Otávio Felipe de Freitas

Avaliação de Desempenho de Um Sistema Computacional com O Padrão SPDM

São Paulo, SP

2023

Otávio Felipe de Freitas

Avaliação de Desempenho de Um Sistema Computacional com O Padrão SPDM

Trabalho de conclusão de curso apresentado ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro.

Universidade de São Paulo – USP

Escola Politécnica

Departamento de Engenharia de Computação e Sistemas Digitais (PCS)

Orientador: Prof. Dr. Marcos Simplicio Jr

Coorientador: Prof. Dr. Bruno Carvalho Albertini

São Paulo, SP

2023

Agradecimentos

Os agradecimentos principais são direcionados à minha família e aos meus amigos pelo apoio oferecido e aos orientadores, Prof. Dr. Marcos Antonio Simplicio Junior e Prof. Dr. Bruno de Carvalho Albertini, pelos ensinamentos.

...

Resumo

A atual conjuntura mundial oferece possibilidades de invasões devido aos diversos fabricantes que interagem entre si durante a produção de componentes eletrônicos. Esse cenário abre brechas para ataques maliciosos contra o sistema privado e público, pois as verificações de modificações indevidas não são suficientes. O padrão Security Protocol and Data Model (SPDM) pretende oferecer uma alternativa de proteção, contudo ainda não foi amplamente testada e implementada. Este projeto objetiva aplicar o protocolo em *drivers* do Linux e executá-los por meio da emulação de hardware, além de apresentar uma análise da sobrecarga em um sistema computacional em comparação aos métodos similares, objetivando incentivar seu uso no meio de segurança de componentes. O padrão SPDM foi inserido no Das U-Boot e implementado no driver do dispositivo de blocos virtual, caso a mídia inserida não seja autenticada com sucesso, o dispositivo não pode ser utilizado pelo firmware. A sobrecarga imposta no sistema computacional implica em um tempo de inicialização do firmware em 10 vezes e as trocas de mensagens através do canal seguro de comunicação durante a inicialização, em até 13 vezes.

Palavras-chave: Segurança de Computadores, Protocolo de Comunicação, Mitigação de Risco, Hardware, Firmware, RISC-V, SPDM.

Abstract

The current world scenario offers possibilities of invasions due to numerous manufacturers which interact among themselves during electronic components production. Thus, it opens security holes to malicious attacks aiming at private and public systems, given that the available verification methods for unauthorized modifications are not sufficient. The standard Security Protocol and Data Model (SPDM) intends to offer a protection alternative, however, it is not widely tested and implemented. This project has the objective of applying the protocol in Linux drivers and executing them utilizing hardware emulation, additionally, it intends to present an analysis of the overhead on the computer system in comparison to similar methods to promote its use in the components security field. The SPDM standard was inserted in Das U-Boot and implemented on the virtual block device, in case the connected media isn't successfully authenticated, the device can not be used by the firmware. The overhead imposed on the computer system implies in a firmware initialization time 10 times greater and the messages exchanges through the secure communication channel during the boot process, at worst 13 times greater.

Keywords: Computer Security, Communication Protocol, Risk Mitigation, Hardware, Firmware, RISC-V, SPDM.

Lista de ilustrações

Sumário

1 Introdução

Neste capítulo é introduzido e contextualizado o tema do trabalho, apresentando sua motivação devido a acontecimentos e iniciativas atuais no âmbito de segurança da informação. Em seguida, apresenta-se o objetivo a ser atingido descrevendo sucintamente o que esta pesquisa propõe-se a concretizar, finalizando com a justificativa da relevância em empregar esforços na área de segurança na infraestrutura de TI.

1.1 Motivação

As relações econômicas no contexto do presente trabalho permitem que diversos países participem de uma complexa cadeia de suprimentos, na qual os componentes de hardware são fabricados e montados em territórios diferentes dos quais os contratantes estão localizados[1]. Esse cenário abre brechas para ataques em nível de hardware e, portanto, maneiras efetivas de proteger tanto as empresas quantos os consumidores são aplicadas (MALAJ; MARINOVA, 2020). Ataques em nível de hardware incluem roubo de propriedade intelectual, modificação, degradação de performance, espionagem, etc. e as mitigações são compostas por criptografia, técnicas de prevenção a vazamento de dados, contramedidas durante o design de chips engenharia reversa, etc. Entretanto, em sistemas embarcados, por possuírem recursos limitados, a aplicação dessas medidas de proteção danificam a performance.

Todas as peças de hardware em sistemas computacionais modernos são de fato um sistema embarcado, o que implica na presença de um software chamado firmware. O firmware é responsável por tarefas que controlam dispositivos de hardware físicos, o que possibilita sua operação correta (e.g. implementa protocolos de comunicação). A modificação de um firmware permite ataques tanto lógicos como físicos contra qualquer outro componente que dependa do seu bom funcionamento (BASNIGHT, 2013).

Ataques maliciosos ao firmware implicam ataques no nível de componente. Esse tipo de ataque é considerado de difícil detecção, agravado quando realizado seletivamente (e.g. o componente se comporta como o esperado, mas também possui um comportamento malicioso em algumas situações). Em comparação com ataques de software realizados no nível de sistema operacional, como os vírus, o ataque no nível de componente passa despercebido por qualquer software de proteção tradicional (e.g.firewalls e antivírus)(CHOI et al., 2016).

Existem na literatura alguns trabalhos que visam abordar tais ataques de modifica-

[1] https://www.nytimes.com/2022/09/01/business/tech-companies-china.html

ção de firmware (WANG et al., 2015; CHOI et al., 2016; BASNIGHT, 2013). Buscando técnicas para mitigar os ataques maliciosos, a organização DMTF – *Distributed Management Task Force* – propôs o padrão SPDM – *Security Protocol and Data Model* (DMTF, 2022), o qual não possui uma implementação sólida e amplamente testada até o momento deste trabalho. Esse padrão é aberto e define mensagens, objetos de dados e sequências para construir um canal de mensagens seguro entre dispositivos, objetivando a autenticação de hardware e firmware.

Segurança em nível de firmware torna-se ainda mais relevante ao considerar o número de dispositivos de internet das coisas, segundo a matéria Sinha (2023) em um documento de maio de 2023, haviam 14,4 bilhões de aparelhos conectados em 2022, com previsão de 16,7 bilhões ao final de 2023. Ademais, seguindo o crescimento de IoT, ocorre fomento no investimento de segurança nesses aparelhos. Em 2023, estima-se o investimento de 7,4 bilhões de dólares e projeta-se para 2028, 9,8 bilhões de dólares (Research and Markets, 2023).

Em um relatório confeccionado pelo time de segurança da Microsoft em 2021 (Microsoft Security Team, 2021), foi realizado uma pesquisa online com 1000 indivíduos envolvidos em decisões de segurança em países como EUA, Reino Unido, Alemanha, China e Japão. Nesse documento publicado, 83% das empresas responderam que sofreram ao menos um ataque em nível de firmware entre 2019 e 2020, entretanto relata-se que apenas 29% do orçamento de segurança é alocado para a proteção de software.

1.2 Objetivo

O padrão SPDM não possui uma implementação sólida e amplamente testada até o momento deste trabalho, portanto, por meio do código aberto de referência da organização DMTF – LibSPDM[2] –, objetiva-se verificar que a sobrecarga oferecida, desde o *boot* até a totalidade do funcionamento, no sistema computacional é similar aos métodos alternativos com o mesmo nível de segurança.

1.3 Justificativa

A LibSPDM pretende ser uma implementação compatível com diferentes plataformas, capaz de impactar ambientes corporativos e públicos oferecendo segurança para os hardwares e firmwares adquiridos. Uma vez implementado e provado sua eficácia no que é proposto, um novo método de mitigação de ataques maliciosos pode ser utilizado, oferecendo mais de um tipo de mitigação de ataques com apenas uma técnica de proteção de firmware.

[2] https://github.com/DMTF/libspdm

Visto a crescente relevância de segurança de embarcados, especialmente a nível de firmware (SINHA, 2023; Research and Markets, 2023; Microsoft Security Team, 2021), este trabalho propõe a implementar e analisar um padrão com poucos números de adoção até 2023. Percebe-se ainda, por outro relatório de segurança da Microsoft, que vários dispositivos embarcados, 57% deles, estão sujeitos a mais de dez vulnerabilidades CVE, essas que estão disponíveis há mais de 10 anos (Microsoft Security Team, 2023).

Logo, compreende-se que a segurança a nível de firmware é negligenciada com frequência, abrindo portas para ataques. É necessário maior conscientização quanto a esse tópico, assim como mais medidas de proteção comprovadamente eficazes, sendo este o objetivo deste trabalho.

1.4 Organização do Trabalho

Este trabalho inicia-se com a explicação dos principais conceitos para compreender o desenvolvimento, destacando o funcionamento do padrão SPDM no capítulo, além de incluir a discussão de trabalhos relacionados em 2. O método do trabalho é apresentado no capítulo 4, detalhando os passos, os critérios e as métricas para avaliação da implementação. No capítulo 3, os requisitos a serem atendidos pelo desenvolvimento são descritos. Em seguida, no capítulo 5 todas as atividades que foram realizadas durante o ano de 2023 são apresentadas, com foco na implementação do padrão SPDM dentro do firmware, possuindo uma seção para a análise dos resultados obtidos. Por fim, o capítulo 6 conclui o documento ressaltando os resultados atingidos, as imperfeições, as contribuições e a perspectiva de trabalhos futuros.

2 Aspectos Conceituais

2.1 Confidencialidade, Integridade e Autenticidade

Os três serviços principais que a segurança da informação oferece são chamados de tríade CIA, são eles: confidencialidade, integridade e disponibilidade. Esses termos foram padronizados no documento ISO/IEC 27000, iniciado em 2009 e atualmente na versão 2018. Contudo, com o surgimento de novas necessidades para a área, alguns outros serviços foram adicionados: autenticidade e não retratabilidade. Neste trabalho, os conceitos de confidencialidade, integridade e autenticidade são os serviços propostos. Logo, conforme ISO Central Secretary (2018), confidencialidade é a propriedade da informação não estar disponível ou não ser apresentada para indivíduos, entidades ou processos não autorizados; integridade é a propriedade de acurácia e completude; autenticidade é a propriedade de uma entidade ser o que alega.

O padrão SPDM possui a capacidade de fornecer a um firmware confidencialidade por meio da sessão segura, a qual é feita no final da negociação de chaves, podendo trocar mensagens criptografadas entre os pontos comunicantes. A autenticidade é oferecida com a análise da cadeia de certificados e assinaturas com *GET_DIGESTS*, *GET_CERTIFICATE* e *CHALLENGE*. Por fim, o último serviço encontra-se na fase de *GET_MEASUREMENTS*, na qual os hashes dos firmwares dos dispositivos podem ser comparados com uma lista de softwares e hardwares permitidos.

2.2 Cadeia de Certificados

Uma cadeia de certificados é uma lista ordenada com uma entidade final que pode incluir um ou mais certificados de uma autoridade certificadora, sendo a entidade final o certificado raiz de uma CA e cada certificado seguinte é emitido pelo anterior. Ao checar a identidade de cada certificado da cadeia por meio da verificação das assinaturas, é possível averiguar se eles são confiáveis (AKRAM et al., 2020).

O padrão SPDM aceita três modelos de cadeias de certificados: certificado de dispositivo, certificado por *alias* e certificado genérico (DMTF, 2022). O certificado de dispositivo é o modelo comum de uma cadeia de certificados, no qual uma autoridade certificadora raiz assina os certificados de possíveis certificados de autoridades intermediárias, que, por fim, assina o certificado folha (o último da cadeia). O certificado por *alias* é semelhante, porém o dispositivo torna-se uma autoridade e assina certificados abaixo dele, nesse caso o par de chaves assimétricos podem ser mutáveis e, portanto, devem ser tratado

como tal. Por último, o certificado genérico é semelhante ao certificado de dispositivo, contudo ele é menos restrito quanto a algumas informações presentes no certificado, como a não necessidade de haver elementos que identificam o dispositivo ou o fabricante.

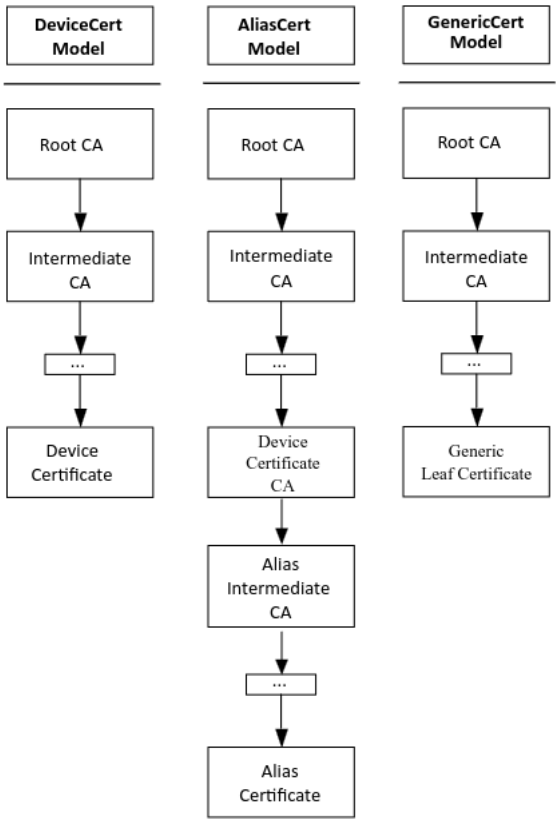


Figura 1 – Modelo de certificados aceitos pelo padrão SPDM. Fonte: DMTF (2022).

2.3 Padrão SPDM

O padrão SPDM, criado em 2019, descreve uma série de troca de mensagens com o intuito de estabelecer um canal seguro de comunicação entre dois pontos – *requester* e *receiver* (DMTF, 2022). Aquele inicia a comunicação com este através da mensagem inicial que objetiva obter a versão SPDM suportada por ambos componentes e, em seguida, diversas outras mensagens são trocadas com a finalidade de configurar o canal de comunicação entre eles – *e.g.* funções implementadas no *receiver*, algoritmos de criptografia disponíveis. As solicitações provenientes do *requester* são classificadas em opcionais ou obrigatórias, a sequência de mensagens é exemplificada pela figura 2.

Para iniciar uma conexão, o *requester* envia a mensagem *GET_VERSION* que também é utilizada para reiniciar uma conexão, mesmo que o *responder* ainda tenha respostas pendentes ao *requester*. A versão da mensagem inicial é 1.0, pois todos os dispositivos que possuem suporte ao padrão SPDM devem ter suporte à primeira versão, em seguida, o *requester* seleciona a maior versão suportada por ambos pontos comunicantes

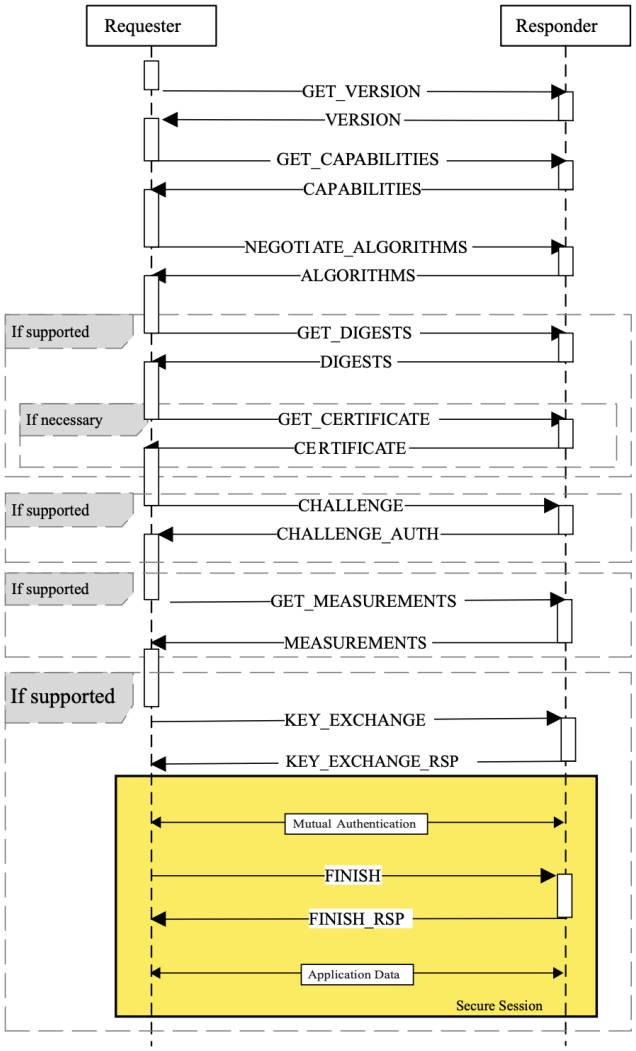


Figura 2 – Troca de mensagens entre dois pontos especificado pelo padrão SPDM. Fonte: DMTF (2022).

[3]. Ademais, o *requester* não pode realizar outras requisições até que receba *VERSION* do *responder*.

Em seguida, o *requester* passa a configurar as capacidades do canal de comunicação com *GET_CAPABILITIES*. A primeira configuração é o tamanho das mensagens que serão trocadas, indicando o *buffer* de transferência de cada dispositivo, assim como o tamanho do *buffer* de montagem de mensagens fragmentadas – *i.e.* quando as mensagens ultrapassam o valor do *buffer* de transmissão, elas são quebradas e remontadas. Outras características são configuradas durante essa fase, *e.g. cache* de configuração caso a conexão seja reiniciada; autenticação mútua; *heartbeat*; atualização de chave da sessão. Mais funcionalidades são descritas na seção 10.3 do documento DMTF (2022).

Após negociar as capacidades da comunicação e, apenas após, negocia-se o algoritmo de criptografia a ser utilizado por meio da mensagem *NEGOTIATE_ALGORITHMS*. Nenhum outro *request* pode ser enviado pelo *requester* até que ele receba uma resposta

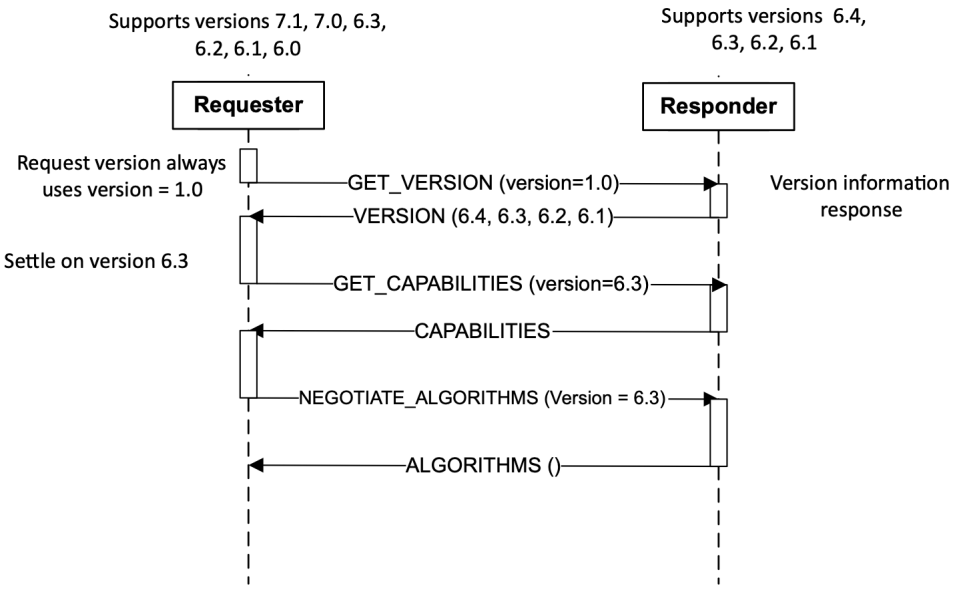


Figura 3 – Troca de mensagens entre dois dispositivos para combinarem a versão do padrão SPDM a ser utilizada. Fonte: DMTF (2022).

ALGORITHMS bem-sucedida.

O próximo passo é a autenticação do dispositivo *responder*, a qual será feita caso esse tenha enviado *CERT_CAP* for um durante a negociação de capacidades. De modo semelhante, o desafio possui suporte se o retorno de *CHAL_CAP* for um. No cenário em que a chave pública do *responder* é fornecida ao *requester* em um ambiente confiável, as mensagens *GET_DIGESTS* e *GET_CERTIFICATE* não são aplicáveis.

O fluxo apresentado em 4 inicia-se com o *requester* enviando *GET_DIGESTS* e o *responder* responde com os hashes, aquele irá procurar em seu cache se possui algum dos hashes recebidos e, caso não possua, manda a requisição *GET_CERTIFICATE*. O *responder* verifica se o *offset* está dentro da cadeia de certificados e, se estiver, envia o tamanho pedido. Pode ocorrer de o tamanho pedido seja maior que o restante a ser enviado, nesse caso os bytes restantes são enviados com valor 0.

A troca de mensagens envolvendo o desafio inicia-se com o envio de *CHALLENGE*, quando o *requester* manda um *nonce*[1] para ser assinado pelo *responder*. Este responde com *CHALLENGE_AUTH* e o *requester* verifica a autenticidade do dispositivo sendo desafiado.

Por fim, para estabelecer a conexão segura, ocorre a troca de chaves públicas entre *requester* e *responder* geradas por esses por meio de um algoritmo criptográfico chamado Diffie-Hellman. Verificações são realizadas para contestar a origem das chaves, com envio de segredos assinados pelas chaves públicas um dos outros. Para cada mensagem *KEY_EXCHANGE*, novos pares de chaves privada e pública são gerados.

[1] Um número arbitrário utilizando apenas uma vez em uma comunicação criptográfica.

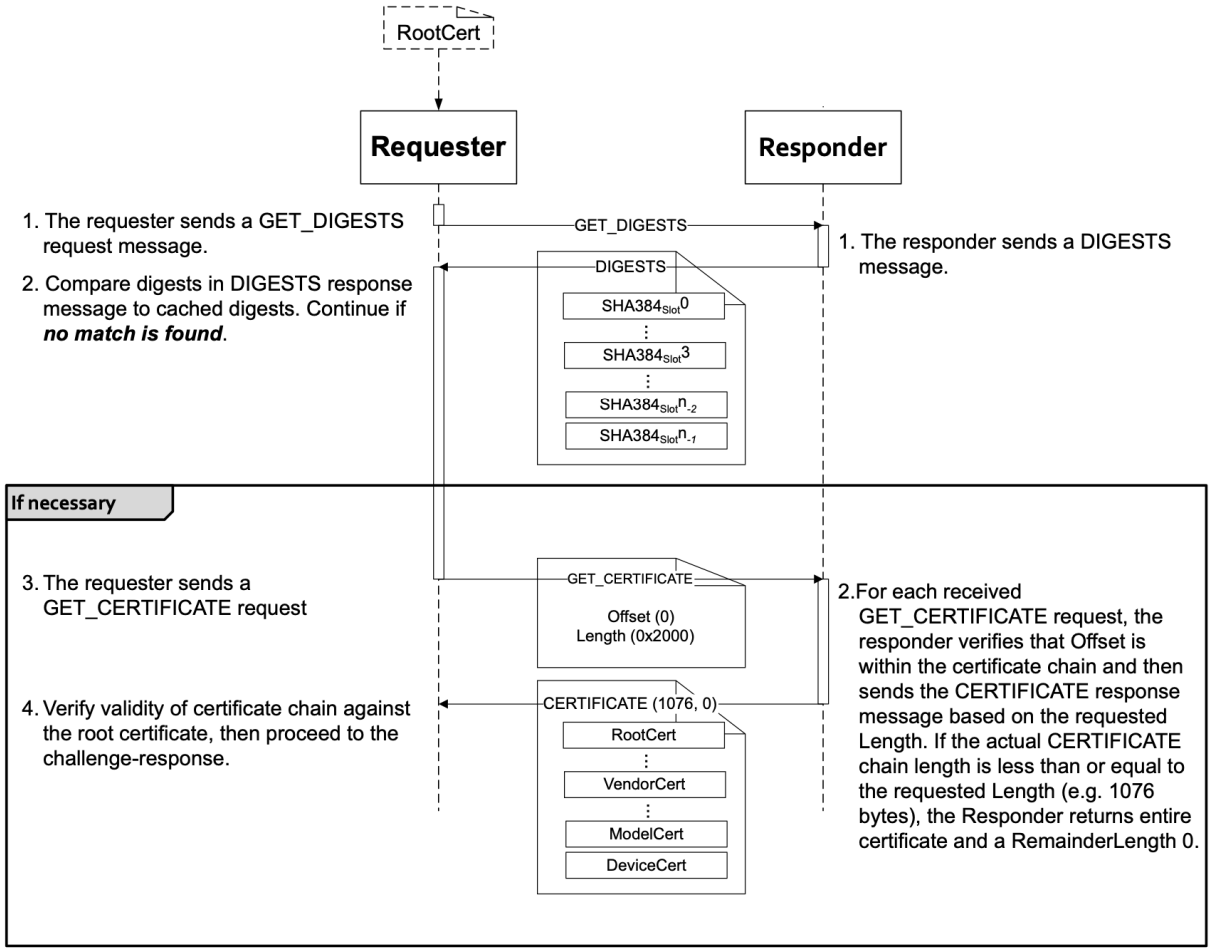


Figura 4 – Fluxo de autenticação do padrão SPDM. Fonte: DMTF (2022).

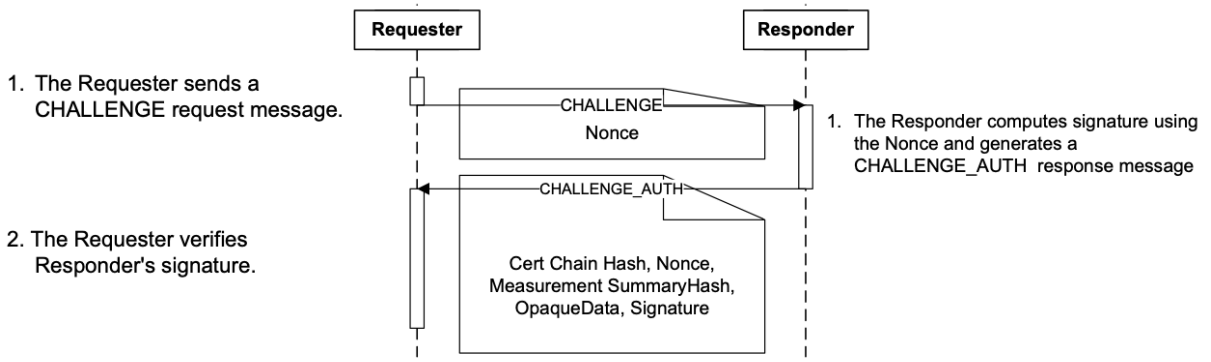


Figura 5 – Fluxo de desafios do padrão SPDM. Fonte: DMTF (2022).

2.4 Trabalhos Relacionados

Uma das maneiras propostas para mitigar a modificação de firmware é por meio da utilização do componente HPC – i.e. *Hardware Performance Counter* –, o qual realiza contagens para avaliar o desempenho. Esse componente é um conjunto de registradores de propósito especial para armazenar informações sobre eventos do hardware, é importante para depurar o desempenho em nível de software e está presente nos microprocessadores

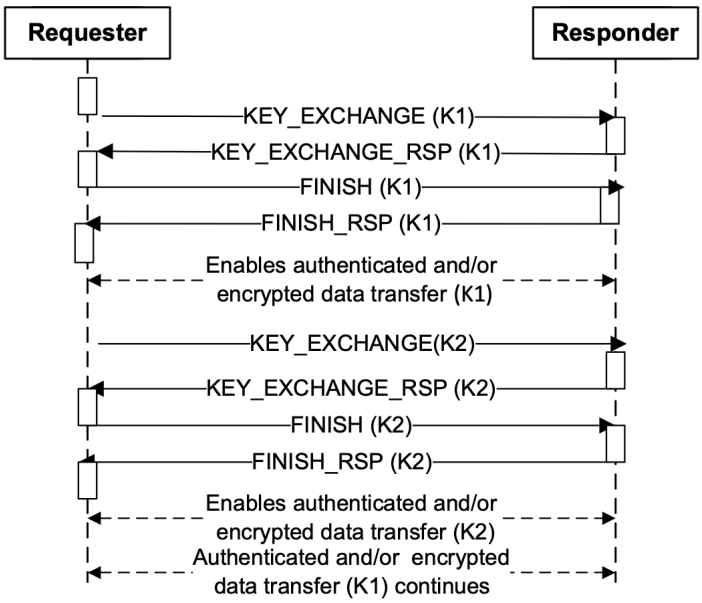
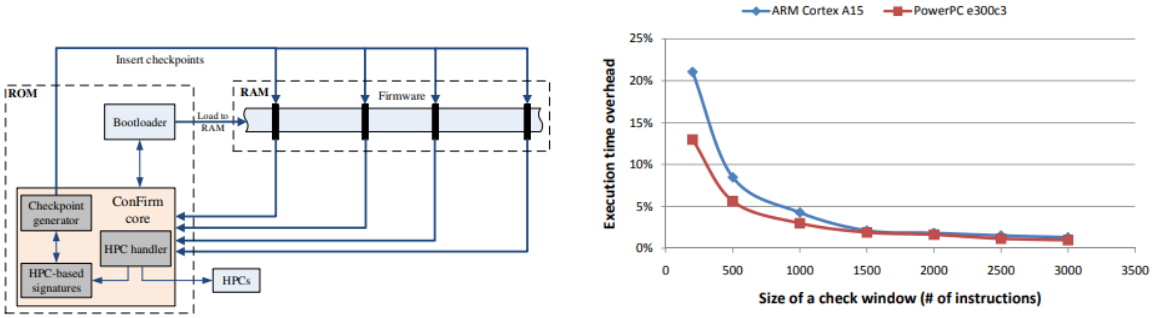


Figura 6 – Fluxo de troca de chaves do padrão SPDM. Fonte: DMTF (2022).

modernos dentro da PMU – i.e. *Performance Monitoring Unit*. Esses registradores podem ser reutilizados para atuar como auxílio em uma solução de segurança ao inserir uma unidade que guarda os valores da contagem do HPC medidos em um *firmware* alvo – i.e. *firmware* que será utilizado como referência aos outros sendo analisados. Essa comparação é feita por meio da inclusão de pontos de checagem na memória RAM do dispositivo, sendo esses pontos a inclusão de saltos para a rotina de verificação da contagem dos eventos até o momento. A figura 7a apresenta a estrutura do método em alto nível, sendo as barras pretas os pontos de checagem inseridos e a contagem de eventos é feita entre esses pontos para, então, serem comparadas com os valores referência armazenados. Esse método apresenta uma baixa sobrecarga no sistema computacional caso o número de eventos do caminho de execução analisado seja alto, contudo, caso o número de eventos seja baixo, a sobrecarga atinge 8.48% para o ARM Cortex A15 e 5.62% para o PowerPC e300c3, mais detalhes são apresentados na figura 7b (WANG et al., 2015).



(a) Estrutura em alto nível do *ConFirm*. (b) Sobrecargas medidas durante os testes.

Figura 7 – Fonte: Wang et al. (2015)

Outra proposta seria a autenticação de firmware por meio da técnica de auten-

ticação mútua e geração de chaves, utilizando, também, uma cadeia de hash aplicada na fragmentação do firmware. O servidor envia o firmware por meio da rede para um gerenciador – i.e. ponto local que realizará a distribuição das atualizações para todos os dispositivos domésticos de uma rede de internet das coisas – , o servidor faz uso de uma chave de sessão para criptografar o firmware e, após a verificação desse pelo gerenciador, ocorre a fragmentação utilizando cadeias de hash. Esses fragmentos serão transmitidos para o dispositivo alvo de atualização por meio de uma sessão e todas as trocas de mensagens entre eles serão criptografadas com uma chave de sessão, sendo essa diferente da chave gerada para a comunicação do servidor com o gerenciador. Por fim, o dispositivo, ao receber todos os fragmentos, verifica a idoneidade do firmware recebido e se atualiza (CHOI et al., 2016).

O padrão SPDM foi alvo de estudo com a inserção do protocolo no driver do HD virtual presente no QEMU e no kernel do Linux, o dispositivo de bloco foi escolhido devido à quantidade de documentação existente para auxiliar no desenvolvimento. Também foi decidido utilizar a LibSPDM como implementação do padrão SPDM, pois é a iniciativa da DMTF para promover o uso do padrão, contudo essa API desenvolvida é para o espaço de usuário do sistema operacional, logo o trabalho teve que modificar parte dos códigos para que fosse possível utilizá-lo no espaço do kernel. Os resultados atingidos foram que um HD virtual com SPDM é, em média, 32% mais lento ao transferir arquivos entre 100 kB e 500 kB, ademais as transferências entre seções do HD – as quais passam pelo sistema operacional duas vezes – são 7 vezes mais lentas (ALVES; ALBERTINI; SIMPLICIO, 2022).

3 Especificação de Requisitos

Com o intuito de testar a hipótese proposta, é necessário possuir o ambiente de desenvolvimento por meio da compilação do *Buildroot* – ferramenta com agregado de *Makefiles* com o intuito de simplificar e automatizar o processo de compilação do Linux com pacotes comuns e sistemas de arquivos, permitindo compilação cruzada – versão 2023.08 com Linux 6.1.26 configurado para RISC-V 64 bits, ademais, a versão do QEMU deve ser maior que 5.1, pois a partir dessa há melhor compatibilidade com a arquitetura selecionada. É necessário selecionar um firmware de código aberto que tenha suporte à arquitetura alvo, além de ser possível emulá-la com o software escolhido para tal feito.

A alteração é composta pela inclusão das trocas de mensagens especificadas no padrão SPDM, utilizando as funções da LibSPDM previamente implementadas e que podem ser acessados por meio da API. A plataforma de emulação e os *drivers* devem trocar mensagens pelo caminho seguro oferecido de maneira transparente desde o *boot*, além de ser possível verificar as mensagens no canal seguro de comunicação a qualquer momento. A versão do padrão SPDM a ser implementada é a 1.1.0, conforme trabalhos prévios (ALVES; ALBERTINI; SIMPLICIO, 2022; BASíLIO; ROJA; BOGER, 2021). Ademais, as funcionalidades existentes anteriores às modificações não devem ser modificadas ou removidas.

4 Método do trabalho

O padrão SPDM, apesar de recente, possui publicações realizadas na EPUSP e esses documentos são o ponto de partida para este trabalho, portanto, como atividade inicial, eles serão estudados. Após a compreensão das atividades já realizadas pelo laboratório, a especificação de requisitos de acordo com o objetivo deve ser realizada antes de começar a implementação de qualquer tecnologia.

Definido a arquitetura alvo, RISC-V 64 bits, os firmwares possíveis precisam ser listados e analisados para, então, decidir qual deles será escolhido para incluir o padrão SPDM. O método de escolha de firmware é a facilidade de utilizá-lo, a existência de documentação, a comunidade e o propósito principal dele. O emulador a ser utilizado, QEMU, deve ser estudado em seguida, pois as alterações que foram realizadas na versão 4.2.0 (ALVES; ALBERTINI; SIMPLICIO, 2022) serão a base para modificações em outras versões, se necessário alterar.

Softwares como o sistema operacional e as ferramentas de compilação cruzada fornecidos pelo Buildroot não precisam ser alterados, nem estudados detalhadamente, pois eles serão objetos de uso, mas não serão modificados. O Linux será utilizado apenas como exemplo de um fluxo de boot completo, entretanto esse SO com o HD virtual modificado por Alves, Albertini e Simplicio (2022) é um exemplo para a implementação que será realizada no firmware, portanto o código do dispositivo de blocos virtual do Linux é o próximo passo.

Após a especificação de requisitos, a decisão do firmware e os estudos subsequentes, a implementação será iniciada. A cada nova funcionalidade, os softwares modificados devem ser testados para garantir conformidade com os requisitos especificados. A implementação é composta pela inserção do padrão SPDM dentro do ambiente do firmware e da criação de uma sessão segura dentro do driver do HD virtual no momento em que esse é reconhecido como um dispositivo válido para boot.

A métrica da sobrecarga será o tempo necessário para obter controle do terminal de um dos firmwares, sendo que a mídia virtual já esteja inicializada e pronta para uso. Quatro medidas temporais serão coletadas para analisar o efeito do padrão SPDM: sem o padrão SPDM, com o padrão SPDM, uma requisição SPDM e várias requisições SPDM.

5 Desenvolvimento do Trabalho

5.1 Escolha de Firmware

SeaBIOS[1], um firmware presente no QEMU e com possibilidade do acesso por meio de flags que atrasam o boot do sistema [5.1] foi considerada inicialmente. Para, possivelmente, alterá-la no futuro, a compilação foi realizada fora do ambiente do emulador e seu binário foi utilizado em conjunto da flag `-bios`.

```
1 $ qemu-system-x86_64 -boot menu=on,splash-time=15000
```

Listing 5.1 – Comando para acessar SeaBIOS.

Outro firmware é a implementação UEFI como referência da Intel, TianoCore EDKII[2] de código aberto. Seu uso, para iniciar um SO, dá-se com um bootloader com capacidade EFI, como o GRUB – *Grand Unified Bootloader* –, um bootloader bastante utilizado em sistemas computacionais com sistema operacional Linux. Nessa imagem de disco, deve haver uma partição de boot com sistema de arquivo EFI cujo arquivo *grubx64.efi* será executado pelo firmware e, então, ocorre a inicialização do sistema computacional por completo. Ademais, há um pacote de aplicações e drivers como exemplo com o padrão SPDM em um dos repositórios disponibilizados pela TianoCore[3], os quais podem ser utilizados como ponto de partida para modificar o driver responsável pelo HD.

Das U-Boot[4], um *bootloader* de primeiro e segundo estágio de código aberto, presente em dispositivos embarcados de várias arquiteturas (ARM, RISC-V, x86, MIPS, etc.). Considerando a arquitetura alvo – RISC-V 64 bits –, Das U-Boot é comumente utilizado como bootloader de segundo estágio, o binário é utilizado como payload de um bootloader primário – o qual, para RISC-V, o OpenSBI é o mais utilizado.

Como x86_64 possui maior número de guias em buscas na internet, os firmwares SeaBIOS e EDKII foram inicialmente testados para essa arquitetura. Uma imagem de disco foi confeccionada utilizando os resultados da compilação do Buildroot para serem executadas no QEMU, a SeaBIOS não reconheceu a mídia com os parâmetros *-drive* do emulador, mas o EDKII aceitou-a. Logo, visto que RISC-V ainda possui menos suporte que x86_64, como a SeaBIOS não apresentou resultados rápido no início, ela foi descartada como escolha. Com o firmware da Intel, foi possível iniciar o Linux e, então, os testes foram para a arquitetura alvo.

[1] https://github.com/coreboot/seabios
[2] https://github.com/tianocore/edk2
[3] https://github.com/tianocore/edk2-staging/tree/DeviceSecurity
[4] https://github.com/u-boot/u-boot

A compilação do EDKII para o RISC-V necessita de outro repositório da TianoCore[5], edk2-platforms, para aceitar mais hardwares. Foi possível compilá-la, mas não houve sucesso em iniciar o SO. Considerando os resultados, a atenção foi voltada para o U-Boot, principalmente por ele ser um firmware comumente utilizado para embarcados e por possuir uma compilação simples. O desafio foi, então, descobrir como iniciar o Linux a partir de seu terminal, porém a escolha limitou-se a ele.

5.2 Configuração do Ambiente

Feito a escolha do firmware, o ambiente de desenvolvimento deve ser preparado para iniciar as modificações. Com o intuito de economizar espaço na memória da máquina que será utilizada, apenas as versões relevantes são clonadas dos repositórios, com exceção da LibSPDM, pois, para manter a compatibilidade com os trabalhos já realizados com ela (ALVES; ALBERTINI; SIMPLICIO, 2022; BASíLIO; ROJA; BOGER, 2021), usa-se a tag de um commit específico.

```
1 $ git clone --depth 1 --branch 2023.08 https://github.com/buildroot/
    buildroot.git
2 $ git clone --depth 1 --branch v2023.07 https://github.com/u-boot/u-boot
    .git
3 $ git clone --depth 1 --branch v6.2.0 https://github.com/qemu/qemu.git
4 $ git clone --depth 1 --branch v1.3 https://github.com/riscv-software-
    src/opensbi.git
```

Listing 5.2 – Comandos para preparar o ambiente de desenvolvimento.

```
1 $ git clone https://github.com/DMTF/libspdm.git
2 $ cd libspdm
3 $ git --checkout dc48779a5b8c9199b01549311922e05429af2a0e
```

Listing 5.3 – Comandos para configurar a LibSPDM.

Dois repositórios possuem submódulos que precisam ser inicializados, são eles: QEMU e LibSPDM. Aquele possui módulos de ROMs (e.g. SeaBios, U-Boot, EDKII), testes de ponto flutuante, *capstone* e *meson*, já este, módulos de testes (cmocka), MbedTLS e OpenSSL. Portanto, basta executar o comando 5.4 dentro dos respectivos diretórios.

```
1 $ git submodule update --init --recursive
```

Listing 5.4 – Inicialização dos submódulos git.

A LibSPDM foi modificada previamente por Alves, Albertini e Simplicio (2022) e está disponível no repositório *spdm-hd-demo*[6]. Os patches vão adicionar duas novas

[5] https://github.com/tianocore/edk2-platforms
[6] https://github.com/rcaalves/spdm-hd-demo/tree/main/libpsdm_patches

toolchains para compilar no Linux – contudo, elas são para x86_64 e serão alteradas –, assim como o header de configuração para o MbedTLS. Para aplicá-los, basta executar os comandos 5.5.

```
1 $ git am -3 --ignore-space-change --keep-cr /path/to/0*.patch
2 $ cp /path/to/config.h /path/to/os_stub/mbedtlslib/include/mbedtls
```

Listing 5.5 – Aplicando modificações na LibSPDM.

5.3 OpenSBI

OpenSBI refere-se a *RISC-V Supervisor Binary Interface*, uma interface que fornece chamadas de função conforme a convenção estabelecida entre o *Supervisor* e o *Supervisor Execution Environment*, i.e. SEE (PATEL, 2019). As chamadas SBI diminuem a existência de códigos repetidos em sistemas operacionais – e.g. Linux, FreeBSD, etc. -; fornecem drivers em comum para um SO, podendo ser compartilhado entre múltiplas plataformas; fornece um meio de acessar diretamente os recursos do hardware.

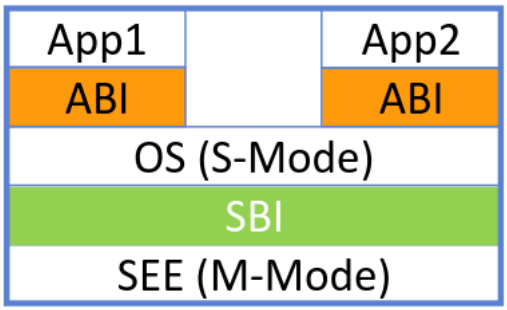


Figura 8 – Estrutura do OpenSBI. Fonte: Patel (2019).

A inicialização utilizando OpenSBI começa com a inicialização do SOC e do clock, ambos com a execução do código presente na ROM do chip, utilizando a SRAM do mesmo. O próximo passo é a inicialização da memória DDR e carregamento do estágio *RUNTIME* (OpenSBI), o qual possui Das U-Boot como payload e esse será o bootloader de último estágio.

Existem três tipos de firmwares referenciáveis para o OpenSBI: *FW_PAYLOAD*, *FW_JUMP* e *FW_DYNAMIC*. O primeiro é o firmware com o próximo estágio de boot

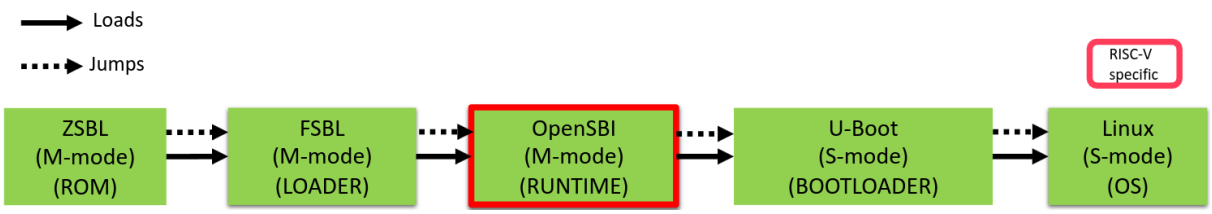


Figura 9 – Exemplo do fluxo de boot na HiFive Unleashed. Os dois primeiros estágios são abstraídos pelo QEMU ao utilizar a máquina *virt*. Fonte: Patra e Patel (2019).

como carga e é padrão para dispositivos RISC-V com capacidade de executar Linux; o segundo possui um endereço fixo de *jump* para o próximo estágio – padrão no QEMU –; o terceiro possui informações dinâmicas sobre o próximo passo de inicialização, utilizado por U-Boot SPL e Coreboot.

Esse trabalho utilizará *FW_PAYLOAD*, sendo a carga o binário do U-Boot. Nenhuma alteração nos códigos do OpenSBI é necessária, pois não é preciso incluir a LibSPDM. Logo, basta gerar o binário *fw_payload.elf* com o comando 5.6.

```
1 CROSS_COMPILE=/path/to/riscv64-linux- PLATFORM=generic FW_PAYLOAD_PATH=/
    path/to/u-boot/u-boot.bin
```

Listing 5.6 – Comando para compilar o OpenSBI utilizando U-Boot como payload.

5.4 Buildroot

O Buildroot será utilizado para obter o Linux, o sistema de arquivos raiz[7] e a *toolchain* para RISC-V 64 bits. O objetivo não é alterar o sistema operacional, portanto a configuração padrão para a arquitetura alvo será utilizada – *i.e.* para o Buildroot 2023.08, a versão do Linux será 6.1.26.

```
1 $ make qemu_riscv64_virt_defconfig
2 $ make
```

Listing 5.7 – Compilação do Buildroot com a configuração padrão para RISC-V 64 bits.

O Linux e o sistema de arquivos estará no diretório *output/images* e a *toolchain*, em *output/host/bin*. Para compilar os softwares necessários – Das U-Boot, OpenSBI e LibSPDM – pode-se incluir o diretório da *toolchain* na variável de ambiente *PATH* ou utilizar o caminho completo.

```
1 $ export PATH=$PATH:/path/to/buildroot/output/host/bin
```

Listing 5.8 – Inclusão do compilador em *PATH*.

5.5 Compilação da LibSPDM

A LibSPDM deve ser compilada para dois cenários diferentes: máquina hospedeira e firmware. Para a máquina hospedeira, nenhuma modificação nos arquivos é necessária, pois basta executar os comandos apresentados em 5.9. Para esse caso, os binários serão utilizados pelo QEMU, o qual é executado em um processador x86_64.

[7] Do inglês root filesystem.

```
$ mkdir build_host
$ cd build_host
$ cmake -DARCH=x64 -DTOOLCHAIN=GCC -DTARGET=Release -DCRYPTO=mbedtls ..
$ make copy_sample_key
$ make
```

Listing 5.9 – Comandos para compilar a LibSPDM para máquina hospedeira.

O segundo cenário é compilado com as mesmas flags que os arquivos do Das U-Boot, pois, caso contrário, acarretará em referências indefinidas ao realizar os links para criar os binários do firmware. Para conseguir as flags utilizadas, pode-se utilizar o comando *make* apresentado em 5.10 ou é possível, caso já tenha compilado pelo menos uma vez, buscar as flags utilizadas nos arquivos *.<nome_do_arquivo>.o.cmd*, em especial *.virtio_blk.o.cmd*, porque é o driver que será alterado.

```
$ make -n <target>
```

Listing 5.10 – Comando que mostra o que será executado, sem executá-los.

A compilação da API é feita com CMake que gera os Makefiles para gerar os binários que serão utilizados, portanto apenas é necessário incluir uma nova opção para Das U-Boot. Como o ambiente de desenvolvimento para esse trabalho é Linux, adiciona-se uma nova *toolchain* chamada *UBOOT* apenas dentro do bloco *if-else* relacionado ao sistema operacional vigente. As flags retiradas da compilação do Das U-Boot é incluída em *CMAKE_FLAGS* e o compilador, definido como *CMAKE_C_COMPILER*, é configurado com *riscv64-linux-gcc* gerado pelo Buildroot. Os testes também são desabilitados devido a incompatibilidades da arquitetura, então define-se a variável *DISABLE_TESTS* e os Makefiles referentes aos testes não são gerados.

```
$ mkdir build_rv64
$ cd build_rv64
$ cmake -DARCH=riscv64 -DTOOLCHAIN=UBOOT -DTARGET=Release -DCRYPTO=
   mbedtls ..
$ make
```

Listing 5.11 – Comandos para compilar a LibSPDM para Das U-Boot.

Os diretórios que são criados nos comandos apresentados em 5.9 e 5.11, *build_host* e *build_rv64*, são para organizar o ambiente. Desse modo, o diretório raiz da LibSPDM não fica poluído com as saídas, além de evitar confusão no momento de realizar os links simbólicos.

5.6 Inclusão da LibSPDM no Firmware

Das U-Boot possui estrutura semelhante ao Linux, composta por vários Makefiles e Kconfigs – os quais são utilizados para construir o menuconfig. Objetivando a facilidade de testes, inclui-se a opção *SPDM* booleana no Kconfig da raiz do firmware, pois, desse modo, ficam dispóníveis as macros *CONFIG_IS_ENABLED(SPDM)* e *CONFIG_SPDM*, as quais podem ser utilizadas conforme o exemplo 5.12.

```
1 #if CONFIG_IS_ENABLED(SPDM)
2 /**
3  * Code to be executed if SPDM option is enabled
4  */
5 #endif
6
7 #ifdef CONFIG_SPDM
8 /**
9  * Code to be executed if SPDM option is enabled
10 */
11 #endif
```

Listing 5.12 – Utilização da macro habilitada pelas opções do Kconfig.

Para incluir a LibSPDM dentro do firmware, é necessário importar a API para dentro da árvore de desenvolvimento[8], portanto, alterações dentro dos diretórios *lib* e *include* são realizadas. Inicialmente, uma linha de código é incluída em *lib/Makefile* para indicar a inclusão de novos arquivos a serem considerados durante a compilação do Das U-Boot [5.13].

```
1 obj-$(CONFIG_SPDM) += spdm/
```

Listing 5.13 – Linha incluída em *lib/Makefile*.

Dentro do diretório *lib*, cria-se um novo chamado *spdm* e adiciona-se o arquivo *spdm_glue.c* o qual verificará a presença de modificações nos binários da LibSPDM, observando a data de alteração. Em conjunto do arquivo em C, coloca-se o Makefile responsável por realizar os links simbólicos dos binários dentro do diretório *spdm*, esses são renomeados com o sufixo *_spdmlib*, e por copiar os *headers* para dentro de *include/spdm*, desse modo as funções da API podem ser utilizadas dentro do firmware.

A LibSPDM faz uso da função de hash *sha256*, descrita dentro do Mbed-TLS e presente no arquivo *lib/sha256.c*, acarretando em conflito na função *sha256_update*. Como ocorre problema em apenas uma função, escolheu-se ignorá-la caso a opção SPDM esteja ativada no arquivo de configuração do Das U-Boot [5.14].

[8] Do inglês *in-tree development*.

```
#ifndef CONFIG_SPDM
void sha256_update(sha256_context *ctx, const uint8_t *input, uint32_t
    length)
{
/**
 * Function here
 */
}
#endif
```

Listing 5.14 – Solução para o conflito de *sha256_update*.

Para que os algoritmos de hash e negociação de chaves funcione, deve-se ativar as opções presentes na configuração do U-Boot. Pode-se realizar tal ato com o comando 5.15 e, em seguida, ativando as seguintes configurações:

- Library Routines
  - Hashing Support
    - Enable SHA256 Support
  - Security Support
    - Use RSA Library

```
$ make menuconfig
```

Listing 5.15 – Comando para acessar o menu de configuração do Das U-Boot.

Para que o U-Boot não inicialize automaticamente, desabilita-se, dentro do menu em *Boot options* e *Autoboot options*, a opção *Autoboot*. Além disso, adiciona-se uma nova opção em *preboot default value*, também em *Boot options*, o comando *virtio device 0*, esse irá selecionar o HD virtual como dispositivo atual a ser utilizado. O comando pré-inicialização não é necessário, porém escolhe-se realizar essa mudança para autenticar o dispositivo de bloco logo antes de deixar o terminal disponível para o usuário, figura 10.

5.7 Inclusão da LibSPDM no QEMU

O trabalho de Alves, Albertini e Simplicio (2022) utilizou o software de emulação na versão 4.2.0, contudo, pensando em trabalhos futuros, a versão deve ser atualizada, pois, por exemplo, a emulação da placa de rede Intel E1000 utilizada por Basílio, Roja e Boger (2021) não é compatível com RISC-V 64 bits. A versão escolhida para atualizar o QEMU foi 6.2.0, porque essa é a padrão do Ubuntu 20.04 e, portanto, é suficientemente estável para ser modificada.

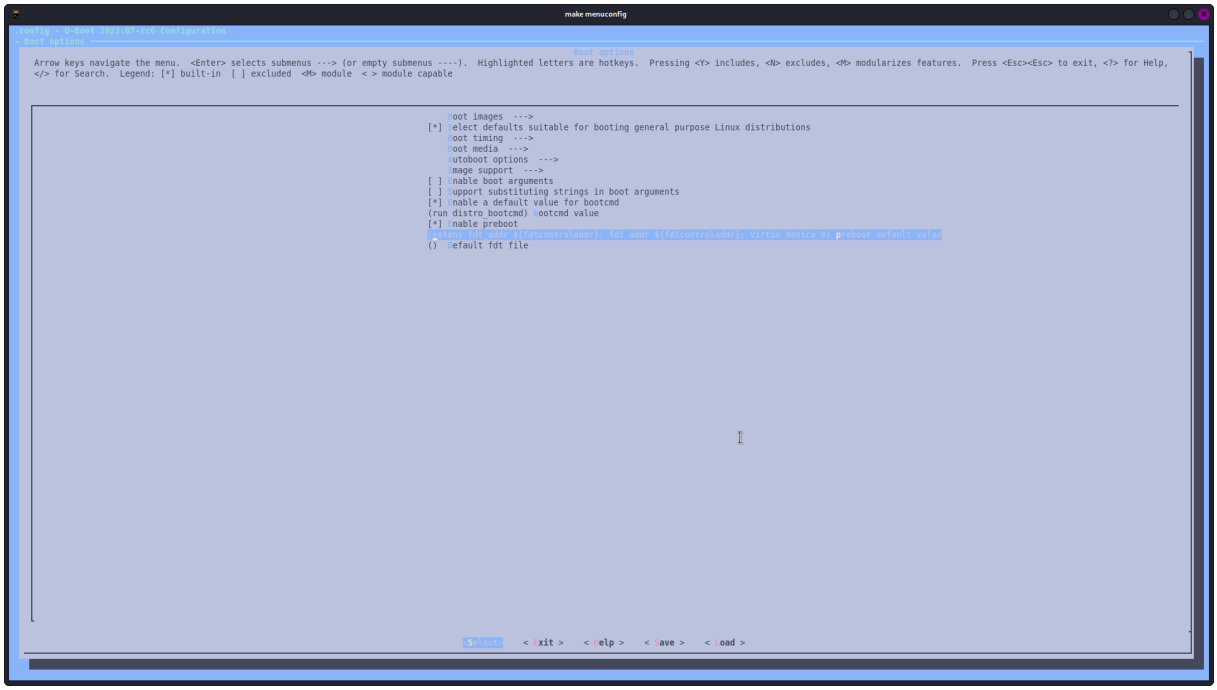


Figura 10 – Menu de configuração do U-Boot, dentro de *Boot options* com o cursor selecionando a opção *preboot default value*.

Em relação à versão anterior, houve mudança na estrutura dos arquivos do QEMU. Anteriormente, o arquivo *configure* era responsável por configurar as variáveis, as flags de compilação, as flags de link, os diretórios observados e os arquivos que serão compilados, porém em versões recentes há a adição do sistema de compilação Meson – uma alternativa ao Make e ao CMake com o objetivo de ser rápido e com baixa curva de aprendizado[9].

O arquivo *configure* ainda é modificado para adicionar as opções de configuração que incluem a LibSPDM a qual, assim como o firmware, deve buscar os *headers* e os binários para realizar links estáticos. As novas variáveis de *configure* são impressas em *config-host.mak* e podem ser acessadas por meio de *meson.build*. Há, também, o arquivo *Kconfig.host* que foi alterado para possuir a opção booleana SPDM, deixando disponível *CONFIG_SPDM* no arquivo *config-host.mak*.

No arquivo de construção Meson, verifica-se a existência de *CONFIG_SPDM* e se essa está com o valor "*y*", permitindo, então, que compile o QEMU utilizando a LibSPDM. As flags de compilação, configuradas em *configure* que indicam os diretórios com os *headers*, são acessadas em *meson.build* e acrescentadas às globais. Os binários, por sua vez, devem ser buscados no diretório da LibSPDM, criando uma dependência com eles e adiciona-a ao coletivo de compilação dos drivers, em especial o responsável por compilar o HD virtual.

```
1 if config_host['CONFIG_SPDM'] == 'y'
2   spdm_flags = []
3   foreach flag : config_host['SPDM_CFLAGS'].split()
```

[9] https://mesonbuild.com/

```
4     spdm_flags += flag
5   endforeach
6
7   add_global_arguments(spdm_flags,
8                        native: false, language: ['c', 'cpp', 'objc'])
9 endif
```

Listing 5.16 – Inclusão das flags necessárias para LibSPDM em *meson.build*.

```
1 if config_host['CONFIG_SPDM'] == 'y'
2   spdm_inc = []
3   foreach inc : config_host['SPDM_INC'].split()
4     spdm_inc += inc
5   endforeach
6
7   spdm_libs = []
8   foreach lib : config_host['SPDM_LIBS'].split()
9     spdm_libs += cc.find_library(lib, dirs : config_host['SPDM_BUILD_DIR
      ']])
10  endforeach
11
12  spdm_dep = declare_dependency(dependencies: spdm_libs)
13
14  softmmu_ss.add(spdm_dep)
15 endif
```

Listing 5.17 – Declaração da dependência dos binários e inclusão no coletivo de compilação em *meson.build*.

A compilação da LibSPDM com arquitetura x86_64 para o hospedeiro deve ser indicada nos novos parâmetros do QEMU conforme o primeiro comando de 5.18

```
1 $ mkdir build
2 $ cd build
3 $ ../configure \
4     --target-list=riscv64-softmmu \
5     --enable-gtk \
6     --enable-system \
7     --enable-virtfs \
8     --enable-sdl \
9     --enable-jemalloc \
10    --enable-nettle \
11    --disable-pie \
12    --enable-debug \
13    --disable-werror \
14    --enable-libspdm \
15    --libspdm-srcdir=/path/to/libspdm \
16    --libspdm-builddir=/path/to/libspdm/build_host \
```

```
17     --libspdm-crypto=mbedtls
18 $ make
```

Listing 5.18 – Comandos para configurar e compilar o QEMU.

Ademais, os certificados que a LibSPDM irá utilizar devem ser posicionados no diretório em que o QEMU foi compilado. Esses encontram-se dentro do repositório da LibSPDM, */path/to/libspdm/build_host/bin*, e é feito um link simbólico para *ecp384* e *rsa3072* para dentro do emulador [5.19].

```
1 $ cd /path/to/qemu/build
2 $ ln -s /path/to/libspdm/build_host/bin/ecp384
3 $ ln -s /path/to/libspdm/build_host/bin/rsa3072
```

Listing 5.19 – Links simbólicos dos certificados para o QEMU.

5.8 HD Virtual

O HD virtual faz parte dos dispositivos da classe VirtIO, os quais operam com maior eficiência por cooperar com o *hypervisor*. Esses podem ser implementados como dispositivos PCI, mas como hardwares embarcados normalmente não incluem esse tipo de conexão, utiliza-se dispositivo mapeado em memória (MMIO). Para as configurações padrões do U-Boot com alvo no QEMU, tanto RISC-V 64 bits, quanto RISC-V 32 bits são configuradas com VirtIO MMIO.

A inicialização dos drivers VirtIO inicia-se com *virtio uclass driver*, o qual possui similaridades com os drivers de transporte – *virtio_mmio.c* e *virtio_pci.c*. Na função *probe* desses, é descoberto qual transporte é utilizado e o ID do dispositivo é armazenado nos dados privados do *virtio uclass*. De forma semelhante, no método *post_probe*, o dispositivo VirtIO – *virtio_net.c* e *virtio_blk.c* – é conectado se for compatível com o ID previamente guardado.

O arquivo *virtio_blk.c* descreve o driver para o HD virtual. A inicialização do dispositivo inicia-se com o método *bind*, o qual tem a função de informar o modelo de driver sobre a existência de um dispositivo compatível, e, em seguida, executa a função *probe*, cujo objetivo é preparar o dispositivo para ser utilizado. As operações do driver são duas: ler e escrever. Essas são implementadas utilizando a função de criar uma requisição, *virtio_blk_do_req*, o que diferencia-as é o tipo da operação que é passada como parâmetro [5.20] – *VIRTIO_BLK_T_IN* com valor 0, indicando leitura, e *VIRTIO_BLK_T_OUT* com valor 1, indicando escrita.

```
1 static ulong virtio_blk_do_req(struct udevice *dev, u64 sector,
2             lbaint_t blkcnt, void *buffer, u32 type)
3 {
```

```
4 /**
5  * Code to make a request
6  */
7 }
8
9 static ulong virtio_blk_read(struct udevice *dev, lbaint_t start,
10       lbaint_t blkcnt, void *buffer)
11 {
12   log_debug("read %s\n", dev->name);
13   return virtio_blk_do_req(dev, start, blkcnt, buffer,
14       VIRTIO_BLK_T_IN);
15 }
16
17 static ulong virtio_blk_write(struct udevice *dev, lbaint_t start,
18       lbaint_t blkcnt, const void *buffer)
19 {
20   return virtio_blk_do_req(dev, start, blkcnt, (void *)buffer,
21       VIRTIO_BLK_T_OUT);
22 }
```

Listing 5.20 – Operações do HD virtual.

5.9 HD Virtual com LibSPDM

Como o método *probe* é responsável por preparar o dispositivo para uso, a LibSPDM é inserida nele para ser configurada e iniciar o canal seguro de comunicação. Inicialmente, deve-se inicializar o contexto da LibSPDM, em seguida registram-se as funções de envio e recepção de mensagens [5.23], em conjunto com as funções de tranporte – essas são fornecidas pela LibSPDM e utiliza-se a codificação MCTP. A versão que será utilizada, 1.1, e as capacidades que o dispositivo possui também devem ser configuradas logo no início, são elas: medidas disponíveis, algoritmo de troca de chaves simétrico e assimétrico, algoritmo de hash e algoritmos de criptografia.

Para incluir a LibSPDM no driver em questão, adiciona-se as variáveis relacionadas ao SPDM na estrutura de dados do descritor de blocos, presente no arquivo *include/blk.h* [5.21]. Da mesma forma, adiciona-se em outro arquivo, *drivers/virtio/virtio_blk.h*, os tipos de comandos possíveis para enviar e receber as mensagens SPDM [5.22].

```
1 /*
2  * With driver model (CONFIG_BLK) this is uclass platform data,
     accessible
3  * with dev_get_uclass_plat(dev)
4  */
5 struct blk_desc {
6 /*
```

```
 * Code omitted
 */

#if CONFIG_IS_ENABLED(SPDM)
  void *spdm_context;
  uint32_t *session_id;
#endif
};
```

Listing 5.21 – Estrutura de dados do descritor de blocos.

```
#define VIRTIO_BLK_T_SPDM     16
#define VIRTIO_BLK_T_SPDM_APP 32
```

Listing 5.22 – Tipos de comandos para as requisições do HD virtual.

```
return_status spdm_virtblk_send_message(
  IN void   *spdm_context,
  IN uintn  request_size,
  IN void   *request,
  IN uint64 timeout
)
{
  struct blk_desc* desc;
  ulong status;
  desc = SPDM_CTX_TO_VIRTIOBLK(spdm_context);
  if (!desc)
    return RETURN_DEVICE_ERROR;

  status = virtio_blk_do_req(desc->bdev, 0, (lbaint_t)request_size,
   request, VIRTIO_BLK_T_SPDM | VIRTIO_BLK_T_OUT);

  return RETURN_SUCCESS;
}

return_status spdm_virtblk_receive_message(
  IN      void    *spdm_context,
  IN OUT  uintn   *response_size,
  IN OUT  void    *response,
  IN      uint64  timeout
)
{
  struct blk_desc *desc;

  desc = SPDM_CTX_TO_VIRTIOBLK(spdm_context);
  if (!desc)
    return RETURN_DEVICE_ERROR;

```

```
32   *response_size = virtio_blk_do_req(desc->bdev, 0, *response_size,
      response, VIRTIO_BLK_T_SPDM | VIRTIO_BLK_T_IN);
33
34   return RETURN_SUCCESS;
35 }
```

Listing 5.23 – Funções para enviar e receber mensagens SPDM para o HD virtual.

As capacidades são configuradas por meio do método *spdm_set_data* [5.24], o qual insere um dado no contexto SPDM. As informações que são inseridas são inicializadas no arquivo *spdm_flags.c* (apêndice A). Todo esse processo foi encapsulado dentro do método *virtblk_init_spdm*, cujo retorno é o contexto SPDM configurado e, para conseguir acessar a estrutura de dados do HD virtual (uma *struct*) a partir do contexto SPDM, cria-se uma *macro* que faz aritmética de ponteiro e retorna o endereço da estrutura. Essa operação é importante para conseguir acessar o método de requisição, utilizado nas funções apresentadas em 5.23.

```
1 return_status spdm_set_data(
2   IN void                    *context,
3   IN spdm_data_type_t        data_type,
4   IN spdm_data_parameter_t   *parameter,
5   IN void                    *data,
6   IN uintn                   data_size
7   );
```

Listing 5.24 – Função *spdm_set_data*.

```
1 #define SPDM_CTX_TO_VIRTIOBLK(spdm_context_ptr) *(struct blk_desc**)((
    char*)(spdm_context_ptr) + spdm_get_context_size())
```

Listing 5.25 – Macro para acessar a estrutura de dados do HD virtual a partir do contexto SPDM.

```
1 desc->spdm_context = virtblk_init_spdm();
2 if (desc->spdm_context == NULL) {
3     log_err("[SPDM @ U-Boot]: SPDM context is NULL.\n");
4     status = -1;
5     goto out;
6 }
7 // hack to be able to access vblk if we only have the context
8 SPDM_CTX_TO_VIRTIOBLK(desc->spdm_context) = desc;
9
10 status = spdm_init_connection(
11     desc->spdm_context,
12     (m_exe_connection & EXE_CONNECTION_VERSION_ONLY) != 0
13     );
14 if (RETURN_ERROR(status)) {
```

```
15      log_err("[SPDM @ U-Boot]: Error initializing SPDM connection: 0x%llx
        .\n", status);
16      goto out_free_spdm;
17  } else {
18      log_warning("[SPDM @ U-Boot]: SPDM connection initialized.\n");
19  }
```

Listing 5.26 – Inicialização do contexto dentro de *probe*.

O próximo passo é iniciar a conexão com a função *spdm_init_connection* que irá fazer as negociações iniciais como descrito em 2.3 – *GET_VERSION*, *GET_CAPABILITIES* e *GET_ALGORITHMS*. Em seguida, os certificados são inicializados no contexto SPDM, novamente com a função *spdm_set_data*, sendo esses a cadeia de certificados de ambos, *responder* e *requester*. Desse modo é possível garantir que os dispositivos tentando se comunicar estão trocando mensagens com um aparelho que é permitido. Nesse trabalho a cadeia de certificados está as claras dentro do código, *hard coded*, pois, caso fosse em um cenário real, ela estaria em algum outro dispositivo em que o driver pudesse consultar.

Para realizar a autenticação SPDM, utiliza-se a função *do_authentication_via_spdm* que aloca memória para receber os certificados do QEMU e chama o método, parte da API da LibSPDM, *spdm_authentication*. Essa realiza as operações *GET_DIGEST*, *GET_CERTIFICATE* e *CHALLENGE*, e, caso esteja tudo correto, o dispositivo está pronto para iniciar a comunicação segura. O canal de comunicação segura, portanto, é configurado ao realizar a troca de chaves, o que é feito por meio da função *spdm_start_session*.

```
1  virtblk_init_spdm_certificates(desc->spdm_context);
2
3  status = do_authentication_via_spdm(desc->spdm_context);
4  if (RETURN_ERROR(status)) {
5      log_err(
6          "[SPDM @ U-Boot]: Error authenticating via SPDM.\n"
7          "\tStatus 0x%02X\n",
8          (u32)status
9          );
10     goto out_free_spdm;
11 } else {
12     log_warning("[SPDM @ U-Boot]: Authenticated via SPDM.\n");
13 }
14
15 use_psk = FALSE;
16 heartbeat_period = 0;
17 desc->session_id = 0;
18 status = spdm_start_session(
19                     desc->spdm_context,
20                     use_psk,
21                 m_use_measurement_summary_hash_type,
22                 m_use_slot_id,
```

```
                    &desc->session_id,
                &heartbeat_period,
                    measurement_hash
                    );
if (RETURN_ERROR(status)) {
    log_warning("[SPDM @ U-Boot]: Error starting SPDM session 0x%02X.\n"
    , (uint32)status);
    goto out_free_spdm;
} else {
    log_warning("[SPDM @ U-Boot]: Started SPDM session.\n");
}
```

Listing 5.27 – Inicialização dos certificados, autenticação por meio do SPDM e inicialização da sessão segura.

É necessário alterar a função de requisição para que essa comporte as mensagens SPDM, pois o tamanho da mensagem é, originalmente, múltiplo de blocos de 512 bits. Para corrigir esse pequeno problema, pode-se incluir no código 5.28 a variável *size* que, por meio de um operador ternário, recebe o tamanho da mensagem se for uma mensagem SPDM e um multiplicador de blocos caso seja uma requisição comum. Então, esse valor é adicionado na estrutura de dados *scatter-gather data_sg*.

```
lbaint_t size = (type & VIRTIO_BLK_T_SPDM) ? blkcnt : blkcnt * 512;
lbaint_t temp_size;

struct virtio_blk_outhdr out_hdr = {
    .type = cpu_to_virtio32(dev, type),
    .sector = cpu_to_virtio64(dev, sector),
};
struct virtio_sg hdr_sg = { &out_hdr, sizeof(out_hdr) };
struct virtio_sg data_sg = { buffer, size };
struct virtio_sg status_sg = { &status, sizeof(status) };
```

Listing 5.28 – Trecho do código de requisição com as alterações de tamanho.

virtio_blk_do_req também precisa lidar com o header do protocolo de comunicação de dispositivos de bloco, caso contrário as funções da API da LibSPDM não conseguirão reconhecer as mensagens e retirar o conteúdo necessário delas. Para isso, basta deslocar o endereço do *buffer* em 5 bytes caso a requisição seja SPDM. Outra modificação é importante para depuração é o retorno, sendo que, caso seja uma requisição comum, o retorno deve ser a quantidade de blocos lidas ou escritas, mas para as mensagens SPDM utiliza-se a variável *size*, desse modo é possível ter conhecimento de quantos bits sofreram a operação.

```
if (type & VIRTIO_BLK_T_SPDM) {
    temp_size = * ((u32*) (buffer+1));
```

```
3     if (temp_size  > size) {
4         ret = -1;
5         size = 0;
6         return ret;
7     }
8     size = temp_size;
9
10    // magic number: assuming 1-byte message type and 4-byte message
      size
11    memmove (buffer, buffer + 5, size);
12
13    return status == VIRTIO_BLK_S_OK ? size : -EIO;
14 } else {
15    return status == VIRTIO_BLK_S_OK ? blkcnt : -EIO;
16 }
```

Listing 5.29 – Deslocamento de 5 bytes para descartar o header caso seja uma requisição SPDM.

Por conta das funções dos certificados, é necessário possuir o método *read_input_file* para conseguir realizar os *links*. Contudo, esse método não é utilizado pelo HD virtual, pois a cadeia de certificados não é lida de uma lista, mas dos *arrays* às claras no código. Logo, pode-se declarar a função com os mesmo parâmetros, porém que não realiza nenhuma operação.

```
1 boolean read_input_file(
2   IN  char8 *file_name,
3   OUT void  **file_data,
4   OUT uintn *file_size
5 )
6 {
7
8 }
```

Listing 5.30 – Função necessária para realizar os *links*.

5.10 Criação da Mídia de Boot

Ao utilizar OpenSBI e U-Boot para iniciar o Linux, o QEMU deve receber o parâmetro *bios* com o arquivo *fw_payload.elf* gerado pela compilação descrita em 5.3. Dessa forma, a imagem do sistema operacional e o sistema de arquivos raiz devem estar em uma mídia particionada que inicialmente é criada com o comando 5.31.

```
1 $ dd if=/dev/zero of=disk.img bs=1M count=128
```

Listing 5.31 – Criando mídia vazia chamada *disk.img* de 128 MB.

Em seguida, cria-se uma tabela de partições GPT e, para operá-las como um dispositivo de blocos, monta-se ela em um dispositivo *loop* vazio designado pelo sistema operacional. Então, fazem-se duas partições de 64 MB do tipo ext4, as quais devem ser formatadas, e habilita-se o boot na primeira partição. Essa sequência é exemplificada em 5.32, sendo que N pode assumir diferentes números em */dev/loopN*.

```
1 $ sudo parted disk.img gpt
2 $ sudo losetup --find --show disk.img
3 $ sudo parted --align minimal /dev/loopN mkpart primary ext4 0% 50%
4 $ sudo parted --align minimal /dev/loopN mkpart primary ext4 50% 100%
5 $ sudo mkfs.ext4 /dev/loopNp1
6 $ sudo mkfs.ext4 /dev/loopNp2
7 $ sudo parted /dev/loopN set 1 boot on
```

Listing 5.32 – Sequência de comandos para criar, formatar e configurar as partições.

As partições devem, então, ser preenchidas pelos arquivos que vão permitir a execução do Linux. Criam-se dois diretórios em */mnt*, *boot* e *rootfs*, para montar ambas partições e copiar a imagem do SO e o sistema de arquivos raiz. Ademais, cria-se um diretório para montar o sistema de arquivos que o Buildroot gerou, encontra um dispositivo de loop para ele e, então, transfere para */mnt/rootfs* [5.33].

```
1 $ sudo mkdir /mnt/boot
2 $ sudo mkdir /mnt/rootfs
3 $ sudo mkdir /mnt/buildroot
4 $ sudo losetup --find --show /path/to/buildroot/output/images/rootfs.
    ext4
5
6 # loopM is returned by losetup
7 $ sudo mount /dev/loopM /mnt/buildroot
8 $ sudo mount /dev/loopNp1 /mnt/boot
9 $ sudo mount /dev/loopNp2 /mnt/rootfs
10 $ sudo cp /path/to/buildroot/output/images/Image /mnt/boot
11 $ sudo cp /mnt/buildroot/* /mnt/rootfs
```

Listing 5.33 – Passos finais para ter uma mídia de boot.

Por fim, todo o ambiente é limpo, desmontando e excluindo os diretórios em */mnt*, além de liberar os dispositivos de loop para futuro uso. Os comandos utilizados para essa finalidade são apresentados em 5.34.

```
1 $ sudo umount /mnt/boot
2 $ sudo umount /mnt/rootfs
3 $ sudo umount /mnt/buildroot
4 $ sudo losetup -d /dev/loopN
5 $ sudo losetup -d /dev/loopM
```

```
6 $ sudo rm -rf /mnt/*
```

Listing 5.34 – Limpando o ambiente e liberando os dispositivos de loop.

5.11 Execução

O comando para iniciar a emulação é demonstrado em 5.35. Utiliza-se a opção *smp* para especificar a quantidade de núcleos do hospedeiro disponíveis para uso e *m*, a quantidade de memória, escolhe-se 4 núcleos e 8 Gb para ser mais próximo das especificações da HiFive Unleashed (SIFIVE INC, 2021b). Essa placa é anunciada como tendo 4+1 núcleos, sendo o quinto do tipo RV64IMAC e o restante, RV64GC (SIFIVE INC, 2021a), logo, como a máquina *virt* do QEMU possui 255 cores RV64GC disponíveis, utiliza-se *-smp 4*. Outra opção necessária é a conexão do disco criado, isso é realizado declarando o *front end device*, descreve como o dispositivo é apresentado para a máquina virtual, e o *back end drive*, descreve como os dados do dispositivo emulado serão processados pelo QEMU.

O disco é conectado por meio de um PCIe virtual ao emulador e, apesar da HiFive Unleashed não possuir esse barramento, é possível adquirir uma placa de extensão que fornece esse tipo de conexão, portanto a proximidade da placa real não é perdida. Embora o QEMU forneça a opção de emular o hardware da SiFive com o parâmetro *-M sifive_u*, escolhe-se *virt* para ter PCIe e manter compatibilidade com as alterações já feitas no arquivo do emulador, *hw/block/virtio_blk.c*, por (ALVES; ALBERTINI; SIMPLICIO, 2022).

```
1 $ /path/to/qemu/build
2 $ ./riscv64-softmmu/qemu-system-riscv64 \
3     -M virt \
4     -smp 4 \
5     -m 8G \
6     -bios /path/to/opensbi/build/platform/generic/firmware/fw_payload.
    elf \
7     -drive file=/path/to/disk.img,format=raw,id=hd0 \
8     -device virtio-blk-device,drive=hd0
```

Listing 5.35 – Comandos para executar a emulação.

As impressões no terminal apresentadas na figura 11 indicam o término de cada fase do padrão SPDM. A primeira mensagem é referente às negociações de versão, capacidades e algoritmos; a segunda, à autenticação do dispositivo por meio da verificação a cadeia de certificados; a última, ao passo de troca de chaves para iniciar o canal de comunicação seguro.

```
U-Boot 2023.07-rc6-00001-gd901c3f500-dirty (Nov 27 2023 - 15:41:27 -0300)

CPU:   rv64imafdcsu
Model: riscv-virtio,qemu
DRAM:  8 GiB
Core:  24 devices, 13 uclasses, devicetree: board
Flash: 32 MiB
Loading Environment from nowhere... OK
In:    uart@10000000
Out:   uart@10000000
Err:   uart@10000000
Net:   No ethernet found.
Working FDT set to ff6c9970

Device 0: [SPDM @ U-Boot]: SPDM connection initialized.
[SPDM @ U-Boot]: Authenticated via SPDM.
[SPDM @ U-Boot]: Started SPDM session.
Triggering tamper of measurement 0
QEMU VirtIO Block Device
            Type: Hard Disk
            Capacity: 1024.0 MB = 1.0 GB (2097152 x 512)
... is now current device
=>
```

Figura 11 – U-Boot verificando a integridade do HD virtual conectado – *Device* 0.

5.12 Depuração

É desejável que as mensagens possam ser visualizadas para que o conteúdo seja depurado, deve ser facilmente habilitada quando necessário. Portanto, para tal feito, utiliza-se da mesma solução com Kconfig apresentada na seção 5.6, adicionando a opção 5.36

```
1 config SPDM_DEBUG
2   bool "Enable SPDM debug"
3   help
4     This enables SPDM debug, so exchanged messages are visible.
```

Listing 5.36 – Opção no Kconfig raiz do U-Boot para visualizar as mensagens SPDM.

De forma similar à apresentada em 5.12, escrevem-se as impressões no código utilizando a macro disponível. O código apresentado em 5.37 é semelhante ao utilizado para visualizar as mensagens recebidas, a diferença consiste no tamanho do *loop* executado, pois, como elas podem ser muito grandes, a saída foi limitada para 17 bytes. Desse modo, o terminal não fica poluído e é possível ver o *header* do padrão SPDM, para conferir qual tipo de mensagem está sendo trocada, e parte do conteúdo transmitido pelo hardware emulado.

```
1 #if CONFIG_IS_ENABLED(SPDM_DEBUG)
```

```
2   log_warning("[SPDM @ U-Boot]: Sending message of size %llu\n",
     request_size);
3   for (int i = 0; i < request_size; i++) {
4     log_warning("%02X ", ((uint8_t*)request)[i]);
5   }
6   log_warning("\n\n");
7 #endif
```

Listing 5.37 – Código para imprimir as mensagens trocadas. O exemplo fornecido é para mensagens enviadas pelo driver do HD virtual no U-Boot.

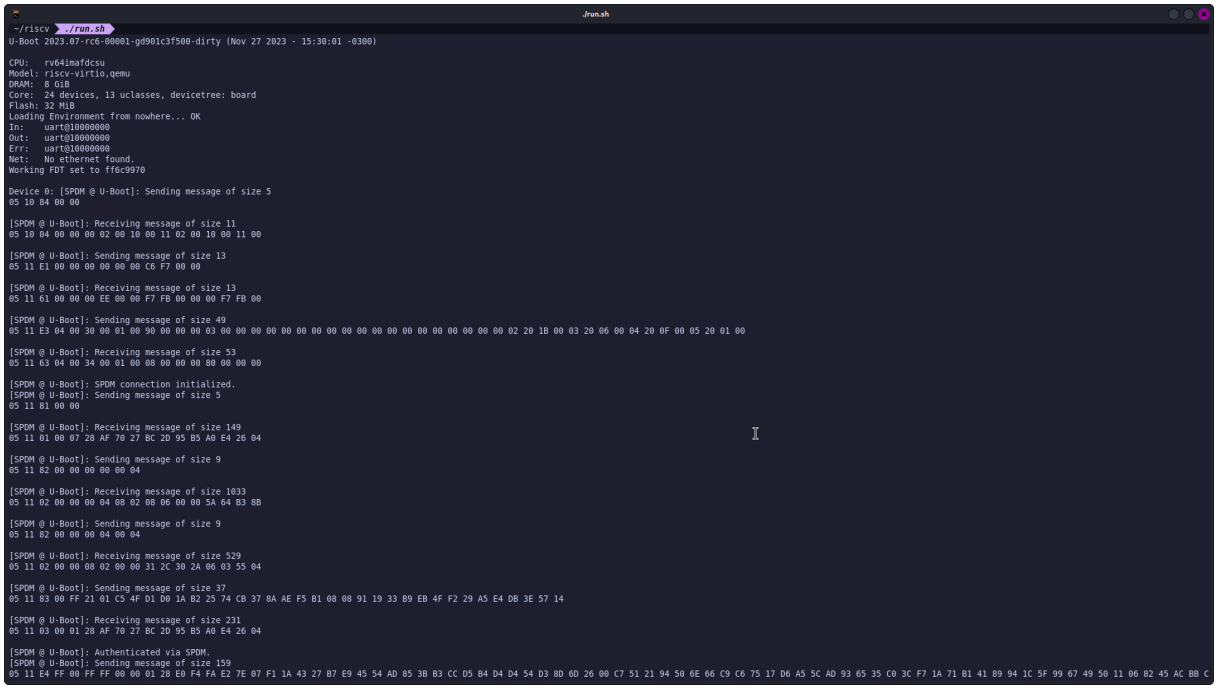


Figura 12 – Exemplo de execução ao ativar a opção *SPDM_DEBUG* nas configurações. A imagem apresenta as mensagens trocadas até a autenticação via SPDM do HD virtual.

É possível, também, visualizar as mensagens pelo ponto de vista do hardware emulado. Basta modificar o valor de duas macros presentes no arquivo *hw/block/virtio-blk.c* dentro do diretório do QEMU, as macros são apresentados no trecho de código 5.38.

```
1 #define BLK_SPDM_DEBUG 0
2 #define BLK_SPDM_DEMO_PRINT 0
3 #define DEMO_PRINT_LIMIT 256
4 #define DEMO_BYTES_PER_LINE 16
5
6 #if BLK_SPDM_DEBUG
7 #define BLK_SPDM_PRINT(format, ...) printf(format, ##__VA_ARGS__)
8 #else
```

```
9 #define BLK_SPDM_PRINT(format, ...)
```

Listing 5.38 – Macros responsáveis por imprimir no terminal as mensagens trocadas pelo hardware emulado. BLK_SPDM_DEBUG e BLK_SPDM_DEMO_PRINT são as macros que ativam e desativam as impressões.

5.13 Iniciando o Linux

Por último, o HD sendo íntegro, é possível iniciar o Linux ao acessar o terminal do U-Boot, basta executar alguns comandos – os quais podem ser automatizados, porém nesse trabalho não o são para fins de demonstração do padrão SPDM. Os comandos necessários para realizar o boot são apresentados em 5.39.

```
1 => setenv bootargs "root=/dev/vda2 rw console=ttyS0 init=/sbin/init"
2 => ext4load virtio 0:1 $kernel_addr_r Image
3 => booti $kernel_addr_r - $fdtcontroladdr
```

Listing 5.39 – Comandos para iniciar o Linux a partir do U-Boot.

Inicialmente, configura-se os argumentos de boot para iniciar o kernel com o comando *setenv*. Com ele, o sistema de arquivo raiz na segunda partição do HD virtual, o *bind* do terminal na interface ttyS0, a indicação do diretórion *init* e a opção de leitura e escrita do kernel é armazenada na variável *bootargs*. O segundo comando é para carregar de uma partição do tipo *ext*4, o qual busca a imagem do SO na primeira partição do HD virtual e armazena o valor na variável *kernel_addr_r*. Por fim, *booti* é utilizado para iniciar a imagem no endereço calculado.

5.14 Condensação do Ambiente

A coleta das métricas para realizar a análise da sobrecarga exige várias compilações com configurações distintas, portanto, com o intuito de otimizar o tempo despendido durante essa atividade, fabrica-se um Makefile para reduzir o número de caracteres a serem digitados para que a compilação do U-Boot e do OpenSBI ocorra. O arquivo utilizado é apresentado no apêndice C.

O Makefile inicia-se com a configuração das variáveis necessárias, facilitando possíveis alterações de caminho – i.e. apenas uma linha seria modificada em caso de mudanças no diretório de trabalho. O Buildroot precisa ser compilado apenas uma vez, porém para motivos de testes, duas regras envolvendo-o são criadas: `buildroot` e `linux-rebuild`. Para que não ocorra erros relacionados a não existência das ferramentas de compilação cruzada e do binário do firmware, regras são criadas para, antes da compilação do U-Boot

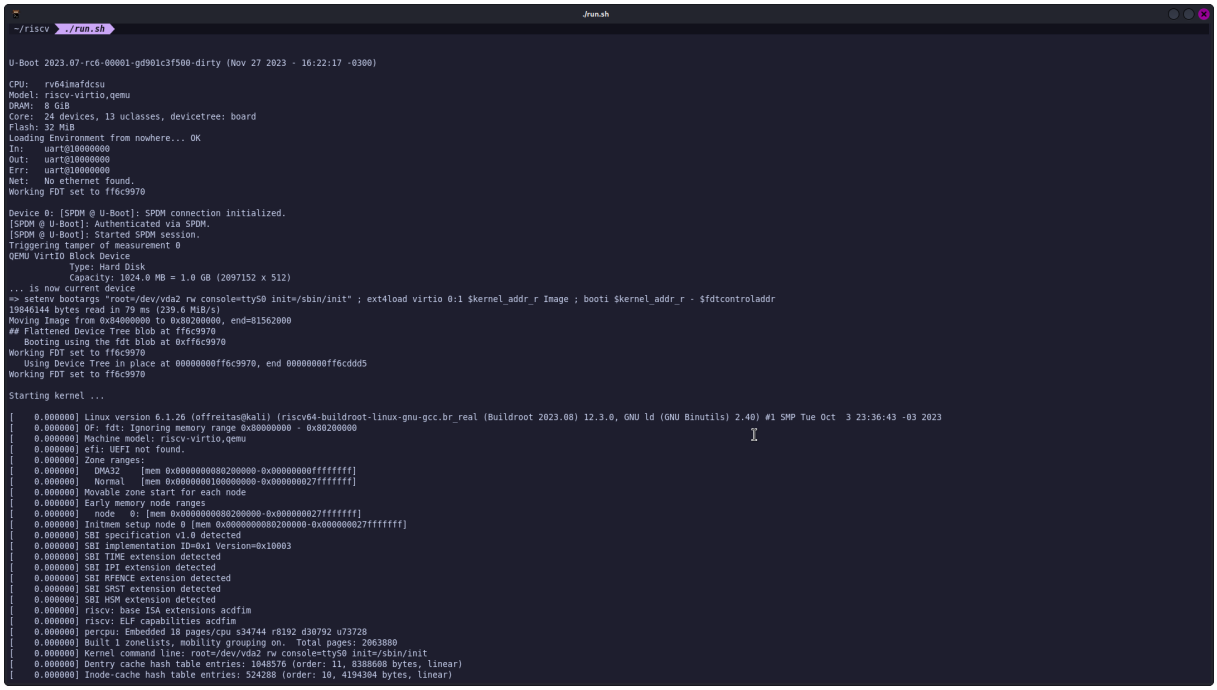


Figura 13 – Os comandos apresentados em 5.39 são posicionados em uma única linha para iniciar o Linux.

e OpenSBI, a checagem de dependências. Um comando no terminal que foi utilizado com frequência durante a coleta de dados é apresentada em 5.40.

```
1 $ make uboot && make opensbi
```

Listing 5.40 – Comando para compilar o U-Boot e, em seguida, utilizar o binário como payload do OpenSBI.

5.15 Medição do Tempo

Inicialmente, tentou-se implementar a medição de tempo com o header *time.h*. A função *timer_get_us* disponível retorna uma variável inteira sem sinal de 64 bits, a qual pode ser utilizada como argumento para outra – *get_timer_us* –, ambas retornam o tempo em microssegundos. Embora exista esse header, ao tentar usá-lo, não houve sucesso, pois da forma que estava sendo feito, chamando a função em momentos diferentes e subtraindo para conseguir o intervalo, acarretava em overflow.

Outro header com objetivos similares foi encontrado, *timer.h*, e passou a ser utilizado por não ocorrer overflow. Para isso deve-se inicializar o driver com o método *dm_timer_init*, permitindo coletar dados para o cálculo do tempo – a inicialização já é feita automaticamente se *CONFIG_TIMER* estiver ativo nas configurações do U-Boot. A fórmula utilizada é apresentada em 5.1, ela é válida devido à frequência retornada pela função *timer_get_rate* ser constante – é a frequência do timer, não da CPU – e a

multiplicação por mil devolve o valor em milissegundos, objetivando uma análise temporal com maior número de casas decimais.

$$t_{ms} = \frac{ticks \times 1000}{rate} \tag{5.1}$$

```
ulong timer_get_boot_ms(void)
{
  u64 ticks = 0;
  u32 rate = 1;
  u64 ms;
  int ret;

  if (!ret) {
    /* The timer is available */
    rate = timer_get_rate(gd->timer);
    timer_get_count(gd->timer, &ticks);
  } else {
    return 0;
  }

  ms = (ticks * 1000) / rate;
  return ms;
}

static int virtio_blk_probe(struct udevice *dev)
{
  struct virtio_blk_priv *priv = dev_get_priv(dev);
  struct blk_desc *desc = dev_get_uclass_plat(dev);
  u64 cap;
  int ret;
  uint64_t elapsed_time;

    /**
     * HD setup code
     */

  elapsed_time = timer_get_boot_ms();
  log_warning("[MEASUREMENT] Elapsed time: %lu us\n", elapsed_time);

  return 0;

out_free_spdm:
  free(desc->spdm_context);
out:
  return status;
```

```
41 }
```

Listing 5.41 – O tempo é mensurado e o valor em micro segundos é armazenado em *elapsed_time*.

Com o intuito de coletar vários resultados, faz-se uso de *shell script* que possui o comando *timeout*, desse modo pode-se executar a emulação com QEMU durante um certo tempo estipulado e então um sinal de *kill* é enviado, persistindo os tempos medidos em outro arquivo. Essa sequência é posicionada dentro de um loop que irá ser executada por um número de vezes estipulado por um argumento passado no momento da chamada do script. O arquivo *sh* está disponível no apêndice B e um exemplo de uso é apresentado em 5.42.

```
1 $ ./collect-time.sh 100
```

Listing 5.42 – Coleta o tempo de configuração do HD como dispositivo padrão 100 vezes.

5.16 Avaliação de Desempenho

Para realizar as análises temporais, utiliza-se o conceito de quartis para observar a distribuição e a variância dos dados. Uma forma de conseguir observar esses dados com melhor visibilidade é com o tipo de gráfico *boxplot*. As caixas representam a distância interquartil, i.e. a distância entre o terceiro e segundo quartil, sendo o quartil igual a um quarto do número de dados. A linha vertical inferior é o valor mais baixo da caixa menos um múltiplo da distância interquartil, assim como a linha superior é o valor mais alto mais um múltiplo da distância interquartil – o múltiplo é igual a três meios. Os valores que estão abaixo da linha inferior ou acima da linha superior são considerados outliers.

$$linha\ inferior = Q_1 - 1.5 \times IQR \tag{5.2}$$

$$linha\ superior = Q_3 + 1.5 \times IQR \tag{5.3}$$

Utilizando os dados temporais coletados como descrito na seção 5.15, utiliza-se a linguagem Python para analisar as 1000 medições para cada situação – com e sem o padrão SPDM –, realizando a construção do gráfico *boxplot*.

Observando a figura 14 temos que o tempo médio para acessar o shell do U-Boot sem o protocolo é 1,2 segundos, enquanto, com o protocolo, é 10,9 segundos, isso demonstra uma piora de, aproximadamente, 9 vezes. A variância dos valores é mais significativa quando há a negociação SPDM, visível na amplitude interquartil e na amplitude entre mínimo e máximo. Há a presença de 16 outliers quando com SPDM – 1,6% – e 1, quando sem – 0,1%. Com a posição da mediana, linha horizontal vermelha, e da média aritmética, triângulo verde, é possível afirmar que a distribuição do tempo sem o padrão SPDM é

mais próxima de uma distribuição normal, pela proximidade dessas duas métricas, que a distribuição com o padrão SPDM

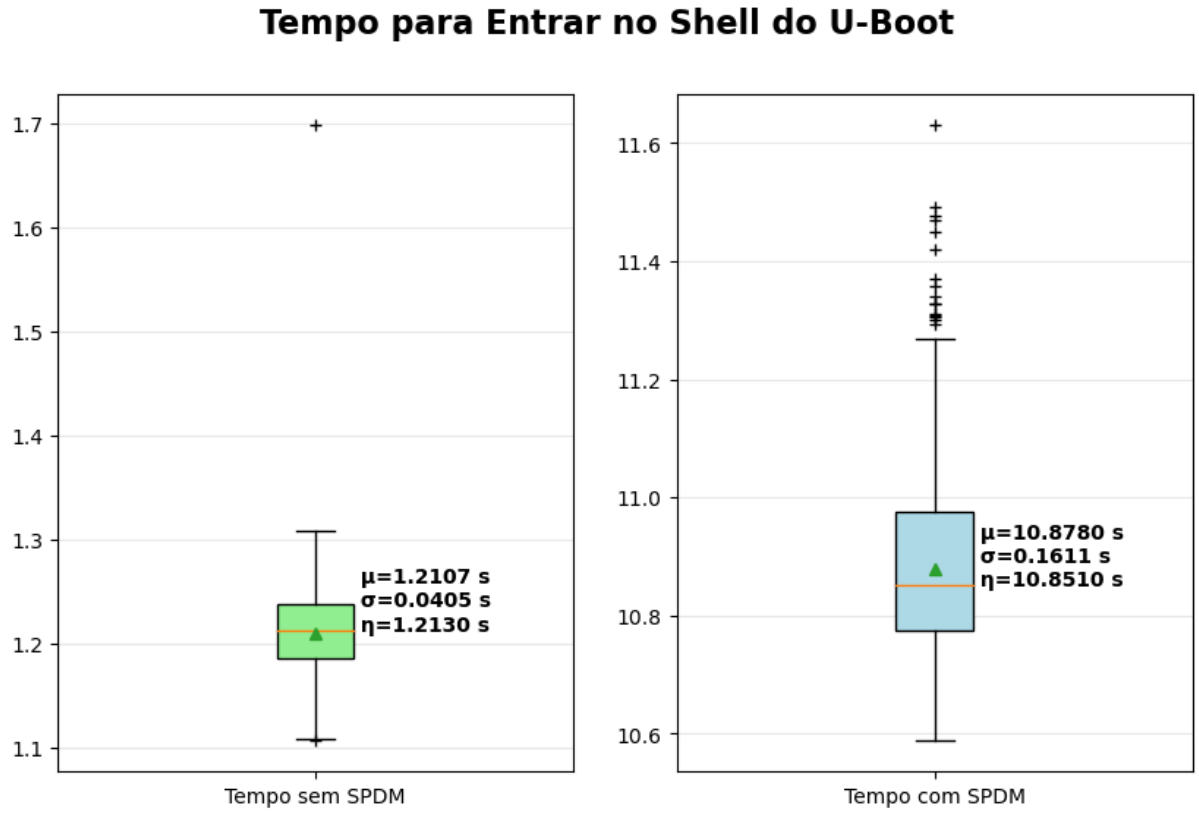


Figura 14 – Comparação dos tempos para acessar o shell do U-Boot. Os valores, em ordem, são: média aritmética, desvio padrão e mediana.

Com o intuito de medir o impacto da troca de mensagens em uma sessão segura, inclui-se no código do driver requisições para medidas – *GET_MEASUREMENTS*. As medidas podem ser requisitadas por meio de duas formas: todas as medidas em uma só requisição e uma medida por vez. Para efeitos de comparação, ambas maneiras são analisadas e suas implementações são apresentadas em D.

É perceptível a sobrecarga imposta no sistema quando mensagens são trocadas através do canal seguro de comunicação, se for uma requisição por vez há uma piora temporal de aproximadamente 13 vezes em relação ao tempo sem o padrão SPDM, com 1.7% de outliers. Um método melhor para conseguir as métricas é uma requisição que retorna todas as métricas de uma vez, a piora nesse caso é de aproximadamente 11 vezes, com a presença de 7 outliers – 0.7%.

Para ambos os casos de trocas de mensagens, percebe-se uma variação alta acima do terceiro quartil, apresentando medidas mais esparsas. Essas distribuições afastam-se ainda mais da distribuição normal em relação ao tempo com o padrão SPDM sem nenhuma mensagem enviada pelo canal seguro, sendo a razão média e mediana aproximadamente 1.004 para ambas situações.

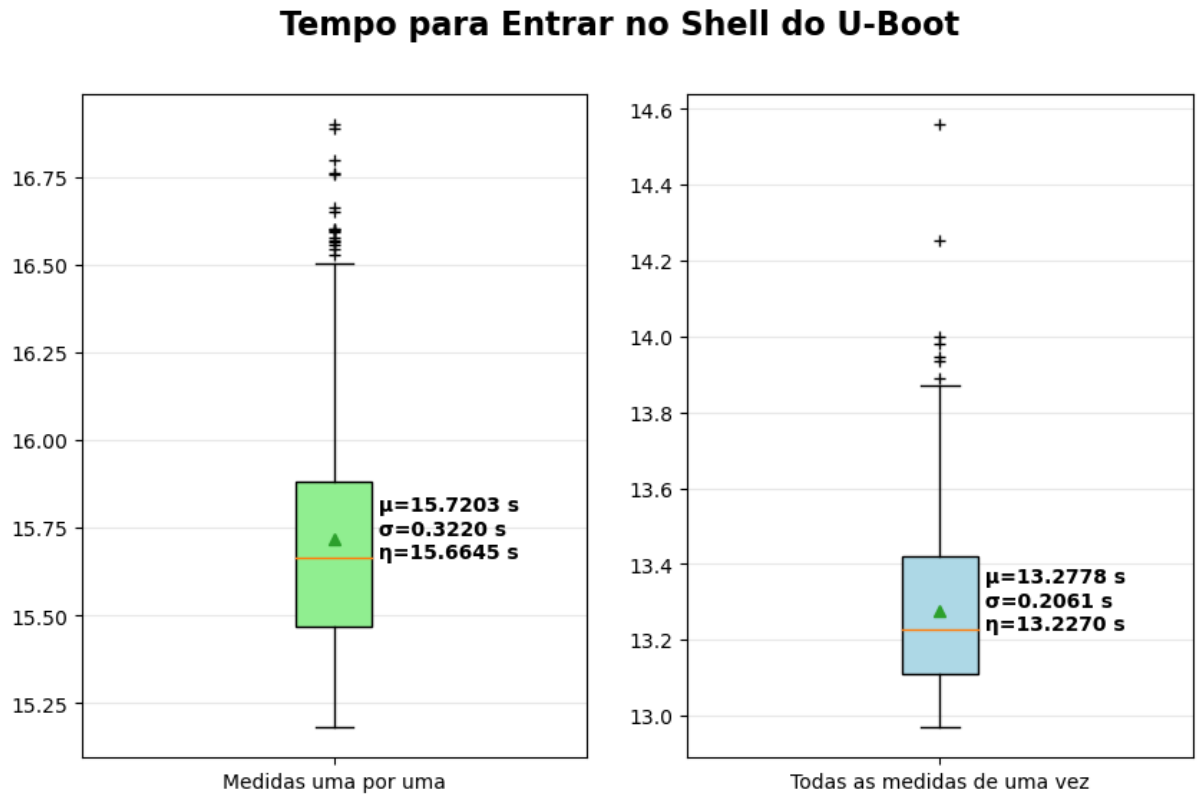


Figura 15 – Comparação dos tempos para acessar o shell do U-Boot após as requisições de medidas. Os valores, em ordem, são: média aritmética, desvio padrão e mediana.

Com a descrição dos resultados, é visível que a sobrecarga imposta em um sistema computacional com o padrão SPDM a nível de firmware pode ser alta. Entretanto, essa degradação de performance era esperada, pois mais funcionalidades são aplicadas em um dispositivo com o intuito de oferecer os três serviços de segurança apresentados no capítulo 2.

6 Considerações Finais

6.1 Conclusões do Projeto de Formatura

Como visto na seção 5.16, a sobrecarga imposta no sistema computacional é alta, no entanto a segurança empregada há de ser considerada. Este projeto de formatura implementou a verificação para um HD virtual que deseja iniciar o Linux, mas o padrão SPDM pode ser aplicado para outros drivers presentes no firmware e oferecer os mesmos serviços: confiabilidade, integridade e autenticidade.

A aplicação desse padrão a nível de firmware pode levar a empecilhos, pois, apesar da segurança em hardware embarcado estar sendo confrontada por ataques, as vulnerabilidades não são corrigidas por um longo período (Microsoft Security Team, 2023) devido a fatores como acesso ao dispositivo, economia energética e negligência do usuário. A sobrecarga apresentada na seção 5.16 implica em maior gasto energético e, também, tempo de desenvolvimento, fatores que podem influenciar entidades fornecedoras a serem resistentes à adoção do padrão SPDM.

Os testes realizados foram em ambiente emulado, o que implica em interações entre dispositivos trafegando em camadas de abstração e, por conseguinte, degradando o desempenho. Para melhor avaliar a sobrecarga imposta, um protótipo físico é necessário, além de incluir o padrão SPDM em outro driver distinto de um virtual.

Um hardware com arquitetura RISC-V 64 bits capaz de executar um SO complexo como Linux ainda é escasso, uma alternativa é a utilização de FPGAs em conjunto com uma linguagem de descrição de hardware para confeccionar uma CPU com essa arquitetura. Entretanto, com o tempo previsto para a finalização de um projeto de formatura, implementar uma CPU capaz de interpretar instruções RISC-V 64 bits e, também, incluir a LibSPDM em um firmware é inviável.

Por mais que muitas tecnologias de código aberto estejam integrando a arquitetura RISC-V, o suporte e a comunidade, mesmo aumentando, informações como guias de comunidade não são comuns. Os softwares utilizados normalmente, como OpenSBI e U-Boot, não oferecem nenhum impasse, pois as instruções são claras e funcionais. Contudo, no momento que algo inesperado ocorre, poucos resultados de busca são encontrados reportando um problema semelhante na arquitetura alvo. Devido a essa característica, as medidas de tempo também não ocorreram como idealizado no início do trabalho, pois a API do U-Boot para medições de tempo aparecem com frequência em CPUs ARM, mas não em RISC-V e a coleta de métricas teve um curto intervalo de tempo entre o fim da implementação e o prazo final.

Um dos objetivos iniciais era a comparação da implementação do padrão SPDM com outras tecnologias de mitigação de vulnerabilidades no firmware, entretanto, devido ao processo de desenvolvimento ter sido lento, não foi possível comparar com outras técnicas de proteção. Por conta do tempo empregado na busca de um firmware e na inclusão do padrão nos softwares necessários, também não houve janela para estudar o uso de outras tecnologias e criar uma métrica eficaz para realizar comparações.

6.2 Contribuições

Não há publicações ou repositórios do GitHub com a inclusão do padrão SPDM em um firmware de código aberto no momento da publicação deste trabalho, portanto, considerando esse cenário, é uma contribuição relevante para a comunidade open-source – o repositório de testes da TianoCore EDKII citado na seção 5.1 está em fases de avaliação e ainda não foi submetido para o repositório principal. Outro impacto para a comunidade open-source encontra-se na implementação focada na arquitetura RISC-V 64 bits, a qual vem recebendo atenção dos firmwares de código aberto e também do emulador QEMU.

Visto que o número de dispositivos embarcados ao redor do mundo vem aumentando, a preocupação com a segurança desses dispositivos em baixo nível também cresce, especialmente quando medidas de segurança tradicionais não são suficientes para garantir a proteção contra um firmware malicioso. Empresas e organizações que dependam de hardware embarcado ou que os comercializam podem considerar o valor do padrão SPDM para seus aparelhos, o desempenho ainda deve ser melhor avaliado, porém uma implementação em baixo nível contribui para que interessados não partam do zero.

6.3 Perspectivas de Continuidade

Alguns dos softwares utilizados necessitam de atualização, especialmente a LibSPDM e o QEMU, pois eles encontram-se atualmente na versão 3.0.0 e 8.0.0, respectivamente. Portanto, antes de enviar um pedido para acrescentar as mudanças no U-Boot e no QEMU com o padrão SPDM, é necessário realizar essas alterações. Ademais, para que as mudanças sejam aceitas, o código deve ser revisto e reescrito, caso necessário, para ser compatível com os estilos de cada software.

Ao invés de utilizar MbedTLS, deseja-se utilizar a biblioteca criptográfica presente no firmware, pois como o objetivo é um hardware embarcado, o tamanho do binário seria reduzido. Para testes emulados, o QEMU também teria que utilizar a OpenSSL e, portanto, haveria necessidade de modificar o código implementado nesse trabalho. Testes com outras arquiteturas também devem ser realizados, pois, antes de enviar um pedido para o repositório, as funcionalidades já existentes não devem ser alteradas indevidamente.

Sendo esse trabalho, o início de um mestrado, deseja-se realizar, em conjunto com outro mestrando, a implementação física do projeto para obter avaliações fidedignas da sobrecarga imposta no sistema. Além da tese para aquisição de título, alguns casos de aplicação do padrão SPDM em pesquisas do LARC-USP serão estudados objetivando a publicação de artigos.

Referências

AKRAM, M. et al. *Securing web transactions TLS server certificate management*. [s.n.], 2020. Disponível em: <http://dx.doi.org/10.6028/NIST.SP.1800-16>. Citado na página 12.

ALVES, R. C. A.; ALBERTINI, B. C.; SIMPLICIO, M. A. Securing hard drives with the security protocol and data model (spdm). In: . IEEE, 2022. p. 446–447. ISBN 978-1-6654-6605-9. Disponível em: <https://ieeexplore.ieee.org/document/9911999/>. Citado 6 vezes nas páginas 18, 19, 20, 22, 27 e 38.

BASNIGHT, Z. H. *Firmware counterfeiting and modification attacks on programmable logic controllers*. [S.l.], 2013. Citado 2 vezes nas páginas 9 e 10.

BASíLIO, L. B.; ROJA, L. B. P.; BOGER, M. d. S. *Integração do Security Protocol and Data Model ao Kernel Linux*. 2021. 50 p. Citado 3 vezes nas páginas 19, 22 e 27.

CHOI, B.-C. et al. Secure firmware validation and update for consumer devices in home networking. *IEEE Transactions on Consumer Electronics*, IEEE, v. 62, n. 1, p. 39–44, 2016. Citado 3 vezes nas páginas 9, 10 e 18.

DMTF. *Security Protocols and Data Models Working Group*. 2022. Disponível em: <https://www.dmtf.org/standards/SPDM>. Citado 8 vezes nas páginas 6, 10, 12, 13, 14, 15, 16 e 17.

ISO Central Secretary. *ISO / IEC 27000:2018*. Geneva, CH, 2018. Citado na página 12.

MALAJ, E. G.; MARINOVA, G. I. Review on hardware solutions for cybersecurity of communication systems. In: *2020 28th National Conference with International Participation (TELECOM)*. [S.l.: s.n.], 2020. p. 129–132. Citado na página 9.

Microsoft Security Team. *Security Signals*. 2021. Disponível em: <https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RWPStZ>. Citado 2 vezes nas páginas 10 e 11.

Microsoft Security Team. *Microsoft Digital Defense Report*. 2023. Disponível em: <file:///home/offreitas/Downloads/MDDR_FINAL_2023_1004.pdf>. Citado 2 vezes nas páginas 11 e 47.

PATEL, A. *OpenSBI Deep Dive*. 2019. Disponível em: <https://riscv.org/wp-content/uploads/2019/06/13.30-RISCV_OpenSBI_Deep_Dive_v5.pdf>. Citado 2 vezes nas páginas 6 e 23.

PATRA, A.; PATEL, A. *An Introduction to RISC-V Boot Flow*. 2019. Disponível em: <https://riscv.org/wp-content/uploads/2019/12/Summit_bootflow.pdf>. Citado 2 vezes nas páginas 6 e 23.

Research and Markets. *Global Embedded Security Market Analysis Report 2023: Market to Reach $9.8 Billion by 2028 with China Dominating*. 2023. <https://finance.yahoo.com/news/global-embedded-security-market-analysis-112300834.html>. Acessado: 29-11-2023. Citado 2 vezes nas páginas 10 e 11.

SIFIVE INC. *SiFive FU540-C000 Manual*. [S.l.], 2021. V1p4. Citado na página 38.

SIFIVE INC. *SiFive HiFive Unleashed Getting Started Guide*. [S.l.], 2021. V1p3. Citado na página 38.

SINHA, S. *State of IoT 2023: Number of connected IoT devices growing 16% to 16.7 billion globally*. 2023. <https://iot-analytics.com/number-connected-iot-devices/>. Acessado: 29-11-2023. Citado 2 vezes nas páginas 10 e 11.

WANG, X. et al. Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In: IEEE. *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. [S.l.], 2015. p. 544–551. Citado 3 vezes nas páginas 6, 10 e 17.

Apêndices

APÊNDICE A – Flags de Configuração do Contexto SPDM no HD Virtual

Listing A.1 – Variáveis que possuem os dados de configuração do contexto SPDM. Os comentários logo acima das variáveis são as possíveis configurações.

```
1  static uint32 m_use_transport_layer = SOCKET_TRANSPORT_TYPE_MCTP;
2
3  static uint8 m_use_version = SPDM_MESSAGE_VERSION_11;
4  static uint8 m_use_secured_message_version = SPDM_MESSAGE_VERSION_11;
5  static uint32 m_use_requester_capability_flags =
6    (0 |
7     SPDM_GET_CAPABILITIES_REQUEST_FLAGS_CERT_CAP |
8     SPDM_GET_CAPABILITIES_REQUEST_FLAGS_CHAL_CAP |
9     SPDM_GET_CAPABILITIES_REQUEST_FLAGS_ENCRYPT_CAP |
10    SPDM_GET_CAPABILITIES_REQUEST_FLAGS_MAC_CAP |
11    SPDM_GET_CAPABILITIES_REQUEST_FLAGS_MUT_AUTH_CAP |
12    SPDM_GET_CAPABILITIES_REQUEST_FLAGS_KEY_EX_CAP |
13    SPDM_GET_CAPABILITIES_REQUEST_FLAGS_PSK_CAP_REQUESTER |
14    SPDM_GET_CAPABILITIES_REQUEST_FLAGS_ENCAP_CAP |
15    SPDM_GET_CAPABILITIES_REQUEST_FLAGS_HBEAT_CAP |
16    SPDM_GET_CAPABILITIES_REQUEST_FLAGS_KEY_UPD_CAP |
17    SPDM_GET_CAPABILITIES_REQUEST_FLAGS_HANDSHAKE_IN_THE_CLEAR_CAP |
18    0);
19
20 static uint32 m_use_capability_flags = 0;
21 /*
22   0
23   1
24 */
25
26 /*
27   SPDM_CHALLENGE_REQUEST_NO_MEASUREMENT_SUMMARY_HASH,
28   SPDM_CHALLENGE_REQUEST_TCB_COMPONENT_MEASUREMENT_HASH,
29   SPDM_CHALLENGE_REQUEST_ALL_MEASUREMENTS_HASH
30 */
31 static uint8 m_use_measurement_summary_hash_type =
32   SPDM_CHALLENGE_REQUEST_ALL_MEASUREMENTS_HASH;
33
34 static uint8 m_use_slot_id = 0;
35 static uint8 m_use_slot_count = 3;
36
37 /*
38   SPDM_KEY_UPDATE_ACTION_REQUESTER
```

```
39   SPDM_KEY_UPDATE_ACTION_RESPONDER
40   SPDM_KEY_UPDATE_ACTION_ALL
41 */
42 spdm_key_update_action_t m_use_key_update_action =
       SPDM_KEY_UPDATE_ACTION_ALL;
43
44 static uint32 m_use_hash_algo;
45 static uint32 m_use_measurement_hash_algo;
46 static uint32 m_use_asym_algo;
47 static uint16 m_use_req_asym_algo;
48
49 /*
50   SPDM_MEASUREMENT_BLOCK_HEADER_SPECIFICATION_DMTF,
51 */
52 static uint8 m_support_measurement_spec =
53   SPDM_MEASUREMENT_BLOCK_HEADER_SPECIFICATION_DMTF;
54
55 /*
56   SPDM_ALGORITHMS_BASE_HASH_ALGO_TPM_ALG_SHA_512,
57   SPDM_ALGORITHMS_BASE_HASH_ALGO_TPM_ALG_SHA_384,
58   SPDM_ALGORITHMS_BASE_HASH_ALGO_TPM_ALG_SHA_256,
59 */
60 static uint32 m_support_hash_algo =
61              SPDM_ALGORITHMS_BASE_HASH_ALGO_TPM_ALG_SHA_384 |
62              SPDM_ALGORITHMS_BASE_HASH_ALGO_TPM_ALG_SHA_256;
63 /*
64   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_ECDSA_ECC_NIST_P521,
65   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_ECDSA_ECC_NIST_P384,
66   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_ECDSA_ECC_NIST_P256,
67   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSAPSS_4096,
68   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSASSA_4096,
69   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSAPSS_3072,
70   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSASSA_3072,
71   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSAPSS_2048,
72   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSASSA_2048,
73 */
74 static uint32 m_support_asym_algo =
75   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_ECDSA_ECC_NIST_P384 |
76   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_ECDSA_ECC_NIST_P256;
77 /*
78   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSAPSS_4096,
79   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSASSA_4096,
80   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSAPSS_3072,
81   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSASSA_3072,
82   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSAPSS_2048,
83   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSASSA_2048,
84   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_ECDSA_ECC_NIST_P521,
```

```
85   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_ECDSA_ECC_NIST_P384,
86   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_ECDSA_ECC_NIST_P256,
87 */
88 static uint16 m_support_req_asym_algo =
89   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSAPSS_3072 |
90   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSAPSS_2048 |
91   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSASSA_3072 |
92   SPDM_ALGORITHMS_BASE_ASYM_ALGO_TPM_ALG_RSASSA_2048;
93 /*
94   SPDM_ALGORITHMS_DHE_NAMED_GROUP_FFDHE_4096,
95   SPDM_ALGORITHMS_DHE_NAMED_GROUP_FFDHE_3072,
96   SPDM_ALGORITHMS_DHE_NAMED_GROUP_FFDHE_2048,
97   SPDM_ALGORITHMS_DHE_NAMED_GROUP_SECP_521_R1,
98   SPDM_ALGORITHMS_DHE_NAMED_GROUP_SECP_384_R1,
99   SPDM_ALGORITHMS_DHE_NAMED_GROUP_SECP_256_R1,
100 */
101 static uint16 m_support_dhe_algo =
    SPDM_ALGORITHMS_DHE_NAMED_GROUP_SECP_384_R1 |
102         SPDM_ALGORITHMS_DHE_NAMED_GROUP_SECP_256_R1 |
103         SPDM_ALGORITHMS_DHE_NAMED_GROUP_FFDHE_3072 |
104         SPDM_ALGORITHMS_DHE_NAMED_GROUP_FFDHE_2048;
105 /*
106   SPDM_ALGORITHMS_AEAD_CIPHER_SUITE_AES_256_GCM,
107   SPDM_ALGORITHMS_AEAD_CIPHER_SUITE_AES_128_GCM,
108   SPDM_ALGORITHMS_AEAD_CIPHER_SUITE_CHACHA20_POLY1305,
109 */
110 static uint16 m_support_aead_algo =
111   SPDM_ALGORITHMS_AEAD_CIPHER_SUITE_AES_256_GCM |
112   SPDM_ALGORITHMS_AEAD_CIPHER_SUITE_CHACHA20_POLY1305;
113 /*
114   SPDM_ALGORITHMS_KEY_SCHEDULE_HMAC_HASH,
115 */
116 static uint16 m_support_key_schedule_algo =
    SPDM_ALGORITHMS_KEY_SCHEDULE_HMAC_HASH;
```

APÊNDICE B – *Shell Script* para Coletar o Tempo de Inicialização do HD

Listing B.1 – A emulação é finalizada após 20 segundos de execução e é repetida por um número de vezes fornecido como argumento

```
#!/bin/zsh

LOG=/home/offreitas/logs/qemu.log
TIME=/home/offreitas/logs/time.log
QEMU=/home/offreitas/riscv/qemu/build/riscv64-softmmu/qemu-system-riscv64
CMD="$QEMU \
      -smp 2 \
      -nographic \
      -m 8G \
      -M virt \
      -bios /home/offreitas/riscv/opensbi/build/platform/generic/firmware/fw_payload.elf \
      -drive file=/home/offreitas/riscv/disk.img,format=raw,id=hd0 \
      -device virtio-blk-device,drive=hd0 > $LOG"

if [[ -f $TIME ]]
then
  rm $TIME
fi

for i in $(seq 1 $1)
do
  timeout --foreground 20s zsh -c $CMD
  cat $LOG | grep -oaE "Elapsed time.+" >> $TIME
done
```

APÊNDICE C – Makefile para Otimização de Tempo

```
SHELL=/bin/bash
WORKSPACE=$(shell pwd)
NPROC=$(shell nproc)
BIN_DIR=${WORKSPACE}/buildroot/output/host/bin
CC_RISCV64=${BIN_DIR}/riscv64-linux-
SPDM_DIR=${HOME}/riscv/libspdm
SPDM_BUILD_DIR=${SPDM_DIR}/build_uboot
CLEANERS=buildroot-clean opensbi-clean uboot-clean

.PHONY:
  all clean linux-rebuild opensbi uboot check-cross-compile ${CLEANERS}
  check-uboot

check-uboot:
  @echo "Checking if u-boot.bin exists..."
ifneq ($(wildcard ${WORKSPACE}/u-boot/u-boot.bin),)
  @echo "u-boot.bin exists"
else
  @echo "Error: u-boot.bin wasn't found. Compile u-boot first."; exit 2
endif

check-cross-compile:
  @echo "Checking if cross compiler exists..."
ifneq ("$(wildcard ${CC_RISCV64}*)","")
  @echo "Cross compiler exists"
else
  @echo "Error: riscv64-linux- wasn't found. Compile buildroot first.";
    exit 2
endif

buildroot: buildroot-clean
  $(MAKE) -C buildroot/ qemu_riscv64_virt_defconfig BR2_JLEVEL=${NPROC}
  $(MAKE) -C buildroot/ BR2_JLEVEL=${NPROC}

linux-rebuild: check-cross-compile
  $(MAKE) -C buildroot/ linux-rebuild BR2_JLEVEL=${NPROC}

uboot: check-cross-compile
  if [ ! -e ${WORKSPACE}/u-boot/.config ] ; then $(MAKE) -C u-boot/
    CROSS_COMPILE=${CC_RISCV64} qemu-riscv64_smode_defconfig -j${NPROC};
```

```
    fi
39  $(MAKE) -C u-boot/ CROSS_COMPILE=${CC_RISCV64} SPDM_DIR=${SPDM_DIR}
    SPDM_BUILD_DIR=${SPDM_BUILD_DIR} -j${NPROC}
40
41 opensbi: check-cross-compile check-uboot
42  $(MAKE) -C opensbi/ CROSS_COMPILE=${CC_RISCV64} PLATFORM=generic
    FW_PAYLOAD_PATH=${WORKSPACE}/u-boot/u-boot.bin -j${NPROC}
43
44 all: clean buildroot uboot opensbi
45
46 buildroot-clean:
47  $(MAKE) -C buildroot/ distclean
48
49 uboot-clean:
50  $(MAKE) -C u-boot/ distclean
51
52 opensbi-clean:
53  $(MAKE) -C opensbi/ clean
54
55 clean: ${CLEANERS}
```

Listing C.1 – Makefile para otimização do tempo de coleta dos resultados.

APÊNDICE D – Formas de Requisitar Medidas no Padrão SPDM

```
// Request all measurements at a time
measurement_record_length = sizeof(measurement_record);
status = spdm_get_measurement(
                    desc->spdm_context,
                    0,
                    0x1,
                    0xFF,
                    m_use_slot_id & 0xF,
                    &number_of_block,
                    &measurement_record_length,
                    measurement_record
                    );
if (RETURN_ERROR(status)) {
    log_err("[SPDM @ U-Boot]: Error getting measurements 0x%02X.\n", (uint32)status);
    goto out_free_spdm;
}

// Request measurements one by one
status = spdm_get_measurement(
                    desc->spdm_context,
                    0,
                    0,
                    0,
                    m_use_slot_id & 0xF,
                    &number_of_blocks,
                    NULL,
                    NULL
                    );
if (RETURN_ERROR(status)) {
    log_err("[SPDM @ U-Boot]: Error requesting number of available measurements 0x%02X.\n", (uint32)status);
    goto out_free_spdm;
}

request_attribute = 0;
for (index = 1; index <= number_of_blocks; index++) {
    if (index == number_of_blocks) {
        request_attribute = 0x1;
    }
```

```
39      measurement_record_length = sizeof(measurement_record);
40      status = spdm_get_measurement(
41                           desc->spdm_context,
42                           desc->session_id,
43                           request_attribute,
44                           index,
45                           m_use_slot_id & 0xF,
46                           &number_of_block,
47                           &measurement_record_length,
48                           measurement_record
49                           );
50      if (RETURN_ERROR(status)) {
51          log_err("[SPDM @ U-Boot]: Error getting measurement 0x%02X.\n",
    (uint32)status);
52          goto out_free_spdm;
53      }
54  }
```