até o desligamento do sistema ou a aparição de um erro.

Neste loop, é inicialmente verificada a necessidade de se gerar uma nova chave secreta para o TOTP. A primeira execução do código deverá sempre criar uma nova chave junto ao dispositivo, a qual será utilizada para as verificações seguintes.

Além disso, a geração de chaves deve ocorrer novamente após uma certa quantidade de verificações, de maneira a minimizar a possibilidade de um atacante conseguir descobrir esta chave secreta. Definiu-se que o valor ideal para esta quantidade era de $log_2 N$, no qual N corresponde ao número de bits do número aleatório. Sendo assim, como os números aleatórios gerados em cada lado da comunicação têm 64 bits, foi calculado o valor de $log_2 64 = 6$. Com isso, a cada 6 verificações periódicas, o fluxo de geração de novas chaves é executado novamente.

A nível de código, foi utilizada uma constante, `TOTP_KEY_CHECKS_UNTIL_REGEN`, que define este valor. Então, uma variável, `totp_spdm_usb_struct->totp_checks`, é responsável por manter salva a quantidade de verificações que ainda devem ser realizadas antes de uma nova verificação, sendo inicializada com o valor 0 antes da primeira.

Dentro do loop infinito, é verificado se o valor desta variável é igual a 0. Se for,

inicia-se o fluxo de geração de chaves e, ao final, o valor da variável é redefinido como TOTP_KEY_CHECKS_UNTIL_REGEN. Então, inicia-se o outro fluxo, referente à verificação TOTP em si; ao final deste, reduz-se o valor da variável em 1, e o sistema aguarda o período de verificação para retornar ao loop.

A estrutura do código é descrita no seguinte trecho.

```
while(true){
   pr_info("Initializing periodic SPDM checks.");

   // Begin creation of new TOTP key periodically
   if (totp_spdm_usb_struct->totp_checks <= 0){
      // [...]

      // Set totp_checks as max number of key checks again
      totp_spdm_usb_struct->totp_checks =
         TOTP_KEY_CHECKS_UNTIL_REGEN;
   }

   // Begin TOTP check
   // [...]

   pr_info("All SPDM checks realized successfully.");

   // totp_checks -= 1 to bring it closer to regenerating the key
   totp_spdm_usb_struct->totp_checks--;
   msleep(VERIFICATION_PERIOD_MS);
}
```

Trecho de Código 5.66: Loop infinito.

A partir deste ponto, pode-se observar mais especificamente o funcionamento de cada fluxo.

5.7.5.2 Fluxo de geração de chave secreta para o TOTP

Para a geração de uma nova chave secreta, o driver deve, inicialmente, gerar o número aleatório que será enviado ao dispositivo. Para isso, é utilizada a função get_random_bytes, que retorna números aleatórios criptograficamente seguros, ade-

quados para uso em chaves desse tipo, sem utilizar o gerador de números aleatórios específico do hardware da máquina [27]. Esta função é utilizada para gerar um valor de 64 bits, como visto no código a seguir.

```
// Create a random number
get_random_bytes(
      &(totp_spdm_usb_struct->random_local_num),
      sizeof(totp_spdm_usb_struct->random_local_num));
```

Trecho de Código 5.67: Função para geração de números aleatórios.

Vale ressaltar apenas que `random_local_num` é inicializado como um vetor de `uint8_t` com 8 posições.

Para enviar este valor ao dispositivo, deve-se inicialmente preparar o vetor que receberá os dados. Este é inicializado com a constante `MCTP_MESSAGE_TYPE_VENDOR_DEFINED_IANA`, que, como visto no código do dispositivo, auxilia a libspdm no processo de identificar o envio e recebimento das mensagens customizadas criptografadas.

Depois de definir o vetor de envio de dados e inicializar o vetor de recebimento da chave, utiliza-se a função `spdm_send_receive_data`, disponibilizada pela biblioteca libspdm, que realiza o envio de mensagens customizadas por uma sessão SPDM já estabelecida.

Esta função envia o número aleatório de 64 bits gerado para o dispositivo, que processa o resultado e envia de volta para o driver a nova chave, com 384 bits, processo descrito na Seção 5.6.3.

É importante notar também que esta função utiliza a encriptação da comunicação SPDM para realizar o envio destes dados, cujo sigilo é fundamental. A ausência de uma proteção dessas poderia facilitar um ataque do tipo Man-In-The-Middle, no qual um atacante poderia enxergar e falsificar códigos TOTP, comprometendo a segurança do dispositivo.

O seguinte trecho de código exemplifica o uso da função:

```
// Send vendor defined request with random number
spdm_send_receive_data(
      totp_spdm_usb_struct->spdm_context,
      &totp_spdm_usb_struct->session_id,
```

```
    FALSE,
    &totp_spdm_usb_struct->spdm_random_num_data_buf,
    sizeof(totp_spdm_usb_struct->spdm_random_num_data_buf),
    &send_receive_local_response_buf,
    &totp_spdm_usb_struct->totp_key_size);
```

Trecho de Código 5.68: Exemplo de uso da função `spdm_send_receive_data`.

Nesta chamada, vale notar que a estrutura `totp_spdm_usb_struct->spdm_random_num_data_buf` guarda o vetor de saída de dados para o dispositivo, contendo a *flag* `MCTP_MESSAGE_TYPE_VENDOR_DEFINED_IANA` e o número aleatório de 8 bytes, enquanto que `send_receive_local_response_buf` guarda o valor de resposta do dispositivo, que conterá a mesma *flag* como primeiro byte, seguido dos 48 bytes referentes à chave secreta enviada pelo dispositivo.

Também é importante notar a variável de tamanho do buffer de saída, `totp_spdm_usb_struct->totp_key_size`. Esta deve ser inicializada com o valor máximo do buffer, que é equivalente a 49 bytes: 48 bytes da chave e 1 byte da *flag* do SPDM. A variável é enviada com este valor, sendo alterada conforme a resposta do dispositivo pela própria libspdm.

Com isso, torna-se necessária a alteração desses valores para o uso pelo TOTP. Para isso, é removido um byte do tamanho da chave, referente à *flag*. O buffer de resposta, então, é copiado para a variável global `totp_spdm_usb_struct->totp_key`, tomando os devidos cuidados para enviar exatamente a chave obtida da resposta, como indicado no código abaixo.

```
// Remove one byte from totp_key_size (SPDM flag)
totp_spdm_usb_struct->totp_key_size -= 1;

// Copy response buffer to totp_key
memcpy(totp_spdm_usb_struct->totp_key,
      send_receive_local_response_buf + 1,
      totp_spdm_usb_struct->totp_key_size);
```

Trecho de Código 5.69: Processamento da chave TOTP.

Por fim, é importante ressaltar também que, em alguns raros casos, a execução da função da libspdm é demorada, fazendo com que a execução do fluxo termine sem a chave propriamente dita estar contida em sua respectiva variável. O método utilizado pelo grupo

para evitar problemas similares nas funções de envio de dados não é de fácil uso neste caso, no entanto, já que envolveria alterar a própria biblioteca da libspdm ainda mais, o que o grupo buscou evitar.

Visto isso, para lidar com tais problemas quando estes ocorressem, este fluxo como um todo está contido em um bloco `do {...} while`, verificando a condição de tamanho da chave obtida na resposta. Assim, sempre que a resposta tiver um tamanho diferente do esperado, igual ao tamanho da chave TOTP, o processo é realizado novamente.

O seguinte trecho de código exemplifica este processo.

```
// Begin creation of new TOTP key periodically
if (totp_spdm_usb_struct->totp_checks <= 0){
   do {
      pr_info("Generating TOTP key.");
      // [...]

      // Final check to see if the loop will be done again
   } while (totp_spdm_usb_struct->totp_key_size != TOTP_KEY_SIZE);
   // [...]
}
```

Trecho de Código 5.70: Exemplo de verificação do tamanho da chave.

Com isso, encerra-se o fluxo de geração da chave, e pode-se observar com maiores detalhes o fluxo de verificação de consistência dos códigos TOTP.

5.7.5.3 Fluxo de verificação de consistência do código TOTP

Com uma chave já gerada e conhecida tanto pelo driver quanto pelo dispositivo, o driver realiza o envio de mais uma mensagem ao dispositivo. Dessa vez, o buffer enviado corresponde a um vetor de 6 bytes, sendo que sua única informação relevante é o primeiro byte, referente à mesma *flag* necessária ao uso da função – `MCTP_MESSAGE_TYPE_VENDOR_DEFINED_IANA`.

Este tamanho se dá devido à função necessitar de algum dado sendo enviado para que o dispositivo possa dar uma resposta. Dessa forma, estes dados enviados não são de fato utilizados, sendo importantes apenas os dados recebidos depois da execução da função.

Então, a função `spdm_send_receive_data` é chamada. O dispositivo gera seu código TOTP e o envia para o driver, que o recebe no buffer `totp_response`. A

chamada para a função de envio e recebimento de dados é como indicado no seguinte bloco de código.

```
// Send vendor defined request with random number
spdm_send_receive_data(
      totp_spdm_usb_struct->spdm_context,
      &totp_spdm_usb_struct->session_id,
      FALSE
      &totp_spdm_usb_struct->spdm_totp_check_data_buf
      sizeof(totp_spdm_usb_struct->spdm_totp_check_data_buf),
      &totp_response,
      &totp_spdm_usb_struct->totp_size);
```

Trecho de Código 5.71: Chamada da função `spdm_send_receive_data`.

Então, o procedimento de verificação do TOTP é realizado por meio da chamada à função `verify_totp`, que recebe o buffer em questão. Nessa chamada, também se elimina diretamente a *flag* recebida junto ao código ao se utilizar o buffer apenas a partir de sua segunda posição de memória:

```
// Verify TOTP from response
verify_totp(totp_response + 1);
```

Trecho de Código 5.72: Chamada da função `verify_totp`.

Dentro da função `verify_totp`, o código TOTP recebido é convertido de hexadecimal para decimal por meio da função `hexdec` já citada previamente. Então, esta função chama uma outra, denominada `totp_challenge`, que gera o código TOTP do driver e o compara ao recebido do dispositivo. Se não houver correspondência, a função de falha é chamada, interrompendo o uso do sistema como um todo. O código referente a esta função pode ser visto no trecho abaixo.

```
// Transform TOTP hex into unsigned int
totp_dec = hexdec(totp_hex);

// Check TOTP consistency
result = totp_challenge(totp_dec);
if (!result){
   pr_alert("TOTP %u did not match the expected value.", totp_dec);
   fail();
```

```
}
else {
   pr_alert("TOTP %u matches the expected value.", totp_dec);
}
```

Trecho de Código 5.73: Processamento do código TOTP.

Olhando mais a fundo a função `totp_challenge`, nota-se que esta inicialmente gera um vetor de códigos TOTP por meio da função `get_totp`.

Esta geração se dá por uma modificação no código fonte da biblioteca TOTP-MCU. Após a inicialização das funções com a chave secreta obtida no fluxo anterior, é criado um *struct* do tipo `ts`, inicializado com a hora atual, e a porção desse *struct* referente ao tempo atual em segundos – o tempo Unix informado pelo kernel – é dividido para encontrar o valor em *time-steps* deste tempo.

Esta informação, que normalmente não seria útil para uma simples geração de códigos TOTP, torna-se vital para a verificação que o driver deve realizar. Para evitar possíveis problemas de dessincronização pequenos, dado o *time-step* atual x, o driver gera também os códigos TOTP para os *time-steps* $x - 1$ e $x + 1$. Assim, mesmo que haja uma dessincronização pequena entre os relógios do dispositivo e do driver, ou se a geração do código TOTP pelo dispositivo tiver acontecido muito próximo ao final do *time-step*, a verificação não retorna erros, que poderiam acontecer sem esta adição.

O número de verificações realizadas pelo driver é facilmente modificado, caso haja uma situação na qual dessincronizações maiores que um *time-step* sejam possíveis. A constante `TOTP_CHALLENGE_ATTEMPTS`, definida inicialmente como 3, representa o número de verificações que o driver realiza, de tal modo que o tempo atual está no meio da lista de *time-steps* verificados. Assim, alterar o valor da constante para 7, por exemplo, automaticamente faz com que o driver passe a gerar códigos TOTP do *time-step* $x - 3$ ao $x + 3$.

Esta função pode ser vista no trecho de código abaixo.

```
/*
* Creates an array of TOTP values and saves them into an array
*/
static void get_totp(uint32_t* code_array){
   // [...]
```

```
    TOTP(totp_spdm_usb_struct->totp_key,
        (uint8_t)totp_spdm_usb_struct->totp_key_size, TOTP_TIMESTEP);
        // key, key size, timestep in s

    // Get current time
    ts = kmalloc(sizeof(*ts), GFP_KERNEL);
    getnstimeofday(ts);

    // Get number of steps
    steps = ts->tv_sec / TOTP_TIMESTEP;

    // This for loop creates TOTP_CHALLENGE_ATTEMPTS TOTPs
    // It should create half of TOTP_CHALLENGE_ATTEMPTS, rounded down,
    // attempts before and after the specific amount of timesteps
    // as such, TOTP_CHALLENGE_ATTEMPTS = 3 tries from steps-1 to
        steps+1
    // TOTP_CHALLENGE_ATTEMPTS = 5 tries from steps-2 to steps+2 and
        so on
    diff = TOTP_CHALLENGE_ATTEMPTS / 2;
    for(i = 0; i < TOTP_CHALLENGE_ATTEMPTS; i++){
        code_array[i] = getCodeFromSteps(steps - diff);
        pr_info("Generated TOTP %u with steps %u", code_array[i],
            steps - diff);
        diff--;
    }
}
```

Trecho de Código 5.74: Função get_totp.

Com isso, o vetor passado como argumento para a função get_totp passa a conter a lista completa de códigos TOTP gerados pelo driver. Concluída esta execução, a função totp_challenge agora chama a função valueinarray, que simplesmente retorna 1 caso o valor especificado esteja contido em um vetor, e 0 caso contrário, para descobrir se o código TOTP recebido a partir do dispositivo corresponde a qualquer um do vetor de códigos gerados localmente – se não houver correspondência, houve uma falha, que redireciona o código para a função de falha.

O código completo desta função pode ser visto no trecho abaixo.

```
/*
* Checks whether the specified TOTP value is valid or not, by
   generating
* TOTP_CHALLENGE_ATTEMPTS TOTP values and chcking whether any of them
* match the value coming from the device, dev_totp.
* Returns 1 if true, 0 otherwise.
*/
static int totp_challenge(uint32_t dev_totp){
   uint32_t driver_totp_array[TOTP_CHALLENGE_ATTEMPTS];

   get_totp(driver_totp_array);

   if(!valueinarray(dev_totp, driver_totp_array,
      TOTP_CHALLENGE_ATTEMPTS)){
      return 0;
   }

   return 1;
}
```

Trecho de Código 5.75: Função `totp_challenge`.

Depois da execução destas funções, entra-se no período de espera, no qual o processo do driver não realiza nenhuma ação. Terminado este período, o código volta ao início do loop infinito do *work handler*, realizando novamente os dois fluxos conforme especificado.

5.8 Implementação do dispositivo físico

Terminados os desenvolvimentos do driver e do dispositivo emulado pelo QEMU, pode-se começar a trabalhar no dispositivo físico, que substituirá o dispositivo emulado pelo QEMU. Este deverá cumprir as mesmas funções citadas na Seção 5.6.

Para tal, é necessário, primeiramente, definir um dispositivo que será visto pela máquina *host* como dispositivo USB com o mesmo Product ID e Vendor ID do dispositivo emulado pelo QEMU, o que permitirá que o driver criado previamente interaja com ele. Para tal, foi utilizada a placa Intel Galileo [28], baseada no processador Quark X1000, também da Intel. Esta placa possui suporte à especificação USB On-The-Go (OTG), que permite que dispositivos USB ajam como dispositivo principal na comunicação USB, de

tal modo que outros dispositivos podem ser conectados a eles.

Dessa forma, a placa pode ser utilizada como intermediário da conexão USB da máquina *host* com um dispositivo criado dentro dela mesma, permitindo, assim, a customização completa deste novo dispositivo.

Este processo é realizado com o auxílio do módulo *Mass Storage Gadget*, disponível no kernel do Linux [29]. Este é acionado por meio do comando `modprobe`, que executa o módulo referente a esse sistema, denominado `g_mass_storage`, configurando um arquivo ou região de memória como um dispositivo de *mass storage*.

Para o uso deste conjunto, é necessário também criar uma nova imagem do Linux, separada da imagem criada previamente. Para tal, foi utilizado inicialmente a ferramenta Yocto, um projeto *open-source* que permite a criação de imagens customizadas com foco em sistemas embarcados [30]. Para acelerar e unificar o processo de configuração da ferramenta, seu uso foi dado em conjunto com o Docker, que permite a definição de um sistema isolado para a execução de uma tarefa em um contêiner [31].

Estas duas ferramentas, no entanto, provaram ser consideravelmente demoradas de se usar e de se aprender. Aliado ao fato de ser necessária a compilação cruzada da biblioteca libpsdm para o novo dispositivo, o grupo optou por utilizar novamente o Buildroot como ferramenta para a criação da nova imagem do Linux.

Com o dispositivo sendo reconhecido corretamente como dispositivo USB, e com a libspdm funcional em seu sistema operacional, pode-se iniciar a codificação de um arquivo em C, que será executado junto ao boot do sistema e cumprirá os requisitos apresentados previamente.

A arquitetura do sistema segue o modelo indicado pela Figura 12.

Neste modelo, nota-se que a placa receberá dados referentes às requisições SPDM e aos dados da conexão USB com o sistema *host*. Recebidos esses dados, a placa realizará o processamento de dados em duas camadas da pilha de software – de aplicação e de driver.

A nível de aplicação, um código cria um dispositivo USB dentro da placa. Este dispositivo é conectado ao computador por meio da tecnologia USB OTG. Assim, esta pode ser reconhecida como o dispositivo USB SPDM que se comunicará diretamente com o driver criado previamente.

A nível de driver, é necessário que duas funções sejam cumpridas pela placa: a geração de chaves e códigos TOTP, e a atuação desta como SPDM Responder na comunicação com o driver.

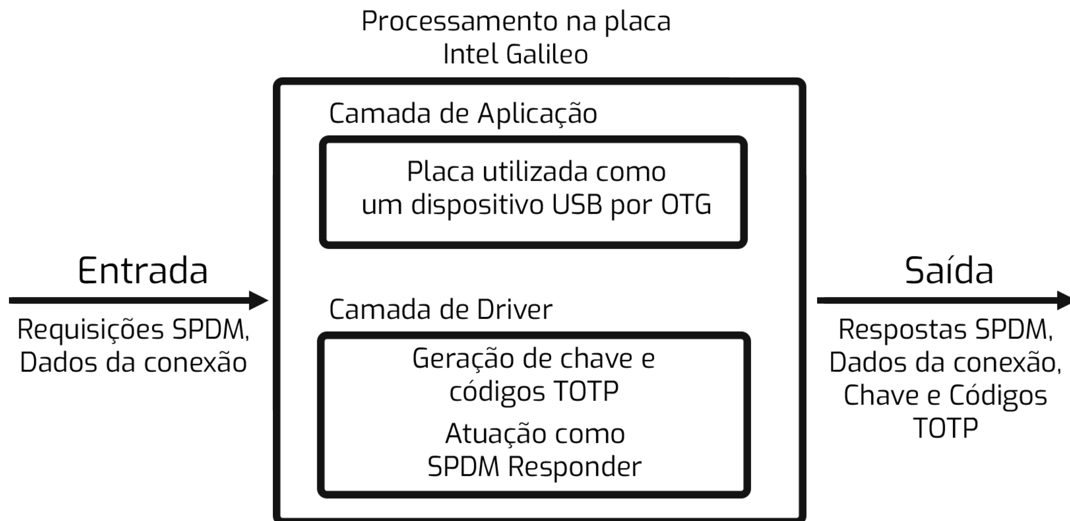


Figura 12: Arquitetura do sistema físico.

Fonte: do autor.

Por fim, a placa deve ser capaz de enviar de volta ao driver as respostas SPDM obtidas de seu processamento, além de expor os dados da conexão e compartilhar, de maneira encriptografada e segura, a chave e os códigos TOTP gerados.

5.8.1 Uso da placa Intel Galileo e da tecnologia USB OTG

Como citado previamente, a placa utilizada para este projeto foi a Intel Galileo, que facilita o desenvolvimento de um dispositivo USB.

Utilizando a tecnologia OTG, elimina-se a necessidade de haver um computador entre dois dispositivos [32], permitindo que a placa Intel Galileo aja como dispositivo USB principal no sistema desenvolvido.

A Figura 13 demonstra a organização da placa Intel Galileo com o sistema *host*. Em particular, nota-se a conexão serial para USB da placa, que permite que o conteúdo do sistema operacional seja visualizado com o auxílio de programas como o `screen`, e a conexão USB OTG, pela qual a placa pode ser reconhecida como dispositivo USB.

Vale notar também que o uso da conexão serial se dá pelo seguinte comando:

```
$ sudo screen /dev/ttyUSB0 115200
```

Trecho de Código 5.76: Conexão serial por meio do comando `screen`.

Figura 13: Organização do sistema físico.

Fonte: do autor.

Neste, ressalta-se que a placa corresponde à interface `ttyUSB0` no sistema *host*, e que o valor `115200` corresponde à taxa baud do sinal.

5.8.2 Configuração de ambiente para uso do dispositivo físico

Para utilizar o módulo *Mass Storage Gadget*, como citado previamente, é necessário inicialmente preparar o ambiente. Para isso, o grupo utilizou um cartão de memória com duas partições – uma que receberá a imagem do Linux que será criada, e outra que será utilizada como dispositivo de *mass storage*, contendo os arquivos que serão executados no momento de boot.

Para formatar o cartão no formato FAT32, utilizou-se as ferramentas `fdisk` e `mkfs.vfat`.

Para que o dispositivo seja inicializado na imagem do Linux correta, basta que a primeira partição do cartão contenha três arquivos – `bzImage` e `rootfs.cpio.bz2`, referentes ao kernel e ao `initrd`, respectivamente, que são gerados pelo Buildroot e contêm a imagem nova como um todo, e um arquivo `boot/grub/grub.conf`, que contém instruções para que o boot aconteça automaticamente utilizando os dois outros arquivos.

Então, com a imagem funcionando, é necessário acionar a partição de *mass storage* como tal. Este processo é realizado por meio do seguinte comando:

```
$ modprobe g_mass_storage file=/dev/mmcblk0p2 removable=y
    productId=0x0666 vendorId=0x0666
```

Trecho de Código 5.77: Comando para acionamento do módulo `g_mass_storage`

Este comando inicializa o módulo `g_mass_storage`, utilizando o arquivo `/dev/mmcblk0p2` como base. Este arquivo é referente à partição do cartão de memória referente ao *mass storage*, e pode ser descoberto com o auxílio do comando `fdisk -l`, que lista todos os discos e partições visíveis.

A opção `removable=y` define o sistema de *mass storage* como um dispositivo USB; sem ela, o sistema *host* o reconhecerá como um CD-ROM. Já as opções `productId=0x0666` e `vendorId=0x0666` definem o Product ID e Vendor ID do dispositivo com os valores desejados.

Feito isso, é possível ver no sistema *host* a adição de um novo dispositivo USB, como indicado pelos logs do `dmesg` indicados na Figura 14. Na figura, também é possível notar

que o Product ID e o Vendor ID foram definidos corretamente.

```
[  449.515480] usb 3-2: New USB device found, idVendor=0666, idProduct=0666, bcdDevice= 3.14
[  449.515486] usb 3-2: New USB device strings: Mfr=3, Product=4, SerialNumber=0
[  449.515487] usb 3-2: Product: Mass Storage Gadget
[  449.515489] usb 3-2: Manufacturer: Linux 3.14.28-ltsi with pch_udc
[  449.543210] usb-storage 3-2:1.0: USB Mass Storage device detected
[  449.543422] scsi host6: usb-storage 3-2:1.0
[  449.543565] usbcore: registered new interface driver usb-storage
[  449.547942] usbcore: registered new interface driver uas
[  450.575641] scsi 6:0:0:0: Direct-Access     Linux    File-Stor Gadget 0314 PQ: 0 ANSI: 2
[  450.575869] sd 6:0:0:0: Attached scsi generic sg2 type 0
[  450.576308] sd 6:0:0:0: Power-on or device reset occurred
[  450.576789] sd 6:0:0:0: [sdb] 2048 512-byte logical blocks: (1.05 MB/1.00 MiB)
```

Figura 14: Logs indicando a adição de um novo dispositivo USB.

Fonte: do autor.

Por fim, é possível utilizar o novo dispositivo no QEMU, de tal forma que ele dialogue com o driver desenvolvido previamente. Para tal, utiliza-se o dispositivo `usb-host`, disponível no QEMU, que realiza a operação de *pass-through*. Este necessita apenas do conjunto de Product ID e Vendor ID do dispositivo USB, ou pelos números da porta e do barramento USB na máquina *host*, que podem ser obtidos pelo comando `lsusb -t`, como visto nos exemplos abaixo:

```
$ -device usb-host,vendorid=0x0666,productid=0x0666
$ -device usb-host,hostbus=3,hostport=10
```

Trecho de Código 5.78: *Pass-through* do dispositivo USB para o QEMU.

Estes dispositivos são passados para o QEMU adicionando-se um destes comandos à chamada do aplicativo, alterando-se o campo de dispositivo no Trecho de Código 5.12.

5.8.3 Criação de imagem com uso de Buildroot e Yocto aliado a Docker

Inicialmente, a imagem do Linux que seria utilizada pela placa Galileo foi criada com o Yocto, ferramenta comumente utilizada para o desenvolvimento de novas imagens do Linux para dispositivos embarcados. Para utilizá-lo, foi criada inicialmente um contêiner no Docker, de tal maneira que se pudesse facilitar o uso do programa.

Foram criados diferentes contêineres para a execução desta tarefa. Estes utilizavam como base o Ubuntu nas versões 14.04 e 16.04, que foram recomendadas como versões mais apropriadas para a compilação das *releases* mais antigas do programa *open-source* que ainda estão sob suporte de longo prazo.

Para este processo, foi utilizada a versão 3.1 do Yocto, intitulada Dunfell.

Dentro do Yocto, foram utilizadas algumas imagens já prontas para o dispositivo. Este processo é realizado por meio do comando `bitbake`, especificando-se a imagem desejada. As três imagens testadas pelo grupo foram:

`core-image-minimal` Uma pequena imagem básica do Linux, com apenas o mínimo necessário para inicializar o dispositivo [33]. Seria utilizada em conjunto com mais funcionalidades adicionadas futuramente.

`core-image-minimal-dev` Outra imagem básica do Linux, baseada na anterior, mas com bibliotecas e *headers* necessários para o desenvolvimento de funções mais avançadas para a placa [33]. Seria ideal para a adição da biblioteca libspdm, por já possuir os requisitos mínimos necessários para seu funcionamento.

`iot-devkit-image` Imagem do Yocto preparada para dispositivos IoT, incluindo também a placa Galileo. Tem *features* desnecessárias para o projeto, mas a equipe julgou válido testá-la também quando as duas supracitadas não compilaram com sucesso.

Infelizmente, no entanto, o desenvolvimento com o Yocto provou-se muito difícil e demorado. As três imagens utilizadas demoraram mais de uma dezena de horas cada para compilar, e surgiram diversos erros durante a compilação. Alguns foram solucionados com sucesso, mas sempre surgia algum erro que o grupo não soube solucionar a tempo.

Visto essa dificuldade, decidiu-se utilizar o Buildroot para criar a imagem do Linux para a placa Intel Galileo. O uso desta aplicação em outro estágio do projeto foi útil como base para a criação desta imagem, e o uso prévio da libspdm em um contexto de driver também facilitou o processo da compilação cruzada da biblioteca, descrito na seção seguinte.

A compilação da imagem inicial foi feita por meio de uma imagem padrão do Buildroot, contendo uma versão base preparada para a placa Intel Galileo. Para criá-la, deve-se executar os seguintes comandos na pasta principal do Buildroot:

```
$ make galileo_defconfig
$ make
```

Trecho de Código 5.79: Comandos para compilação da imagem para a placa pelo Buildroot.

Para o propósito de execução na placa, também foi habilitada a opção de criar uma imagem do sistema de arquivos no formato CPIO, que pode ser lido pelo kernel. Para isso,

no Buildroot, deve-se ativar a opção *cpio the root filesystem (for use as an initial RAM filesystem)*, dentro do menu *Filesystem images*, na interface gráfica do `menuconfig`.

Além disso, este arquivo referente ao sistema de arquivos da nova versão do Linux foi compactada com o formato `bzip2`. Após habilitar a criação do arquivo CPIO, fica disponível uma nova opção no menu *Filesystem images*, chamada *Compression method*. Nesta, deve-se selecionar a opção `bzip2`.

Com isso, são gerados dois arquivos que serão necessários para a inicialização do sistema – a imagem, `bzImage`, e o sistema de arquivos, `rootfs.cpio.bz2`, ambos os quais ficam disponíveis no diretório `output/images` do Buildroot.

Então, é necessário também um terceiro arquivo, `grub.conf`, responsável por inicializar o dispositivo em si. Este arquivo de configuração, que é usado para definir a lista de sistemas operacionais que podem ser inicializados na interface de menu do GRUB, essencialmente permite ao usuário selecionar um conjunto de comandos pré-definidos para executar. No caso, este inicializa a imagem do kernel com o sistema de arquivos correspondente.

O conteúdo deste é descrito no seguinte trecho de ccódigo:

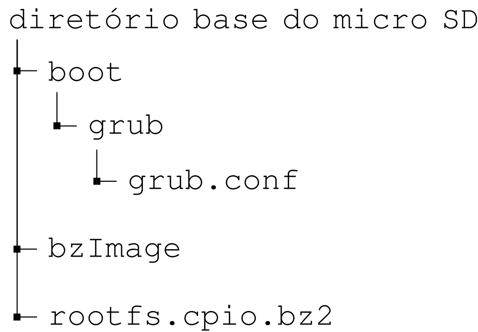
```
default 0
timeout 3

title Linux on Intel Galileo
root (hd0,0)
kernel /bzImage console=ttyS1,115200n8
   earlycon=uart8250,mmio32,0x9000b000,115200n8 reboot=efi,warm
   apic=debug rw
initrd /rootfs.cpio.bz2
```

Trecho de Código 5.80: Conteúdo do arquivo `grub.conf`.

É importante notar, em particular, a definição do kernel (`kernel /bzImage`) e do sistema de arquivos inicial, no modo de compressão CPIO (`initrd /rootfs.cpio.bz2`).

Os três arquivos devem, então, ser adicionados a um cartão microSD formatado no formato FAT32. A estrutura dos arquivos deve ser a seguinte:



Com isso, ao se ligar a placa com o micro SD inserido, ela deverá inicializar imediatamente na imagem criada.

5.8.4 Compilação cruzada da libspdm para a versão do Linux customizada

Para realizar a compilação cruzada da libspdm, é importante primeiramente considerar que a placa Intel Galileo possui um processador do tipo i586. Assim sendo, a compilação deve ser realizada com esta arquitetura em mente.

Com isso em mente, nota-se que o arquivo CMakeLists.txt da libspdm, responsável por executar os comandos do CMake e especificar as *flags* de compilação, precisa ser alterado. Por ter o Buildroot no mesmo sistema, é possível utilizar os binários gerados na seção anterior – o *linker* e o gcc, por exemplo – para realizar a compilação, garantindo a funcionalidade da biblioteca.

Assim, alterou-se o arquivo relevante da biblioteca, adicionando uma opção referente ao Buildroot como *toolchain*. Nesta, devem ser definidos os seguintes parâmetros:

```
if(TOOLCHAIN STREQUAL "BUILDROOT")
   SET(CMAKE_C_COMPILER i586-buildroot-linux-uclibc-gcc)
   SET(CMAKE_AR i586-buildroot-linux-uclibc-ar)
   SET(CMAKE_LINKER i586-buildroot-linux-uclibc-ld)
```

Trecho de Código 5.81: Novas adições ao arquivo CMakeLists.txt.

São necessárias também adições às *flags* de compilação. Para evitar maiores transtornos, foram utilizadas as mesmas *flags* utilizadas pelo Buildroot para compilar os códigos do próprio kernel. Com a posse destas, elas podem ser adicionadas ao CMakeLists.txt com o comando SET(CMAKE_C_FLAGS <flags>).

Para identificar os binários gerados pelo Buildroot, é necessário também adicionar o

caminho para eles na variável de ambiente PATH, o que pode ser realizado pelo seguinte comando:

```
export PATH=<path-to-buildroot>/output/host/bin/:$PATH
```

Trecho de Código 5.82: Definição do caminho para os binários do Buildroot na variável PATH.

No qual <path-to-buildroot> corresponde ao caminho ao diretório base do Buildroot. Então, a compilação pode ser realizada a partir do diretório base da libspdm pelos seguintes comandos:

```
mkdir build_galileo
cd build_galileo
cmake -DARCH=ia32 -DTOOLCHAIN=BUILDROOT -DTARGET=Release
   -DCRYPTO=mbedtls ..
make copy_sample_key
make
```

Trecho de Código 5.83: Comandos para a compilação da libspdm.

Por fim, é necessário adicionar a libspdm ao kernel previamente compilado. Este processo é realizado por meio da cópia dos arquivos necessários da libspdm, desenvolvidos pelos professores do LARC, como citado na Seção 5.4, com algumas pequenas alterações necessárias para a execução na versão específica do kernel preparado para a placa Intel Galileo. Então, realiza-se a recompilação do kernel do Linux gerado, especificando-se o caminho para a pasta na qual a libspdm foi compilada para a placa.

Terminadas estas etapas, o kernel está pronto para receber o código em C, que cumprirá as funcionalidades necessárias do dispositivo USB.

5.8.5 Implementação do código em C do dispositivo SPDM

O código do dispositivo C em si foi iniciado com o estudo das possibilidades de criação de um dispositivo USB com o uso da interface USB Raw Gadget, desenvolvida por xairy, disponível em seu repositório no GitHub [34].

Esta é uma interface de baixo nível para o subsistema USB Gadget do Linux, podendo ser usado para emular dispositivos USB físicos com hardware especial, ou virtuais (para o kernel em que está rodando) com o Dummy HCD/UDC, que define um *Host Controller*

Driver, ou HCD, que é essencialmente um driver para um dispositivo genérico, baseado no *chip UDC*, ou *User-Defined Configuration*, que indica que as funcionalidades em si devem ser definidas em um código separado do próprio Dummy.

Especificamente, a interface definida pelo Dummy HCD/UDC configura dispositivos USB virtuais e controladores Host que estão conectados um ao outro dentro do kernel, permitindo também conectar dispositivos USB do espaço do usuário ao kernel subjacente através de qualquer uma das interfaces do subsistema Gadget, como Raw Gadget e GadgetFS [35].

O repositório conta com exemplos de uso desta interface, o que facilita o processo de entendimento. Em particular, o código de exemplo referente à ativação de um teclado com esta interface foi extremamente útil para a inicialização da codificação.

Em primeiro lugar, é importante notar o formato da solução a ser desenvolvida. A árvore do kernel deve ser modificada, de maneira a introduzir os códigos relevantes e compilá-los com o auxílio do Buildroot, de maneira similar à realizada na Seção 5.4.

Com isso em mente, tem-se a seguinte estrutura de arquivos:

```
diretório base
└─ drivers
   └─ usb
      ├─ Makefile
      ├─ dummy_hcd
      │  ├─ Makefile
      │  └─ dummy_hcd.c
      └─ spdm_gadget
         ├─ Makefile
         └─ spdm_gadget.c
```

Entre os arquivos citados, vale notar que o `Makefile` dentro do diretório `usb` contém apenas instruções para adicionar os novos diretórios a serem compilados. Já dentro dos dois diretórios, ambos devem conter uma instrução para a compilação de seus respectivos arquivos `.c`, enquanto que o `spdm_gadget`, que utilizaria funções da libspdm, deve conter também instruções para a inclusão dos diretórios nos quais se localizam os arquivos relevantes da biblioteca, como visto no seguinte bloco de código.

```
SPDM_INCLUDE := -Iinclude/spdm -Iinclude/spdm/hal
```

```
CFLAGS_spdm_gadget.o += $(SPDM_INCLUDE)
```

Trecho de Código 5.84: Conteúdo do `Makefile` para inclusão dos diretórios da libspdm.

Além disso, é importante ressaltar também que o `dummy_hcd.c` é idêntico ao apresentado no repositório USB Raw Gadget, sendo que o código a ser desenvolvido estará contido no arquivo `spdm_gadget.c`.

Então, utilizando-se como base os exemplos disponíveis no repositório da interface USB Raw Gadget, pôde-se extrair algumas conclusões quanto ao funcionamento esperado de um dispositivo que utiliza o Dummy HCD/UDC como base.

Primeiramente, o Raw Gadget é aberto por meio de uma chamada ao arquivo `/dev/raw-gadget` interagindo diretamente com a interface padrão de Raw Gadget do Linux. Então, o Dummy é instanciado por meio de uma chamada à função `ioctl` com a *flag* `USB_RAW_IOCTL_INIT`, especificando o nome do dispositivo, `"dummy_udc.0"`, e o nome do driver, `"dummy_udc"`. A mesma função `ioctl` é utilizada, então, para de fato executar a instância do Raw Gadget com a passagem da *flag* `USB_RAW_IOCTL_RUN`.

No código do teclado, este processo acontece nas funções `usb_raw_open()`, `usb_raw_init()` e `usb_raw_run()`, como indicado no trecho de código a seguir.

```
int usb_raw_open() {
   int fd = open("/dev/raw-gadget", O_RDWR);
   // [...]
   return fd;
}

void usb_raw_init(int fd, enum usb_device_speed speed,
         const char *driver, const char *device) {
   struct usb_raw_init arg;
   strcpy((char *)&arg.driver_name[0], driver);
   strcpy((char *)&arg.device_name[0], device);
   arg.speed = speed;
   int rv = ioctl(fd, USB_RAW_IOCTL_INIT, &arg);
   // [...]
}

void usb_raw_run(int fd) {
   int rv = ioctl(fd, USB_RAW_IOCTL_RUN, 0);
```

```
    // [...]
}
```

Trecho de Código 5.85: Inicialização de dispositivo USB

Estas funções são, então, chamadas na função principal do código, definindo também o nome do dispositivo e do driver referente ao Dummy, como visto no trecho de código a seguir.

```
int main(int argc, char **argv) {
    const char *device = "dummy_udc.0";
    const char *driver = "dummy_udc";
    int fd = usb_raw_open();
    usb_raw_init(fd, USB_SPEED_HIGH, driver, device);
    usb_raw_run(fd);

    // [...]
}
```

Trecho de Código 5.86: Inicialização de dispositivo USB

As funcionalidades do dispositivo em si devem ser criadas nas funções `ep0_loop` e `ep_int_in_loop`, que contêm *loops* nos quais eventos relevantes ao dispositivo acontecem repetidamente.

Estas funções também realizam a leitura e escrita de dados, podendo também dialogar diretamente com o driver com o qual o próprio dispositivo se comunica.

Infelizmente, este processo não pôde ser concluído a tempo das entregas finais deste trabalho. Dessa forma, as funcionalidades previstas para o dispositivo físico não puderam ser finalizadas.

5.9 Finalização da implementação

Após todo este processo, foi desenvolvido um sistema composto principalmente por dois elementos: o driver USB e o dispositivo emulado pelo QEMU. Estes trabalham em conjunto, de tal forma que qualquer inconsistência no firmware do sistema *host* ou falha na comunicação SPDM inativa o sistema como um todo, garantindo, assim, a segurança do sistema a nível de firmware.

Em seguida, foi desenvolvido também um protótipo físico, que utilizou a placa Intel® Galileo como plataforma para a criação de um dispositivo USB que substituiria o dispositivo emulado pelo QEMU, buscando cumprir seus mesmos requisitos de funcionamento.

Por fim, vale ressaltar que o código desenvolvido pode ser encontrado no repositório do GitHub do projeto [36]. Em seguida, são realizados testes com os protótipos desenvolvidos, além de uma avaliação dos resultados obtidos.

6 TESTES E AVALIAÇÃO

Neste capítulo, detalha-se os testes realizados e os resultados obtidos, comparando-os com os resultados esperados.

6.1 Testes de Segurança do dispositivo emulado

Quanto aos **requisitos de segurança**, o dispositivo emulado pelo QEMU atinge todos os objetivos estipulados na especificação do projeto.

O uso da máquina virtual com o driver e o dispositivo inserido é normal, sem grandes prejuízos de desempenho, e a VM pode ser utilizada como se não houvesse alterações para o usuário médio.

Neste caso de uso padrão, o tempo de execução dos códigos desenvolvidos foi guardado com o auxílio do comando `dmesg`, presente em quase qualquer distribuição do Linux, que permite a visualização do buffer completo do kernel, incluindo mensagens de depuração do sistema. Para garantir o funcionamento dos códigos executados, o `dmesg` foi largamente utilizado para ter certeza de que os dados a serem enviados estavam corretos.

O uso do `dmesg` pelo GDB, no entanto, permite também a exibição de marcações de tempo em segundos de cada mensagem. Esta ferramenta foi incrivelmente útil para o grupo neste quesito, facilitando a visualização do tempo de execução de cada seção do fluxo normal do sistema. Os tempos de execução observados também foram relativamente consistentes depois de centenas de execuções, indicando que a funcionalidade continua consistente, sem evidências de falha potencial.

Em todos os casos executados, o tempo de execução das primeiras verificações foi de aproximadamente um segundo, como pode-se observar pela diferença entre os tempos indicados nas Figuras 15 e 16.

Entre os casos de uso mais específicos para os quais o desenvolvimento foi voltado, pode-se citar inicialmente a remoção do dispositivo USB enquanto o sistema *host* está

```
[     4.235980] urandom_read: 4 callbacks suppress
[     4.236020] random: dd: uninitialized urandom
[     6.278414] SPDM device found on attempt 1
[     6.278670] spdm_context size: 0x12c10
[     6.279806] spdm_send_spdm_request[0] (0x4):
[     6.280000] 0000: 10 84 00 00
[     6.280324] Sending SPDM request with size 5
```

Figura 15: Logs do início da execução do fluxo normal do sistema.

Fonte: do autor.

```
[    7.290858] Generated TOTP 427462 with steps 27725358
[    7.290949] Generated TOTP 837700 with steps 27725359
[    7.291098] Generated TOTP 558505 with steps 27725360
[    7.291287] TOTP 837700 matches the expected value.
[    7.291418] All SPDM checks realized successfully.
[   22.766876] random: crng init done
(gdb)
```

Figura 16: Logs da inicialização da comunicação SPDM.

Fonte: do autor.

sendo executado. Neste caso, o resultado esperado era que o sistema ficasse inoperável, ou fosse até desligado.

Para realizar este processo na emulação pelo QEMU, foi utilizada a linha de comando do monitor do QEMU, que permite um controle mais granulado da máquina virtual, incluindo ações que não são possíveis apenas pelo terminal usual da VM. Em particular, foi passado o comando `device_del totp-spdm`, o que essencialmente remove o dispositivo da máquina virtual.

O resultado final deste caso foi a inviabilização do sistema. Ele continua operando em *background*, de modo que ainda é possível extrair informações acerca de seu funcionamento pelo GDB, mas o usuário não consegue mais realizar ação alguma além de desligar o sistema, como indicado na Figura 17.

```
[    7.132810] All SPDM checks realized successfully.
[   27.041518] random: crng init done
[   37.197093] usb 1-1: USB disconnect, device number 2
[   37.201182] usb_totp_spdm_disconnect
[   37.201528] Shutting down system...
```

Figura 17: Logs da remoção do dispositivo USB.

Fonte: do autor.

Outro caso de uso importante é a execução do sistema *host* sem o dispositivo USB estar inserido na máquina. A simulação desse caso é simples, bastando não adicionar o

tipo de dispositivo ao iniciar a máquina virtual do QEMU.

Aqui, nota-se que há um período de *timeout* modificável, após o qual o dispositivo deverá entrar em um estado de uso inviabilizado. A adição do dispositivo antes desse *timeout* ser atingido deve permitir o uso normal da máquina.

Como esperado, o resultado final deste caso foi, novamente, a inviabilização do sistema, como visto na Figura 18

```
[    4.084295] random: dd: uninitialized urandom read (512 bytes read)
[   47.873691] SPDM device not found!
[   47.874043] Shutting down system...
[   47.875894] sd 0:0:0:0: [sda] Synchronizing SCSI cache
[   47.928298] sd 0:0:0:0: [sda] Stopping disk
[   47.936634] reboot: System halted
(gdb)
```

Figura 18: Logs da inicialização do sistema sem o dispositivo.

Fonte: do autor.

De maneira similar, o caso em que o dispositivo é adicionado antes do sistema atingir o *timeout* inicializa o sistema normalmente. No QEMU, isso pode ser realizado utilizando novamente o monitor do QEMU, e utilizando o comando `device_add usb-totp-spdm,bus=usb-bus.0,id=totp-spdm`. Este comando essencialmente cria o dispositivo emulado, adicionando-o à máquina virtual como se o dispositivo tivesse sido simplesmente plugado no mesmo momento.

Neste caso de uso, o sistema foi, de fato, inicializado normalmente, como pode ser visto nas Figuras 19 e 20. Nelas, nota-se em particular que o dispositivo foi encontrado no instante 31.01 segundos, com o início da comunicação SPDM ocorrendo em 32.68 segundos. A partir desse ponto, a execução do caso ocorre sem maiores diferenças do fluxo de uso padrão.

```
[    4.199701] urandom_read: 4 callbacks suppressed
[    4.199742] random: dd: uninitialized urandom read (512 bytes read)
[   31.015173] usb 1-1: new full-speed USB device number 2 using uhci_hcd
[   31.224172] usb_totp_spdm_probe
[   31.224528] TOTP + SPDM USB Driver 1-1:1.0: USB Driver Probed: Vendor ID : 0x666,    Product ID : 0x666
[   31.225329] USB_INTERFACE_DESCRIPTOR:
[   31.225582] ----------------------------
```

Figura 19: Logs da adição do dispositivo USB após *boot*.

Fonte: do autor.

Um último teste, exemplificando os efeitos que ocorreriam caso uma inconsistência com a máquina *host* seja encontrada, pode ser simulada pelo QEMU.

```
[   32.683293] SPDM device found on attempt 6
[   32.684141] spdm_context size: 0x12c10
```

Figura 20: Logs da inicialização do SPDM após adição do dispositivo USB.

Fonte: do autor.

Neste teste, a execução do sistema emulado é parada com o auxílio do GDB, deixando seu relógio interno dessincronizado com o relógio do dispositivo.

Como esperado, o TOTP, que gera códigos a partir de uma marcação de tempo atual, não gera códigos iguais em ambos os lados da comunicação, ocasionando um erro que inviabiliza o uso da máquina *host*, como ilustrado pela Figura 21.

```
[   49.167451] Generated TOTP 350138 with steps 27735415
[   49.167754] Generated TOTP 633612 with steps 27735416
[   49.168524] Generated TOTP 495601 with steps 27735417
[   49.169565] TOTP 518258 did not match the expected value.
[   49.170391] Shutting down system...
[   49.174907] sd 0:0:0:0: [sda] Synchronizing SCSI cache
[   49.215064] sd 0:0:0:0: [sda] Stopping disk
[   49.216911] reboot: System halted
(gdb)
```

Figura 21: Logs da dessincronização da máquina *host* e do dispositivo USB.

Fonte: do autor.

Nota-se que esse caso simula, por exemplo, um ataque no qual uma mudança no firmware da máquina *host* é realizada, ou um ataque que ocasiona uma dessincronização dos relógios mais significativa.

6.2 Testes de desempenho do dispositivo emulado

Já em termos de **desempenho**, nota-se que o sistema desenvolvido não afeta de maneira considerável o desempenho da máquina *host*. A principal execução do driver ocorre unicamente dentro de uma kernel thread do sistema, de tal modo que o desempenho nunca é muito afetado.

Pelas marcações temporais do GDB, listados anteriormente, a execução do fluxo inicial demora um segundo, no qual a thread de execução possui um uso relativamente alto. Esta execução, no entanto, deve ser realizada apenas no momento de *boot*, e demorando apenas um segundo não consiste em uma perda relevante de desempenho.

Já quanto às avaliações periódicas, nota-se que a presença do TOTP, gerando códigos

de verificação que são enviados por meio do protocolo SPDM, agiliza o tempo de execução deste processo em até **99%**.

Isso se dá pela simplicidade de execução do fluxo de troca de mensagens TOTP pela encriptação com SPDM. Um teste adicional, no qual a comunicação SPDM era estabelecida periodicamente desde o início, gerava uma perda considerável de desempenho, na qual o uso da kernel thread específica do driver era alto durante todo o período de execução, de aproximadamente um segundo, como foi visto anteriormente.

Por outro lado, a adição deste fluxo com TOTP demora aproximadamente 1 centésimo de segundo para ser executado, no caso em que a criação de uma nova chave não é necessária. Ainda neste "pior caso", o tempo de execução chega a até 2 centésimos de segundo, o que ainda representa um **ganho de aproximadamente 98%** neste quesito. A Figura 22 ilustra este processo, lembrando que os tempos informados pelo `dmesg` do GDB são todos expressos em segundos.

Utilizando a ferramenta htop para determinar estatísticas mais específicas do uso do driver, pode-se extrair mais informações que comprovam o bom funcionamento do sistema desenvolvido. Utilizando-se o atalho `Shift + K`, pode-se ativar a verificação de kernel threads no sistema no qual o aplicativo está sendo executado, permitindo a análise mais minuciosa do desempenho.

Ao se executar o sistema por grandes períodos de tempo, nota-se que o tempo de execução ativa total da thread é baixíssimo. Por este aplicativo, pode-se determinar que, em média, após 20 minutos de execução, a kernel thread não atingia um único segundo de execução ativa, exibindo um uso total de tempo de CPU de aproximadamente **0.083%**. A Figura 23 ilustra este resultado, indicado pelos valores de `Uptime`, indicando o tempo total de execução do sistema, e `TIME+`, indicando o tempo de execução ativa de cada kernel thread. A thread referente ao driver sendo executado pode ser encontrada na terceira linha com ID de processo 37, sob o nome `kworker/u2:2+totp_spdm`

Em termos de uso da CPU durante a execução, o sistema desenvolvido também apresentou bons resultados. Em seu auge de uso durante as verificações periódicas, o sistema atinge apenas **0.7%** de uso máximo, como visto na Figura 24. Na imagem, vale ressaltar que outras kernel threads também possuem alguma influência do driver SPDM e TOTP, como o `/sbin/syslogd`, entretanto é mais difícil categorizar quanto da execução deste é referente ao código sendo executado, e quanto é em decorrência da execução normal do sistema. Mesmo considerando todos estes processos laterais, mesmo durante o pico das verificações periódicas, o uso não chega a 3% de uso da CPU.

```
[  132.802629] All SPDM checks realized successfully.
[  174.186382] Initializing periodic SPDM checks.spdm_send_spdm_request[ffffffff] (0x6):
[  174.188524] 0000: 7f 00 00 00 00 00
[  174.189487] Sending SPDM request with size 55
[  174.191669] spdm_request_sent completed
[  174.193687] original size: 4608
[  174.196069] size_dec: 63
[  174.196091] header_size = 4
[  174.196638] totp_spdm_usb_struct->response_data:
[  174.197685] 06 FF FF FF FF 05 00
[  174.197729] 36 00 64 E1 FF BE 7F 66
[  174.198182] 32 1D 34 6C 5B CB 0D 25
[  174.198905] 96 F3 B6 9D CC 58 D9 F5
[  174.199376] DC C7 E8 78 02 E4 F3 50
[  174.199888] C2 1E F3 C6 CB 1E DC E7
[  174.200347] 3F 3C 20 76 59 C1 3C CD
[  174.200799] 11 56 FB 7F B7 99 F8 6B
[  174.201356] spdm_response_done completed
[  174.207929] Received SPDM response with size 63
[  174.209309] SPDM response:
[  174.209657] 06 FF FF FF FF 05 00
[  174.209696] 36 00 64 E1 FF BE 7F 66
[  174.210500] 32 1D 34 6C 5B CB 0D 25
[  174.210969] 96 F3 B6 9D CC 58 D9 F5
[  174.211434] DC C7 E8 78 02 E4 F3 50
[  174.211892] C2 1E F3 C6 CB 1E DC E7
[  174.212203] 3F 3C 20 76 59 C1 3C CD
[  174.212519] 11 56 FB 7F B7 99 F8 6B spdm_receive_spdm_response[ffffffff] (0x7):
[  174.213665] 0000: 7f 62 64 63 32 63 00
[  174.214112] totp_dec: 777260
[  174.214359] totp_key:
[  174.214576] E1 E8 E7 9F 74 6A 54
[  174.214601] F7 C3 30 8B 5F 5E DD 3C
[  174.214966] 4A 7C A0 B3 C5 C9 F0 28
[  174.215273] DD AF C6 09 89 DD DD 94
[  174.215578] 2F 4F 1C B3 A5 A5 93 B0
[  174.215884] D7 BD DA CB 6E F1 82 CA
[  174.216273] 61
[  174.216618] Generated TOTP 316216 with steps 27735411
[  174.216809] Generated TOTP 834376 with steps 27735412
[  174.217147] Generated TOTP 777260 with steps 27735413
[  174.217507] TOTP 777260 matches the expected value.
(gdb)
```

Figura 22: Logs das verificações periódicas com SPDM.

Fonte: do autor.

```
QEMU [Paused]
Machine   View

CPU[                                   0.0%]   Tasks: 7, 0 thr, 34 kthr; 1 running
Mem[|#*                           10.1M/993M]   Load average: 1.08 1.02 0.75
Swp[                                  0K/0K]   Uptime: 00:19:12

PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
139 root       20   0  1712  1280  1064 R  0.7  0.1  0:13.48 htop
 23 root       20   0     0     0     0 I  0.0  0.0  0:00.70 kworker/0:1-events_power_efficient
 37 root       20   0     0     0     0 D  0.0  0.0  0:00.91 kworker/u2:2+totp_spdm
 60 root       20   0  1776   200   160 S  0.0  0.0  0:00.36 /sbin/syslogd -n
 10 root       20   0     0     0     0 I  0.0  0.0  0:00.47 rcu_sched
  9 root       20   0     0     0     0 S  0.0  0.0  0:00.17 ksoftirqd/0
 64 root       20   0  1772   204   160 S  0.0  0.0  0:00.44 /sbin/klogd -n
  1 root       20   0  1784   192   148 S  0.0  0.0  0:00.89 init
  2 root       20   0     0     0     0 S  0.0  0.0  0:00.00 kthreadd
```

Figura 23: Teste de desempenho do driver desenvolvido por meio do htop.

Fonte: do autor.

```
CPU[#**                                          2.7%]   Tasks: 7, 0 thr, 34 kthr; 1 running
Mem[|#*                                     10.1M/993M]   Load average: 1.00 0.98 0.68
Swp[                                             0K/0K]   Uptime: 00:16:35

PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 60 root       20   0  1776   176   132 S  1.3  0.0  0:00.33 /sbin/syslogd -n
135 root       20   0  1712  1300  1080 R  0.7  0.1  0:09.60 htop
 23 root       20   0     0     0     0 I  0.7  0.0  0:00.55 kworker/0:1-events_power_efficient
 37 root       20   0     0     0     0 D  0.7  0.0  0:00.83 kworker/u2:2+totp_spdm
```

Figura 24: Teste de desempenho do driver desenvolvido por meio do htop.

Fonte: do autor.

Por fim, pode-se considerar também o tempo de espera da CPU em operações de leitura e escrita, que permite a definição exata de quanto tempo o sistema passa enviando dados, e não processando-os. Como pode-se notar, no entanto, as transmissões periódicas são sempre muito pequenas, possuindo número de bytes da ordem de grandeza de uma centena.

A ferramenta htop permite a visualização detalhada do uso da CPU, indicando também o tempo de espera com operações de I/O, que ajudará a visualizar esta informação. O teste com a ferramenta no auge da execução pode ser vista na Figura 25

```
CPU[**$                                          3.2%]   Tasks: 7, 0 thr, 38 kthr; 1 running
Mem[|#*                                     10.1M/993M]   Load average: 0.97 0.45 0.18
Swp[                                             0K/0K]   Uptime: 00:02:56
                                                          CPU: 0.0% sy: 2.6% ni: 0.0% hi: 0.0% si: 0.6% wa: 0.0%

PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
136 root       22   2  1724  1344  1128 R  3.9  0.1  0:02.54 htop
 23 root       20   0     0     0     0 I  0.6  0.0  0:00.18 kworker/0:1-events_power_efficient
 60 root       20   0  1776   200   160 S  0.6  0.0  0:00.20 /sbin/syslogd -n
```

Figura 25: Teste de desempenho do driver desenvolvido por meio do htop.

Fonte: do autor.

Da figura, pode-se extrair que o tempo de espera, determinado pela porcentagem indicada no campo `wa:`, a direita, pode ser aproximada a 0%, indicando um tempo baixíssimo de espera para as operações de I/O. Dessa forma, conclui-se que as transmissões realizadas depois do *boot* são todas rápidas, e ocorrem por meio da transferência de pouquíssimos bytes.

6.3 Testes e avaliação do dispositivo físico

Levando-se em consideração os resultados obtidos com o uso da placa Intel Galileo, pode-se considerar que, embora não tenha sido possível finalizar o desenvolvimento de todas as funcionalidades planejadas de início, o progresso do grupo ainda foi satisfatório, faltando em particular a última tarefa proposta.

Ao final, a placa utilizada possui as seguintes funcionalidades:

- Configuração como dispositivo USB OTG;
- Imagem customizada do Linux criada com o Buildroot;
- libspdm disponível a nível de kernel, inserida dentro da imagem customizada.

Com estas três grandes tarefas concluídas, bastaria a criação do código em C referente às funcionalidades desejadas do SPDM e do TOTP. Embora pareça uma tarefa demorada, esta já havia sido inicializada na etapa do desenvolvimento do dispositivo USB emulado pelo QEMU.

O dispositivo emulado já cumpre todas as funcionalidades desejadas para o projeto. Para estas serem aplicadas em um ambiente físico, no entanto, o código deveria ser adicionado à placa Galileo.

O código do dispositivo foi feito com o auxílio de funções próprias do QEMU, incluindo, por exemplo, funções que gerenciam os pacotes de dados USB (os blocos de requisição denominados URBs), e conseguem copiar os arquivos para os pacotes USB do QEMU, além de extrair os dados relevantes dos mesmos pacotes quando são enviados do driver para o dispositivo. Dessa forma, seria necessário adaptar estas funções ao novo ambiente, gerenciando o pacote inteiramente no próprio código.

Além deste detalhe, no entanto, seria possível copiar todas as mesmas funcionalidades do código original, e estes deveriam funcionar, já que a plataforma na qual eles são executados são similares, com ambos fazendo parte do kernel de versões embarcadas do Linux.

Infelizmente, no entanto, atrasos no projeto impediram o grupo de seguir com esta última etapa do desenvolvimento das funcionalidades principais. Estes atrasos surgiram, em particular, durante a etapa de criação da imagem do Linux embarcado com a compilação cruzada da biblioteca libspdm, como pode-se ver nas Seções 5.8.3 e 5.8.4.

O desenvolvimento deste código seria realizado como indicado na Seção 5.8.5, baseando-se na interface USB Raw Gadget. A maior dificuldade que o grupo encontrou durante este desenvolvimento foi quanto ao uso das funções presentes no código da interface USB Raw Gadget.

Como dito pelo autor, o código foi criado com a versão `5.7` do kernel em mente, mas é compatível com versões até a `4.14`. O kernel utilizado, no entanto, gerado a partir da

imagem padrão da placa Galileo no QEMU, é baseado na versão `3.14`. Dessa maneira, a compilação do Dummy gerou vários erros referentes ao uso de funções que o kernel utilizado pelo grupo não encontrava, como pode ser visto na Figura 26.

```
include/linux/log2.h:22:1: warning: ignoring attribute ‘noreturn’ because it conflicts with attribute ‘con
st’ [-Wattributes]
   22 | int ____ilog2_NaN(void);
      | ^~~
drivers/usb/dummy_hcd/dummy_hcd.c:131:27: error: field ‘caps’ has incomplete type
  131 |  const struct usb_ep_caps caps;
      |                           ^~~~
drivers/usb/dummy_hcd/dummy_hcd.c:144:3: error: implicit declaration of function ‘USB_EP_CAPS’ [-Werror=im
plicit-function-declaration]
  144 |   USB_EP_CAPS(USB_EP_CAPS_TYPE_CONTROL, USB_EP_CAPS_DIR_ALL)),
      |   ^~~~~~~~~~~
drivers/usb/dummy_hcd/dummy_hcd.c:136:11: note: in definition of macro ‘EP_INFO’
  136 |   .caps = _caps, \
      |           ^~~~~
drivers/usb/dummy_hcd/dummy_hcd.c:144:15: error: ‘USB_EP_CAPS_TYPE_CONTROL’ undeclared here (not in a func
tion)
  144 |   USB_EP_CAPS(USB_EP_CAPS_TYPE_CONTROL, USB_EP_CAPS_DIR_ALL)),
      |               ^~~~~~~~~~~~~~~~~~~~~~~~
drivers/usb/dummy_hcd/dummy_hcd.c:136:11: note: in definition of macro ‘EP_INFO’
  136 |   .caps = _caps, \
      |           ^~~~~
drivers/usb/dummy_hcd/dummy_hcd.c:144:41: error: ‘USB_EP_CAPS_DIR_ALL’ undeclared here (not in a function)
  144 |   USB_EP_CAPS(USB_EP_CAPS_TYPE_CONTROL, USB_EP_CAPS_DIR_ALL)),
      |                                         ^~~~~~~~~~~~~~~~~~~
drivers/usb/dummy_hcd/dummy_hcd.c:136:11: note: in definition of macro ‘EP_INFO’
  136 |   .caps = _caps, \
      |           ^~~~~
drivers/usb/dummy_hcd/dummy_hcd.c:147:15: error: ‘USB_EP_CAPS_TYPE_BULK’ undeclared here (not in a functio
n)
  147 |   USB_EP_CAPS(USB_EP_CAPS_TYPE_BULK, USB_EP_CAPS_DIR_IN)),
      |               ^~~~~~~~~~~~~~~~~~~~~
```

Figura 26: Erros encontrados devido à diferença nas versões do kernel utilizado.

Fonte: do autor.

Este erro impediu o progresso do grupo na semana final de desenvolvimento.

Com isso, dadas as funcionalidades presentes no dispositivo emulado pelo QEMU, seria possível criar os códigos relevantes caso o grupo tivesse algum tempo a mais para a criação do projeto.

7 CONCLUSÃO

O projeto consistiu na integração de duas tecnologias já existentes, o SPDM e o TOTP, com o intuiro de criar uma solução de segurança a nível de firmware mais robusta do que as dispníveis atualmente no mercado. Dessa forma, visando impedir e dificultar ataques prejudiciais e que podem até mesmo serem indetectáveis.

A segurança de sistemas a nível de firmware mostrou-se um tema promissor como área de desenvolvimento e estudo, possibilitando a criação de novas tecnologias, as quais podem utilizar conceitos totalmente novos, mas também fazer a integração com conceitos mais estabelecidos, mesmo que em estágios iniciais, como foi o caso do projeto desenvolvido.

7.1 Contribuições

Dentre as principais contribuições desse trabalho, destacam-se: a implementação do protocolo SPDM em uma aplicação real, visto que o protocolo está ainda em sua infância e não tem ainda muitos usos concretos; a aplicação em si, pois ainda não existe no mercado uma chave física de autenticação para sistemas que utilize o protocolo SPDM, que tem a vantagem de ser resistente a ataques a nível de firmware; e, por último, a implementação conjunta com TOTP, o que melhora a eficiência de verificações continuadas da conexão do *dongle* com o sistema, além de garantir uma camada temporal à segurança da conexão.

Tendo isso em mente, as contribuições apresentadas pelo projeto mostram que é possível a implementação de uma aplicação real que utiliza o protocolo SPDM, em conjunto com o TOTP, de forma a obter um grande aumento do nível de segurança em troca de uma queda baixíssima em termos de desempenho, tanto no momento do boot do sistema quanto durante sua operação.

7.2 Trabalhos Futuros

Como não foi possível finalizar todas as tarefas planejadas, trabalhos futuros baseados neste projeto podem ter como guia finalizar a adaptação do código do dispositivo USB para a placa Intel Galileo. Os códigos desenvolvidos para este trabalho encontram-se no repositório do GitHub do projeto [36], permitindo, assim, a continuação dos códigos relevantes a partir do estado no qual o grupo terminou o desenvolvimento.

Outra direção que poderia ser tomada é a implementação de comunicação Bluetooth entre o sistema *host* e o dispositivo USB. Além de permitir uma maior flexibilidade no uso do dispositivo como ferramenta de segurança, visto que não é mais necessário estar conectado fisicamente ao sistema *host*, a prototipagem da utilização do protocolo SPDM via Bluetooth pode ser utilizada em uma variedade de aplicações onde a segurança do ambiente sem fio é desejável, como em sistemas de pagamento sem contato ou em sistemas de monitoramento de saúde. Além disso, a combinação do Bluetooth e do SPDM também permite que os dispositivos se conectem de forma fácil e rápida, sem a necessidade de configurações complicadas ou senhas difíceis de lembrar.

REFERÊNCIAS

1 CUI, A.; COSTELLO, M.; STOLFO, S. When firmware modifications attack: A case study of embedded exploitation. In: NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM. New York, US: The Internet Society, 2013.

2 BASNIGHT, Z. H. *Firmware counterfeiting and modification attacks on programmable logic controllers*. Ohio, US, 2013.

3 CHOI, B.-C. et al. Secure firmware validation and update for consumer devices in home networking. *IEEE Transactions on Consumer Electronics*, v. 62, n. 1, p. 39–44, 2016.

4 VIEW, M. et al. *TOTP: Time-Based One-Time Password Algorithm*. RFC Editor, 2011. RFC 6238. (Request for Comments, 6238). Disponível em: <https://www.rfc-editor.org/info/rfc6238>.

5 BASÍLIO, L.; ROJA, L.; BOGER, M. *Integração do Security Protocol and Data Model ao Kernel do Linux*. 54 f. Monografia (TCC (Graduação)) — Engenharia de Computação, Escola Politécnica da USP, São Paulo, 2021.

6 GUTTMAN, B.; ROBACK, E. A. *SP 800-12. An Introduction to Computer Security: The NIST Handbook*. Gaithersburg, MD, USA, 1995.

7 STALLINGS, W.; BROWN, L. *Computer Security: Principles and Practice*. 3rd. ed. USA: Prentice Hall Press, 2014. ISBN 0133773922.

8 SAMONAS, S.; COSS, D. The cia strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information System Security*, v. 10, n. 3, 2014.

9 DISTRIBUTED MANAGEMENT TASK FORCE. *Security Protocol and Data Model (SPDM) Specification*. Oregon, US, 2021. Ver. 1.2.0.

10 PLANK, J. Perspectives in security measurement utilizing the dmtf security protocol and data model (spdm). *Platform Security Summit*, 10 2019.

11 KRAWCZYK, D. H.; BELLARE, M.; CANETTI, R. *HMAC: Keyed-Hashing for Message Authentication*. RFC Editor, 1997. RFC 2104. (Request for Comments, 2104). Disponível em: <https://www.rfc-editor.org/info/rfc2104>.

12 VIEW, M. et al. *HOTP: An HMAC-Based One-Time Password Algorithm*. RFC Editor, 2005. RFC 4226. (Request for Comments, 4226). Disponível em: <https://www.rfc-editor.org/info/rfc4226>.

13 ONELOGIN. *What's the Difference Between OTP, TOTP and HOTP?* California, US, 2022.

14 UMAWING, J. Has two-factor authentication been defeated? a spotlight on 2fa's latest challenge. *Malwarebytes Labs*, 6 2019.

15 LOVE, R. *Linux Kernel Development*. 3rd. ed. Boston, US: Addison-Wesley Professional, 2010. ISBN 0672329468.

16 QEMU: A generic and open source machine emulator and virtualizer. 2022. Disponível em: <https://www.qemu.org/>. Acesso em: 06 abr. 2022.

17 QEMU. *qemu*. França: GitHub, 2022. <https://github.com/qemu/qemu>.

18 BUILDROOT: Making Embedded Linux Easy. 2022. Disponível em: <https://buildroot.org/>. Acesso em: 06 abr. 2022.

19 DMTF. *libspdm*. Oregon, US: GitHub, 2022. <https://github.com/DMTF/libspdm>.

20 GDB: The GNU Project Debugger. 2022. Disponível em: <https://www.sourceware.org/gdb/>. Acesso em: 30 jun. 2022.

21 NETTHAW. *TOTP-MCU*. Japão: GitHub, 2022. <https://github.com/Netthaw/TOTP-MCU>.

22 LEURENT, G.; PEYRIN, T. SHA-1 is a shambles: First chosen-prefix collision on SHA-1 and application to the PGP web of trust. In: *29th USENIX Security Symposium (USENIX Security 20)*. California, US: USENIX, 2020. p. 1839–1856.

23 WILLOUGHBY, M. Sha-1 flaw seen as no risk to one-time password proposal. *Computerworld*, 05 2005.

24 SHA-1 & HMAC OTP Frequently Asked Questions. 2015. Disponível em: <https://openauthentication.org/wp-content/uploads/2015/09/FAQ1.pdf>. Acesso em: 25 ago. 2022.

25 DUTHOMHAS. *CSPRNG*. New Mexico, US: GitHub, 2022. <https://github.com/Duthomhas/CSPRNG>.

26 KROAH-HARTMAN, G. *Writing USB Device Drivers*. 2002. Disponível em: <https://www.kernel.org/doc/htmldocs/writing_usb_driver/index.html>. Acesso em: 26 ago. 2022.

27 TORVALDS, L. *Linux kernel*. Helsinki, Finlândia: Linux, 2022. <https://www.kernel.org/>.

28 INTEL. *Intel® Galileo Datasheet*. California, US, 2013.

29 TORVALDS, L. *Mass Storage Gadget (MSG)*. Helsinki, Finlândia: Linux, 2022. <https://www.kernel.org/doc/Documentation/usb/mass-storage.txt>.

30 YOCTO Project – It's not an embedded Linux distribution – it creates a custom one for you. 2022. Disponível em: <https://www.yoctoproject.org/>. Acesso em: 22 nov. 2022.

31 DOCKER: Accelerated, Containerized Application Development. 2022. Disponível em: <https://www.docker.com/>. Acesso em: 22 nov. 2022.

32 KOEMAN, K. *Understanding USB On-The-Go*. 2001. Disponível em: <https://www.edn.com/understanding-usb-on-the-go/>. Acesso em: 22 nov. 2022.

33 10 Images — The Yocto Project ® 4.1.999 documentation. 2022. Disponível em: <https://docs.yoctoproject.org/ref-manual/images.html>. Acesso em: 04 dez. 2022.

34 XAIRY. *USB Raw Gadget — low-level interface for the Linux USB Gadget subsystem*. [S.l.]: GitHub, 2022. <https://github.com/xairy/raw-gadget>.

35 TORVALDS, L. *USB Raw Gadget*. [S.l.]: Linux, 2022. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/usb/raw-gadget.rst>.

36 LUCASTYPH. *Buildroot module for TOTP + SPDM integration*. São Paulo, Brasil: GitHub, 2022. <https://github.com/LucasTyph/TOTP-SPDM-buildroot-module>.

APÊNDICE A - EXEMPLO DE SCRIPT DE *DAEMON* DE UM MÓDULO EXTERNO

```
#!/bin/sh
#
# Start driver at boot
#

case "$1" in
  start)
    echo "Starting TOTP SPDM driver..."
    modprobe totp_spdm_driver
    ;;
  stop)
    modprobe -r totp_spdm_driver
    ;;
  restart|reload)
    ;;
  *)
    echo "Usage: $0 {start|stop|restart}"
    exit 1
esac

exit $?
```

Trecho de Código 7.1: Exemplo de script de *daemon*.