Gustavo Cerqueira Bastos

Implementação em hardware do protocolo SPDM

São Paulo, SP

2023

Gustavo Cerqueira Bastos

Implementação em hardware do protocolo 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 Antonio Simplicio Junior

Coorientador: Prof. Dr. Bruno de Carvalho Albertini

São Paulo, SP

2023

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

Catalogação-na-publicação

Bastos, Gustavo
Implementação em hardware do protocolo SPDM / G. Bastos -- São Paulo, 2023.
72 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Segurança 2.Hardware I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

Ao Deus triúno, meu Senhor e autor da vida. Aos meus pais, Silvio e Anália pelo amor e apoio durante todos estes anos.

Agradecimentos

Primeiramente ao Deus triúno, Pai, Filho e Espiríto Santo, pela fé, vida e esperança.

Agradeço a minha família, minha mãe Anália, meu pai Silvio e irmã Cibele pelo amor e carinho durante todos estes anos.

Agradeço aos amigos e colegas pelos bons momentos.

Agradeço aos professores Marcos Simplicio e Bruno Albertini pela boa orientação e paciência durante todo este trabalho.

Resumo

Security Protocol and Data Model (SPDM) é um protocolo de segurança desenvolvido para evitar ataques de segurança em baixo nível, especialmente, aqueles ataques realizados em *firmware* ou em nível de barramento. O protocolo é hoje uma alternativa moderna de segurança em baixo nível com implementação em aberta (livre), com desempenho testado e comprovado em ambiente emulado. No entanto, ambientes emulados podem sofrer de distorções no modelo computacional, aplicando uma maior sobrecarga do que efetivamente se espera. Este trabalho consiste na primeira implementação conhecida deste protocolo em um hardware dedicado, abrindo novos caminhos para a utilização deste modelo de segurança de uma forma efetiva em aplicações embarcadas e de de baixo nível.

Palavras-chave: Segurança, Protocolo, SPDM, Implementação, Hardware

Abstract

Security Protocol and Data Model (SPDM) is a security protocol designed to prevent low-level security attacks, especially those performed at the firmware or bus level. It has been a low-level security alternative with an open source implementation, tested and measured in an emulated environment. However, emulated environments can cause distortions in the model, applying a greater overload to the computational system than expected, so this work has as its study the implementation of this protocol in dedicated hardware, with the possibility of opening new paths for the use of this protocol. new type of security in an effective way as in embedded applications.

Keywords: Security, Protocol, SPDM, Implementation, Hardware.

Lista de ilustrações

Figura 1 – Modelo de comunicação do SPDM 16
Figura 2 – Flags de compilação com a biblioteca Newlib 23
Figura 3 – Flags de compilação com a biblioteca Linux 24
Figura 4 – Esquema em blocos do SoC. (LINUX. . . , 2023) 28
Figura 5 – Arquivo JSON para inicialização do sistema 28
Figura 6 – Sistema Operacional em execução 29
Figura 7 – Endereço da interface Ethernet na máquina hospedeira 42
Figura 8 – Configuração da placa Ethernet na FPGA seguido de testes com o comando de ping para máquina hospedeira 43
Figura 9 – Comando `ping` realizado da máquina hospedeira para a FPGA 43
Figura 10 – Construção do registrador dentro da placa de rede Ethernet 45
Figura 11 – Descrição das funções de manipulação do registrador 46
Figura 12 – Descrição em RTL do registrador de controle do SPDM ligada aos LEDs da FPGA 46
Figura 13 – Função para manipulação do registrador do SPDM construída dentro do *firmware* 47
Figura 14 – Controle do Registrador SPDM pela BIOS 48
Figura 15 – Makefile libspdm_litex.mk 49
Figura 16 – Adição da biblioteca junto as flags de compilação na Common.mak . . 50
Figura 17 – Adição do caminho da biblioteca junto as flags de compilação na Common.mak 51
Figura 18 – Adiação dos binários da biblioteca LibSPDM 52
Figura 19 – Flag de compilação -L adicionada no Makefile 52
Figura 20 – Espaços de memória no SoC 53
Figura 21 – Modificação do Linker.ld para comportar o símbolo Heap 54
Figura 22 – Mudança nos tamanhos padrão da memória SRAM e ROM. 55
Figura 23 – Funções da BIOS declaradas 55
Figura 24 – Função de SPDM Requester dentro da BIOS 56
Figura 25 – Makefile com adição da compilação das funções de SPDM 57
Figura 26 – Funções SPDM adicionadas como comandos na BIOS 57
Figura 27 – Flags de compilação para a LibSPDM no Kernel Linux 58
Figura 28 – Mudanças na função de bignum.c 59
Figura 29 – Segunda mudança na função bignum.c 60
Figura 30 – Configuração de toolchain externa dentro do Buildroot 61
Figura 31 – Makefile para LibSPDM - 1 63
Figura 32 – Makefile para LibSPDM - 2 63

Figura 33 – Makefile para inclusão de headers 64
Figura 34 – Adição da flag de compilação 65
Figura 35 – Estabelecimento de dois *responders* 67
Figura 36 – Uso de memória para o SPDM dentro do Kernel Linux 68
Figura 37 – Possível estrutura de um trabalho futuro 70

Lista de quadros

Quadro 1 – Comando de construção do SoC 27
Quadro 2 – Configurações e Funções da LibSPDM - 1 31
Quadro 3 – Configurações e Funções da LibSPDM - 2 32
Quadro 4 – Configurações e Funções da LibSPDM - 3 33
Quadro 5 – Configurações e Funções da LibSPDM - 4 34
Quadro 6 – Configurações e Funções da LibSPDM - 5 35
Quadro 7 – Configurações e Funções da LibSPDM - 6 36
Quadro 8 – Configurações e Funções da LibSPDM - 7 37
Quadro 9 – Configurações e Funções da LibSPDM - 8 38
Quadro 10 – Configurações e Funções da LibSPDM - 9 39

Lista de tabelas

Tabela 1 – Uso de Memória 66

Lista de abreviaturas e siglas

SPDM	Security Protocol and Data Model
RISC	Reduced Instruction Set Computer
SoC	System on Chip
FPGA	Field Programmable Gate Array

Sumário

1 INTRODUÇÃO 13
1.1 Motivação 13
1.2 Objetivos 14
1.3 Justificativa 14
1.4 Organização do Trabalho 15

2 ASPECTOS CONCEITUAIS 16
2.1 Security Protocol and Data Model 16
2.2 RISC-V 17

3 MÉTODO DO TRABALHO 19

4 ESPECIFICAÇÃO DE REQUISITOS 20

5 DESENVOLVIMENTO DO TRABALHO 21
5.1 Tecnologias Utilizadas 21
5.1.1 Construção inicial do SoC 21
5.2 Projeto e Implementação Inicial 22
5.2.1 Construção do compilador 22
5.2.2 Construção do sistema operacional e do bootloader 23
5.2.3 Construção inicial do hardware 26
5.2.4 Inicialização do Kernel Linux no SoC 28
5.2.5 LibSPDM 29
5.3 Testes e Avaliação 41
5.3.1 Testes da Ethernet 41
5.3.2 Testes do registrador de controle do SPDM através da BIOS 43
5.4 Desenvolvimento - Hardware com SPDM 48
5.4.1 Integração da LibSPDM na BIOS 48
5.4.2 Integração da LibSPDM no driver da Ethernet 58

6 RESULTADOS E CONSIDERAÇÕES FINAIS 66
6.1 Conclusões do Projeto de Formatura 66
6.2 Contribuições 68
6.3 Perspectivas de Continuidade 69

REFERÊNCIAS 71

1 Introdução

Com o aumento do uso de dispositivos computacionais e igual aumento das necessidades de segurança, surgiram questões sobre a preocupação de ataques internos que se aproveitam do *firmware*. Estes ataques podem ser realizados por fabricantes maliciosos ou com segurança comprometida pois têm a possibilidade de abusar de seu poder na cadeia de suprimentos para adicionar elementos não desejáveis. O SPDM ou *Security Protocol and Data Model* (DTMF, 2022) atualmente é um dos modos mais gerais para se defender de ataques realizados em baixo nível, ao propor modelos de dados e modos de comunicação em barramentos com acordo de chaves. O seguinte trabalho se propõe a explorar possibilidades quanto ao uso do SPDM, com a probabilidade de abertura para uso em aplicações comerciais com dados sensíveis.

1.1 Motivação

Ataques realizados em baixo nível, podem causar danos significativos para a vítima através de eventos como, por exemplo, um vazamento de memória com dados significativos, estes ataques são amplamente registrados na literatura, um exemplo disto são os ataques de espectro eletromagnético (THU et al., 2023), no qual as emissões de ondas eletromagnéticas sob um barramento do tipo AXI (Advanced eXtensible Interface) podem ser utilizas para realizar vazamento de dados. Uma das propostas que apareceram recentemente na indústria foi a proposta do protocolo “Security Protocol and Data Model” (SPDM), que cria mecanismos e formatos para autenticação do *hardware* e do *firmware*, o que entrega uma maior segurança contra fabricantes com intenções de manipulação de um *hardware*. Além disso, o SPDM acabou incluindo também um protocolo de acordo de chaves para proteger a integridade dos dados transmitidos entre os componentes do hardware, isto é, a comunicação realizada em barramento. O protocolo possui uma implementação *open source* disponível (LIBSPDM, 2023).

Um bom protocolo de segurança não deve ser apenas seguro, é também necessário que o mesmo tenha um desempenho aceitável, isto é, a sobrecarga imposta no sistema deve ser balanceada pois, caso contrário, seu uso é inviável. Neste sentido, existem trabalhos publicados cujo objetivo foi medir a sobrecarga imposta sobre o ambiente computacional, mas todos foram realizados sobre ambientes emulados (ALVES; ALBERTINI; SIMPLICIO, 2022). É latente a necessidade de medições similares em uma implementação física, para que não seja mais necessário se utilizar um ambiente emulado e por fim obter métricas com uma maior precisão que validem o modelo de segurança proposto pelo SPDM.

1.2 Objetivos

Um bom protocolo de autenticação não deve ser apenas seguro, o mesmo deve possuir um bom desempenho para evitar sobrecargas no sistema computacional que venham a desestimular o uso do protocolo, assim, o SPDM já possui uma avaliação de seu desempenho (ALVES; ALBERTINI; SIMPLICIO, 2022), porém, neste trabalho anterior as métricas obtidas foram realizadas sobre um sistema emulado no `QEMU`, o que acaba por colocar uma sobrecarga extra não imposta pelo protocolo e sim por uma fonte secundária. Neste trabalho o objetivo consiste em eliminar este ambiente emulado e partir para um ambiente real ao construir um *hardware* dedicado no qual use o SPDM.

Em vista das necessidades nos quais um protocolo de segurança deve atender, isto é, principalmente confiabilidade e desempenho, o objetivo deste trabalho é realizar a implementação de um dispositivo físico (hardware) do protocolo SPDM, utilizando como prova o periférico da placa de rede Ethernet e uma BIOS, visto que uma versão em software do protocolo já existe na forma de uma biblioteca *open source*.

A implementação física, que foi materializada em uma FPGA, permite a avaliação da sobrecarga imposta no sistema de uma maneira precisa e satisfatória, visto que não é utilizado um ambiente emulado. Com a implementação, abre-se a possibilidade de se provar a viabilidade do uso do protocolo SPDM para sistemas embarcados em aplicações comerciais sensíveis como, por exemplo, bancos de dados e servidores, isto ao se utilizar este hardware para retirar métricas sobre o uso do protocolo, como, por exemplo, uso de memória.

O objetivo se encontra na área de inovação, com os resultados esperados sendo divulgados em forma de repositório aberto na plataforma GIT.

1.3 Justificativa

Em vista do crescimento e da normalização do uso de computadores de propósito geral e de sistemas embarcados como celulares, *drones* e até mesmo equipamentos de segurança como câmeras, surge uma necessidade de se garantir para o usuário de que suas informações pessoais inseridas no sistema, não estão sendo vazadas ou capturadas por um agente externo através de algum tipo de ataque. A problemática sobre o nível de segurança exigido não está apenas relacionada com agentes externos em nível de software ou rede, mas também com potenciais ataques internos. Devido a quantidade de terceiros fornecedores inseridos na cadeia de produção de um dispositivo físico, é necessário garantir que nenhum deles está abusando de sua posição na cadeia de insumos para realizar ataques, ou seja, de que estes mesmos não estão, por projeto, manipulando componentes para se ter acesso a informações privilegiadas ou sigilosas do usuário com a consequência de se violar

a privacidade como um todo, e com um agravante de ser um ataque de difícil detecção (CUI; COSTELLO; STOLFO, 2013).

Estes ataques, além de difícil detecção, são assim realizados em nível de *firmware*, uma camada de software de baixo nível responsável por controlar os dispositivos físicos. Uma modificação maliciosa em baixo nível pode permitir ataques do tipo lógico, físico ou até mesmo comportamental, em qualquer componente de hardware que dependa de seu correto funcionamento. Estes ataques, por serem de baixo nível, conseguem ser bem sucedidos sem serem detectados por algum tipo de proteção convencional como um antivírus ou *firewalls* (CHOI et al., 2016).

Foram realizadas algumas propostas para solução do problema da autenticação de *firmware* dentro da literatura, como a utilização de arquitetura de hardware para realizar contagem do número de modificações (WANG et al., 2015) com o objetivo de se conseguir mapear quantas foram realizadas até o momento ou até mesmo foi proposto na literatura a utilização da tecnologia de *blockchain* aplicada a inversores inteligentes, o que inclui um sistema computacional (BERE et al., 2021) com o objetivo de se perceber através do *hash* quais versões do *firmware* foram de fato produzidas (suas versões são armazenadas juntamente com o *hash* de quando o *firmware* foi submetido a rede) e se estas devem ser aceitas pelos nós da rede.

O SPDM é uma recente proposta da indústria no qual possui uma implementação em formato de biblioteca *open source*. Este protocolo possui um amplo suporte da indústria o que indica uma oportunidade de se investigar, cientificamente, o verdadeiro potencial de utilização do protocolo, especialmente ao se tratar de sistemas com baixa quantidade de recursos computacionais e de questões gerais de desempenho.

1.4 Organização do Trabalho

Procuramos dividir este trabalho de uma maneira concisa para que qualquer leitor da área de sistemas embarcados possa apreciá-lo. No Capítulo 2, são apresentados os conceitos base no para a sustentação do trabalho. Os conceitos são seguidos de como o trabalho foi realizado, no Capítulo 3, ou seja, quais passos que foram seguidos para se chegar aos resultados finais. O Capítulo 4 continua nesta introdução geral, logo, neste capítulo se denomina quais são os requisitos necessários para a implementação e a conclusão do trabalho. Por fim, os Capítulos 5 e 6 detalham e explicam o que e como foi realizado todo o percurso desta monografia e encerra-se com a conclusão final do trabalho.

2 Aspectos Conceituais

2.1 Security Protocol and Data Model

O protocolo SPDM (*Security Protocol and Data Model*) é um dos principais componentes deste trabalho. O documento define as mensagens, estruturas de dados e sequências de troca de mensagens, que incluem autenticação e provisão das identidades dos hardwares envolvidos, medidas para identificar o *firmware* utilizado e até mesmo trocas de chaves para permitir confiabilidade na comunicação em baixo nível. Dentro do protocolo existe uma nomenclatura chave no modo em que a comunicação acontece, o componente que inicia a comunicação é chamado de *requester*, enquanto o lado solicitado é chamado de *responder*, a conexão entre ambos é estabelecida por meio de um canal seguro, no qual o *requester* solicita e negocia as capacidades de segurança de seu *responder*, por exemplo. Ao longo desta negociação, um conjunto de algoritmos criptográficos podem ser selecionados a partir da intersecção dos algoritmos suportados por ambos. Outro ponto importante de como essa comunicação ocorre é a necessidade de uma identificação do *responder*, determinada por uma assinatura digital. Este protocolo com essas configurações, pode fornecer uma maior segurança em baixo nível, dependendo apenas da confiança entre os fabricantes (fabricantes maliciosos que sejam considerados confiáveis não são identificados pelo protocolo). A Figura 1 mostra um diagrama de blocos de como a comunicação ocorre dentro do protocolo.

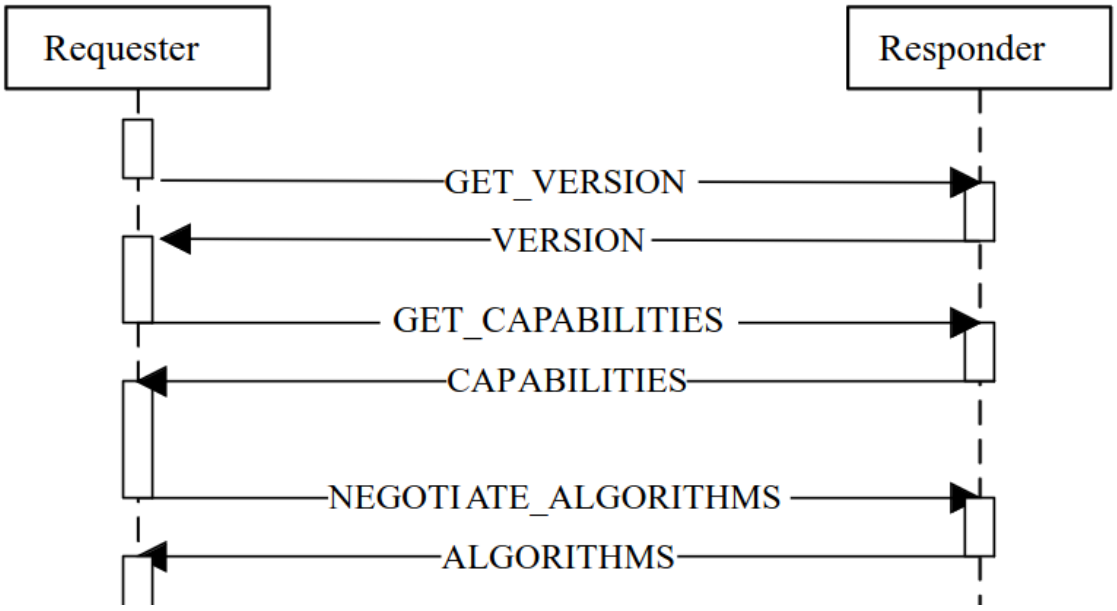

Figura 1 – Modelo de comunicação do SPDM

2.2 RISC-V

O *Reduced Instruction Set Computing* ou simplesmente RISC-V é uma arquitetura de conjunto de instruções para propósito geral, desenvolvido primariamente na Universidade de Berkeley no departamento de Computação, de uso livre e aberto, ou seja, uma arquitetura no qual não se precisa pagar royalties. A escolha desta arquitetura se deu exatamente por este motivo, uso aberto com código disponível que permite modificações caso seja necessário. Além disso, existem várias implementação de Kernel Linux para esta arquitetura, assim como *bootloaders* que podem ser aproveitados para acelerar a etapa de prototipação e evitar inércia inicial. Dificuldades sobre manipulação de uma nova arquitetura foram planejadas, porém não impactaram o projeto.

O processador escolhido não foi prototipado do zero, pois encontramos na pesquisa inicial a plataforma LiteX. Partindo de uma linguagem de descrição (e.g. Verilog) ou até mesmo de Python, a LiteX permite personalizar a geração dos componentes de hardware. Como exemplo, é possível personalizar desde o número de núcleos computacionais, como o barramento será organizado ou até mesmo como a memória *cache* será organizada. O *framework* gera componentes sintetizáveis em RTL, que podem ser alimentados em qualquer ferramenta de síntese que suporte descrições em Verilog

Um dos desafios presentes decorrido do campo de hardware passa pela escolha de um processador de implementação aberta, cujo foi escolhido uma implementação de RISC-V, no qual se tenha a possibilidade de realizar alterações em seu código fonte, o processador também deve ser capaz de suportar um Kernel Linux. A primeiro momento, a plataforma LiteX (KERMARREC et al., 2020) é adequada, visto que a mesma fornece modos de modificação, ao fornecer periféricos compatíveis dentro da própria plataforma, além disto, a plataforma fornece como partir de casos base de processadores, tais como o `Ariane` ou o `Rocket Chip` (ASANOVIć et al., 2016), no qual ao final de sua sintetização é entregado como resultado uma implementação em Verilog do processador escolhido com as modificações desejadas, ou seja, juntamente com os periféricos escolhidos.

Ao se considerar o objetivo deste trabalho, implementar o protocolo SPDM em um hardware dedicado, um SoC, foi utilizada a plataforma LiteX. Esta biblioteca foi escolhida, em detrimento de outras como a Chipyard, devido a possuir suporte a FPGA da Nexys 4 DDR (diferentemente de outras plataformas como a Chipyard), isto acaba por realizar uma retirada da inércia de prototipação de modo a retirar a necessidade de se construir um modelo da FPGA dentro da LiteX, de modo que não houveram perdas, isto pois, a LiteX, assim como a Chipyard, possuem periféricos desejáveis, a saber, a capacidade de comunicação serial e a placa de rede Ethernet, logo, há um ganho consideravelmente maior ao se utilizar a LiteX. Com a plataforma de protipação escolhida, o processador foi o passo seguinte, dentre os quais o `Rocket Chip` acabou por ser selecionado, pois, além de ser uma implementação aberta de Risc-V, possui um bom suporte dado pela comunidade,

de modo que este processador possui fácil compatiblidade com o `QEMU`, logo, isto acaba por ajudar no momento de testes em ambiente emulado, sendo assim, o `Rocket Chip` foi selecionado e construído dentro da plataforma LiteX juntamente com os periféricos.

3 Método do trabalho

Escolhido adequadamente o processador, nos deparamos com a escolha dos periféricos de teste. Apesar da prova de conceito poder ser realizada com qualquer periférico, selecionamos uma placa de rede Ethernet devido a popularidade de tal periférico em roteadores, sistemas embarcados e dispositivos IoT (nota-se que não há diferença relevante entre uma placa de rede cabeada e sem fio do ponto de vista de comunicação com o processador). Os requisitos que limitaram nossa busca foram: deve ter código aberto e livre, quando sintetizada, deve caber em uma placa FPGA NEXYS 4 da Xilinx, e sua pinagem deve ser realizada de tal modo em que há a possibilidade de depuração com o uso de pinos de propósito geral. Um outro ponto a se destacar se refere ao cuidado de que deve se ter memória disponível não apenas para os componentes físicos, mas também para o sistema operacional.

Em relação ao sistema operacional, escolhemos um Kernel Linux construído a partir de uma implementação da plataforma *Buildroot* (BUILDROOT-DOCS, 2020), pois já possui as características necessárias para o seu uso, como o endereçamento de periféricos e o suporte do *bootloader* OpenSBI (OPENSBI, 2020). Em primeiro momento, o Kernel Linux foi executado em ambiente simulado, utilizando o `QEMU` (QEMU, 2020) para se garantir funcionalmente que as configurações do Kernel foram realizadas de maneira correta, especialmente no que se refere ao uso do SPDM. Uma parte relevante deste trabalho se refere a análise de periféricos com implementação não-segura, que foram escolhidos igualmente de acordo com a disponibilidade de implementações abertas e livres para que possam ser analisados internamente.

4 Especificação de Requisitos

Um dos maiores problemas quando se trata de uma especificação são os requisitos necessários para o correto funcionamento da implementação. Deste modo, elencamos alguns destes requisitos. O primeiro destes, refere-se a velocidade de processamento do SPDM: uma prova de conceito implementada em um dispositivo físico deve ser de tal modo que seja possível visualizar o correto funcionamento do dispositivo. Deste modo, identificamos a necessidade de comunicação serial com elementos visuais para identificação do protocolo, sejam estes valores numéricos de códigos de resposta ou simplesmente demonstrações de inicialização.

O dispositivo físico deve ser implementado em uma FPGA, portanto a memória disponível deve ser suficiente, ou seja, a memória deve conter a implementação física sintetizada a partir de uma descrição em Verilog e igualmente um Kernel Linux com seu respectivo *bootloader* para inicialização do sistema operacional. Os periféricos que forem acoplados na implementação devem ser de código libre e aberto, funcionais para suas respectivas características. Questões de segurança nesta etapa serão desenvolvidas pelo próprio protocolo do SPDM, não havendo a necessidade de maior atenção.

Resumidamente, os requisitos levantados são a implementação de toda plataforma disponível de forma livre e aberta, com o suporte aos periféricos e sistema operacional escolhidos, e cuja síntese seja suportada na placa disponível para testes. Não há *stakeholders* pois trata-se de uma prova de conceito. A derivação de pesquisa recai na hipótese de que o protocolo SPDM provê segurança no nível de barramento com uma sobrecarga no *firmware* (dispositivo) e no software (*bootloader* e sistema operacional) aceitável.

5 Desenvolvimento do Trabalho

5.1 Tecnologias Utilizadas

5.1.1 Construção inicial do SoC

Definido o objetivo de se construir um *System on Chip* (SoC) com a execução do SPDM em um hardware, pode-se separar as tecnologias utilizadas em duas categorias distintas: hardware e software.

Em relação ao hardware será utilizado uma FPGA da Digilent, a Nexys 4 DDR. Esta placa conta com um FPGA e uma memória de 128 MiB utilizada para armazenar os conteúdos de desenvolvimento do hardware, o que inclui a implementação física do processador e de seus periféricos como placa de rede Ethernet e comunicação serial. A descrição do hardware, escrita em linguagem Verilog, foi gerada pela plataforma de prototipação "LiteX", uma plataforma *open source* capaz de gerar um processador do tipo "Rocket Chip"com uma placa de rede Ethernet funcional, e adaptada às necessidades do projeto. Em relação ao software, utilizamos as ferramentas: `QEMU`, *Buildroot* e `OpenSBI`. As duas primeiras ferramentas são responsáveis por gerar o sistema de arquivos e o Kernel Linux, ou seja, o sistema operacional do SoC.

A arquitetura Risc-V devido a sua natureza aberta, acaba por possuir variações na sua arquitetura que alteram o momento de compilação de um software (THE. . . , 2017), isto é realizado na compilação através das *flags* -mabi e -march, por exemplo, existem as variações de arquitetura (march/mabi) rv64imac/lp64 e a rv64imafdc/lp64d, ou seja, a *flag* de -march controla conjunto de instruções de usuário com os registradores que estão disponíveis e a *flag* -mabi controla a chamada de funções, isto é, o mapeamento de chamadas para certos registradores, assim, este sistema operacional deve ser compatível com o conjunto de instruções do processador escolhido, que no caso é a combinação de rv64imac/lp64 (march/mabi), logo, para se garantir que as configurações adequadas estão sendo aplicadas, utiliza-se o software gerado pelo *Buildroot* no `QEMU`, um ambiente emulado do hardware que foi construído pela LiteX, de modo que o desenvolvimento de hardware e software podem ser paralelizados. Assim, com o sistema operacional em pleno funcionamento é natural que se precise de um *bootloader* para realizar o carregamento inicial das configurações do Kernel. Utilizamos a `OpenSBI` para compilar o *bootloader* com as informações necessárias do processador, por exemplo, a árvore de periféricos que implementamos (*Device Tree*).

Por fim, utilizamos o software comercial da Xilinx, o Vivado. Este software é um sintetizador de hardware que gera o *bitstream* necessário para configuração do FPGA a

partir da descrição Verilog. Todas nossas modficações foram aplicadas nesta etapa, ou seja, após a geração de código da LiteX e antes da síntese pelo Vivado. Além da síntese, o Vivado traz um programador que é capaz de configurar a placa física escolhida para prototipação.

5.2 Projeto e Implementação Inicial

5.2.1 Construção do compilador

Todo o seguinte trabalho foi construído baseado na arquitetura RISC-V. No entanto, como os computadores de trabalho são de arquitetura x86_64, usamos a técnica de "cross compiler", ou seja, um compilador que executa em uma máquina x86_64, porém gera código objeto para a arquitetura RISC-V.

O compilador utilizado foi o RISC-V GNU Toolchain (RISC-V. . . , 2023). A ferramenta possui pré requisitos de algumas bibliotecas quando se utiliza o sistema operacional Ubuntu, que podem ser instaladas da seguinte maneira:

```
sudo apt-get install autoconf automake autotools-dev curl python3
python3-pip libmpc-dev libmpfr-dev libgmp-dev gawk build-essential
bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
ninja-build git cmake libglib2.0-dev
```

Com os pré requisitos instalados é possível instalar o compilador para o sistema operacional Ubuntu. Começamos com download a partir de seu repositório GitHub em sua versão 2023.06.09, assim, para manter a compatibilidade (devido as mudanças que podem ocorrer em um repositório *open source* pode ocorrer de haver variações no modo em que ocorre a compilação) com o momento em que este trabalho foi realizado:

```
$ mkdir riscv
$ cd riscv
$ git clone https://github.com/riscv/riscv-gnu-toolchain
$ git checkout tags/2023.06.09
```

Com o repositório baixado, pode-se configurar o mesmo dentro do caminho `riscv/riscv-gnu-toolchain`. A configuração escolhida permite que o compilador gerado tenha suporte a uma série de combinações de arquiteturas e ABIs (*Application Binary Interface*) definidas por padrão e ainda adicionamos suporte a dois tipos de bibliotecas, a saber a "Newlib"e a "Linux", representadas por `riscv64-unknown-elf-` e `riscv64-unknown-linux-gnu-`. A primeira é utilizada para compilações em *bare metal*

(sem suporte de sistema operacional), como no caso dos binários do SoC, e a segunda para compilações de códigos que assumes a presença de um sistema operacional, como o Kernel Linux e o *bootloader*.

Para o suporte à biblioteca "Newlib", cujos binários ficarão no caminho `/opt/riscv/` definido pela *flag* --prefix", usamos os seguintes parâmetros:

```
$ ./configure --prefix=/opt/riscv --enable-multilib
$ make
```

Caso se deseje ver se o resultado da instalação foi bem sucedido, basta ir para o caminho `/opt/riscv/bin` e executar os seguintes comandos:

```
./riscv64-unknown-elf-gcc --verbose
./riscv64-unknown-elf-gcc --print-multi-lib
```

O primeiro mostra as configurações utilizadas e o segundo mostra as combinações de arquiteturas e ABIs disponíveis como descrito anteriormente. Os resultados devem ser esperados como na seguinte figura 2:

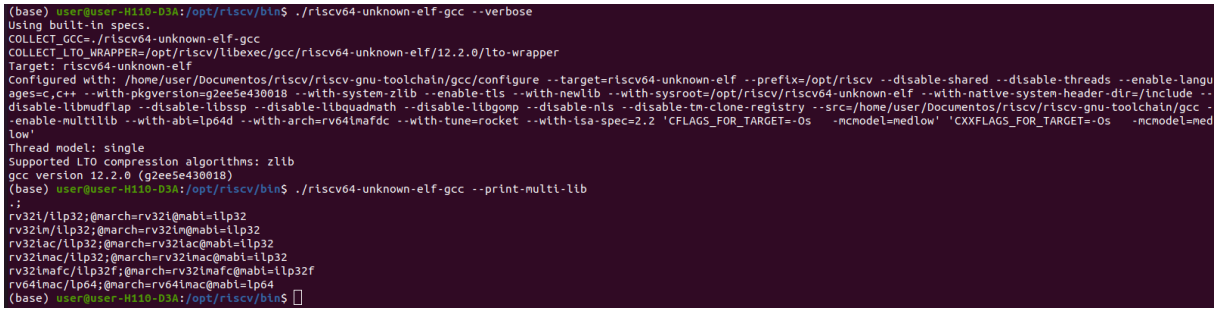

Figura 2 – Flags de compilação com a biblioteca Newlib

Em seguida, a configuração para a biblioteca "Linux", no qual segue o mesmo padrão da configuração anterior:

```
$ ./configure --prefix=/opt/riscv --enable-multilib
$ make linux
```

O resultado é como na Figura 3:

5.2.2 Construção do sistema operacional e do bootloader

Para utilização do SoC, é necessário a construção do sistema operacional que o mesmo irá executar. Este sistema é um Kernel Linux com operações básicas disponíveis,

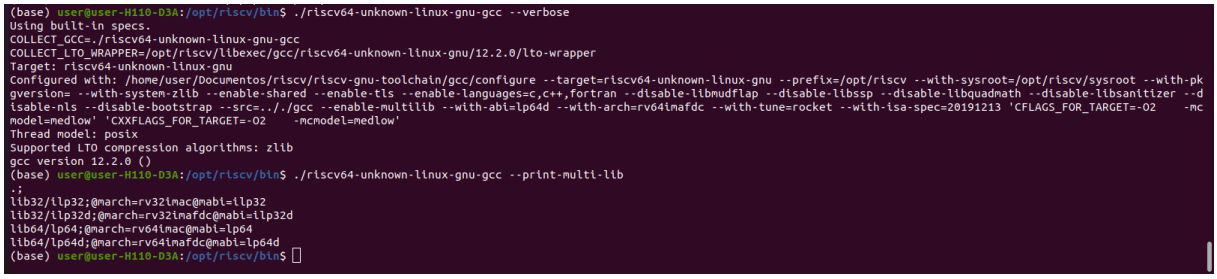

Figura 3 – Flags de compilação com a biblioteca Linux

construído em duas partes: uma realizada na plataforma para sistemas embarcados *Buildroot*, que irá construir a imagem do Kernel em si, e a outrana plataforma *BusyBox* que gera o sistema de arquivos. O *bootloader* será compilado externamente.

Começamos com a construção do sistema de arquivos. A plataforma LiteX já possui configurações padrões que inclui os *drivers* da placa de rede e as configurações necessárias para gerar o sistema de arquivos. Optamos por aceitar as configurações padrões após análise, portanto bastou gerar o `initramfs` dentro do *BusyBox*, ou seja, o sistema de arquivos temporário utilizado para inicialização do sistema. O *BusyBox* utilizado foi a versão 1.33.2 e sua instalação inicial pode ser realizada com os comandos abaixo. Deve-se atentar para o uso das configurações corretas, que estão codificadas no arquivo disponível no GitHub, no caminho `configs/busybox-1.33.2`.

```
$ curl https://busybox.net/downloads/busybox-1.33.2.tar.bz2 | tar xfj -
$ cp configs/busybox-1.33.2/.config busybox-1.33.2/.config
$ cd busybox-1.33.2
$ make CROSS_COMPILE=/opt/riscv/bin/riscv64-unknown-linux-gnu-
```

Com as configurações corretas aplicadas sobre o *BusyBox*, o sistema de arquivos pode ser gerado. Os comandos necessários para o gerar podem ser sumarizados no seguinte bloco de comandos:

```
$ mkdir initramfs
$ pushd initramfs
$ mkdir -p bin sbin lib etc dev home proc sys tmp mnt nfs root \
usr/bin usr/sbin usr/lib
$ cp ../busybox-1.33.2/busybox bin/
$ ln -s bin/busybox ./init
$ cat > etc/inittab <<- "EOT"
::sysinit:/bin/busybox mount -t proc proc /proc
::sysinit:/bin/busybox mount -t devtmpfs devtmpfs /dev
::sysinit:/bin/busybox mount -t tmpfs tmpfs /tmp
```

```
::sysinit:/bin/busybox mount -t sysfs sysfs /sys
::sysinit:/bin/busybox --install -s
/dev/console::sysinit:-/bin/ash
$ EOT
$ fakeroot <<- "EOT"
$ find . | cpio -H newc -o > ../initramfs.cpio
$ EOT
$ popd
```

Ainda dentro do *BusyBox*, deve-se já configurar o Kernel Linux. Apesar do Kernel em si ser construído apenas dentro do *Buildroot*, estas configurações são realizadas no *BusyBox* e posteriormente serão importadas para o *Buildroot*, portanto não há duplicidade de trabalho nesta etapa. As configurações podem ser importadas do repositório GitHub, no caminho `configs/busybox-1.33.2/linux/.config`. Este arquivo `.config` deve ser colocado dentro do caminho `busybox-1.33.2/linux`, com a atenção de que uma configuração específica que deve ser alterada devido a um caminho do sistema de arquivos onde está o `initramfs` construído anteriormente. Dentro do .config deve-se alterar:

```
CONFIG_INITRAMFS_SOURCE="PATH/TO/busybox-1.33.2/initramfs.cpio"
```

Com o sistema de arquivos construído e já configurado, a próxima etapa consiste em construir a imagem inicial do Kernel, que será carregada na memória RAM do SoC para ser executada. Com a utilização do *Buildroot* o mesmo deve ser configurado de modo que as configurações do Kernel sejam aquelas criadas na etapa anterior pelo *BusyBox*, o que inclui o sistema de arquivos gerado juntamente com a utilização dos *drivers* da LiteX. O *Buildroot* foi construído com sua versão de 2023.05.1 através da seguinte série de comandos:

```
$ mkdir buildroot_riscv64
$ cd buildroot_riscv64
$ wget https://git.busybox.net/buildroot/snapshot/buildroot-2023.05.1.tar.bz2
$ tar -xjf buildroot-2023.05.1.tar.bz2
$ cd buildroot-2023.05.1
```

O *Buildroot* funciona com um esquema similar do *BusyBox*, ou seja, se utiliza de arquivos de configuração para construir o Kernel corretamente. As configurações devem ser importadas para evitar quaisquer erros e algumas destas configurações devem ser atentadas pois também dependem de outros arquivos.

```
BR2_DEFCONFIG="PATH/TO/buildroot_riscv64/buildroot-2023.05.1/configs/
```

```
        qemu_riscv64_virt_defconfig"
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="PATH/TO/busybox-1.33.2/linux/.config"
BR2_PACKAGE_BUSYBOX_CONFIG="PATH/TO/busybox-1.33.2/.config"
BR2_LINUX_KERNEL_CUSTOM_DTS_PATH="PATH/TO/nexys4ddr.dts"
```

Por fim, a última peça de software necessário para utilização do SoC é o *bootloader*. Neste caso foi utilizado o `OpenSBI 0.8` (OPENSBI, 2020) devido a sua vasta documentação e utilização dentro dos ambientes de RISC-V. O *bootloader* é construído com a seguinte série de comandos:

```
$ git clone https://github.com/litex-hub/opensbi
$ cd opensbi
$ git checkout 84c6dc17f7d41c5c02760a5533d7268b57369837
$ export PATH=$PATH:/opt/riscv/bin
$ make CROSS_COMPILE=riscv64-unknown-linux-gnu- PLATFORM=generic \
    FW_FDT_PATH=PATH/TO/nexys4ddr.dtb FW_JUMP_FDT_ADDR=0x82400000
```

O binário do *bootloader* da `OpenSBI` se encontrará no caminho `/opensbi/build/platform/-generic/firmware/fw_jump.bin`.

5.2.3 Construção inicial do hardware

A plataforma LiteX funciona como uma intermediária entre diversas bibliotecas como a Migen, no qual permite a descrição de um *hardware* em uma linguagem de alto nível, como Python e a partir deste script gera arquivos de Verilog com o SoC, a Migen foi integrada junto a diversos processadores, bibliotecas de software, periféricos e FPGAs de modo que ao se utilizar os comandos implementados pela LiteX é possível passar pelos passos intermediários destas bibliotecas e gerar um SoC funcional diretamente, logo, se retira a inércia inicial destas bibliotecas. Assimm, o seguinte comando foi utilizado para gerar o código fonte necessário para o *hardware*:

```
digilent_nexys4ddr.py --build --cpu-type rocket --cpu-variant linux
--sys-clk-freq 50e6 --with-ethernet --with-sdcard
```

Cada chave no comando utilizado para geração do SoC possui um significado e é indispensável para o objetivo deste trabalho. A tabela abaixo sumariza cada uma das chaves.

Quadro 1 – Comando de construção do SoC.

`-build`	Constrói o binário para ser utilizado na FPGA
`-cpu-type rocket`	Indica qual o processador será utilizado, neste trabalho é utilizado o processador Rocket Chip
`-cpu-variant linux`	Indica qual a variante do processador será utilizado, neste trabalho é utilizado a variante linux
`-sys-clk-freq 50e6`	Indica qual a frequência será utilizada no processador (em Hz)
`-with-ethernet`	Indica que o periférico placa de rede Ethernet deve estar disponível no SoC
`-with-sdcard`	Indica que o SoC deve ter suporte a cartão de SD (caso seja necessário utilizar mais memória do que a disponível na placa)

Este comando, quando utilizado dentro do ambiente da plataforma, gera os códigos em Verilog do processador, da placa Ethernet e da comunicação serial. Devido ao uso da chave (*flag*) `-build`, a descrição em Verilog será sintetizada em um binário (*bitstream* capaz de configurara FPGA. Este binário será descartado pois ainda não implementamos nossas modificações referentes a este trabalho, em especial as modificações necessárias para os testes com a placa de rede Ethernet.

O esquema em diagrama de blocos do SoC pode ser visto na seguinte Figura 4, na qual é possível observar como é realizada a comunicação do SoC com o mundo externo através do bloco de UART/Serial, e como o processador Rocket interage com os dispositivos de memória e com os periféricos como a placa Ethernet:

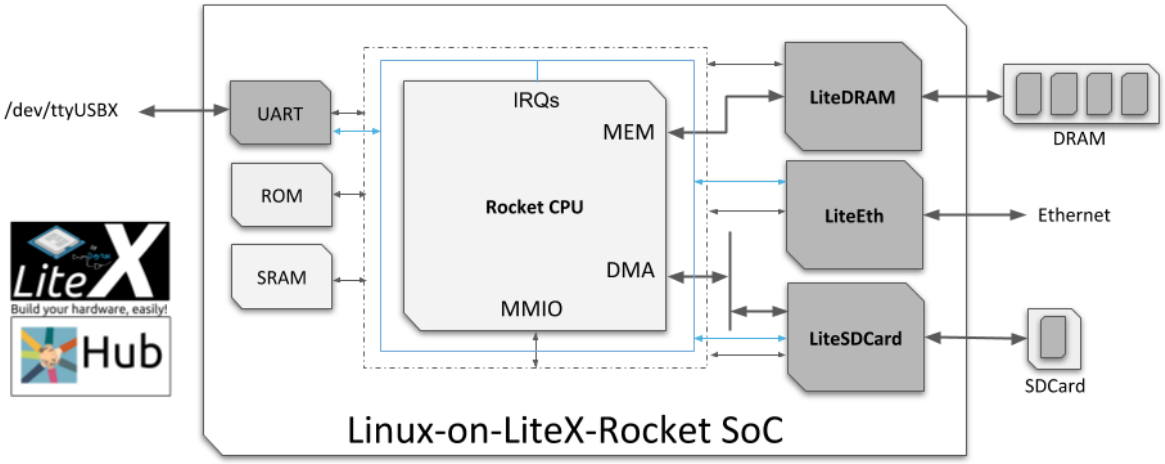

Figura 4 – Esquema em blocos do SoC. (LINUX. . . , 2023)

5.2.4 Inicialização do Kernel Linux no SoC

Com o hardware construído e juntamente com todos as partes do software: sistema operacional, *bootloader* e sistema de arquivos `initramfs`, a inicialização do sistema Kernel Linux pode ser executada. Para isso, é necessário carregar os arquivos criados anteriormente (`Image`, `rootfs.cpio` e `fw_jump.bin`) em uma região específica da memória do SoC, descrita por um arquivo JSON (*JavaScript Object Notation*), contendo o nome do arquivo e o endereço na memória onde deve ser carregado. A notação do arquivo pode ser vista na Figura 5.

```
1 {
2         "rootfs.cpio":   "0x82000000",
3         "Image":        "0x80200000",
4         "fw_jump.bin": "0x80000000"
5 }
```

Figura 5 – Arquivo JSON para inicialização do sistema

Para realizar a carga dos arquivos, deve-se atentar a porta serial no qual a FPGA está conectada (já configurada com o processador e periféricos gerados) e utilizar o seguinte comando em um terminal do Ubuntu junto com o caminho onde se encontram os arquivos `boot.json`, `Image`, `fw_jump.bin` e `rootfs.cpio`:

```
litex_term /dev/ttyUSB1 --images=/PATH/TO/boot.json
```

O sistema operacional será executado e o resultado é como na Figura 37.

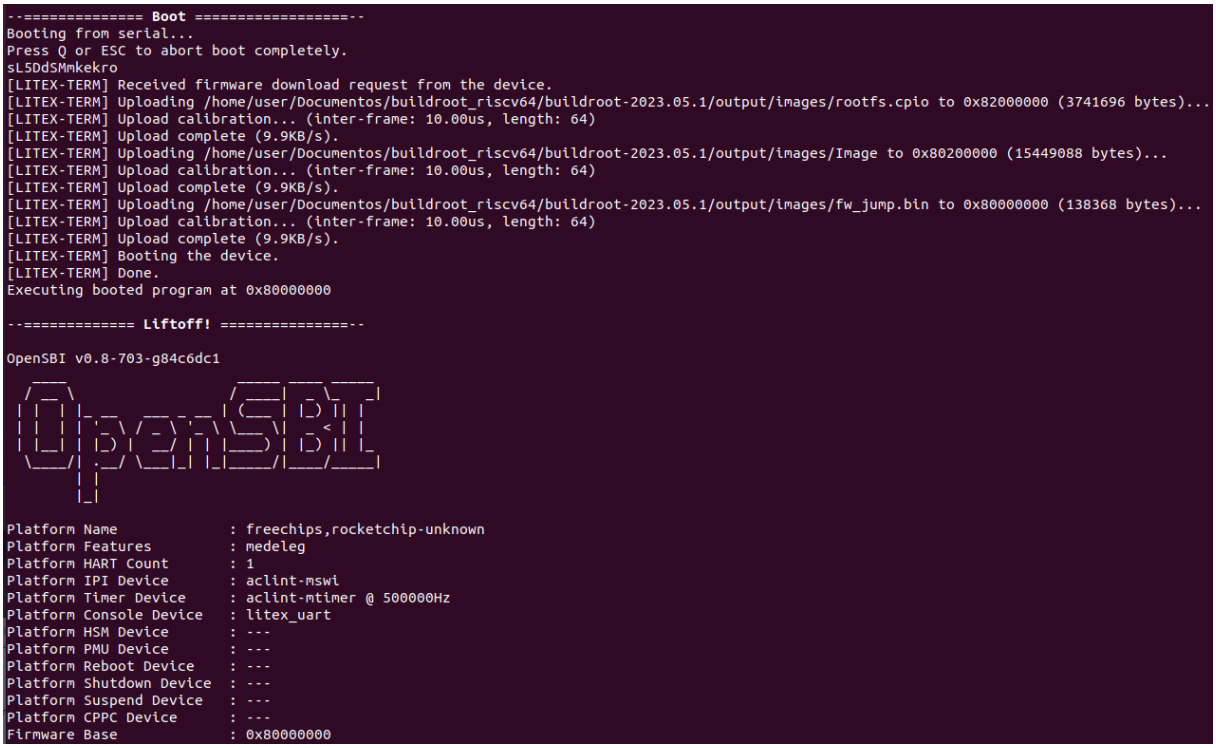

Figura 6 – Sistema Operacional em execução

5.2.5 LibSPDM

A LibSPDM é uma biblioteca de código aberto que fornece uma API, construída com base nas especificações do protocolo SPDM (DTMF, 2022), que entrega meios de se construir *requesters* e *responders*, isto é, componentes que podem se utilizar do protocolo SPDM para serem capazes de se comunicar entre si com a devida autenticação. Neste trabalho, será utilizada a LibSPDM para que a BIOS do SoC, construída na seção anterior, possa se comunicar com os periféricos e componente de software.

O primeiro caso de SPDM construído será referente a BIOS do SoC. A LibSPDM deve ser compilada em código objeto para então ser adicionada juntamente com o código objeto da própria BIOS e então as funções da biblioteca da LibSPDM podem ser utilizadas pela BIOS. Este processo inicial da compilação da LibSPDM em código objeto é realizado somente uma vez, já que estes mesmos códigos podem ser utilizados pelo *driver* da placa de rede Ethernet. Usamos a versão 2.3 da biblioteca LibSPDM, o que corresponde a definição 1.2.1 do protocolo SPDM, diretamente do repositório Git. Segue as linhas de comando utilizadas:

```
git clone https://github.com/DMTF/libspdm.git
cd libspdm
git switch release-2.3
git submodule update --init --recursive
```

Após a obtenção da biblioteca e dos módulos necessários, é necessário configurar a mesma antes de realizar qualquer tipo de compilação. Essencialmente deve-se modificar dois pontos da biblioteca: o primeiro são as configurações da `mbedtls`, uma biblioteca está incluída na `libspdm` por fornecer uma implementação do protocolo TLS, e o segundo diretamente em uma função da `mbedtls` para evitar um conflito de dupla declaração de função.

Primeiramente, as configurações da `mbedtls` devem ser aplicadas. Para isto deve-se modificar o arquivo em `libspdm/os_stub/mbedtlslib/include/mbedtls/config.h` (disponibilizado em Git de forma aberta). Este arquivo rege todas as configurações da biblioteca `mbedtls`, desde o uso de memória, declarações de funções e até mesmo algoritmos que podem ser utilizados na cifração. As configurações utilizadas neste trabalho podem ser vistas nas seguintes tabelas, onde estão descritas não apenas as configurações que foram alteradas do estado padrão (sejam adicionadas ou retiradas), como também uma breve descrição de cada característica usada:

Quadro 2 – Configurações e Funções da LibSPDM - 1.

Configuração retirada	Função
#define MBEDTLS_HAVE_ASM	O compilador tem suporte a asm()
`define MBEDTLS_HAVE_TIME`	Sistema possui a função time() e time.h
`define MBEDTLS_HAVE_TIME_DATE`	Sistema possui a função time(), time.h e suporte a função mbedtls_platform_gmtime_r(). Foi retirada devido a necessidade de tempo.
`define MBEDTLS_FS_IO`	Ativa funções que usam o sistema de arquivo.
define MBEDTLS_SELF_TEST	Ativa as funções de check.
define MBEDTLS_SSL_RECORD-_CHECKING	Ativa a função mbedtls_ssl_check_record() para checar a autenticidade de um registro.
`define MBEDTLS_SSL_CONTEXT-_SERIALIZATION`	Ativa a serialização das funções do TLS, o que aumenta o consumo de RAM.
`define MBEDTLS_SSL_KEEP_PEER-_CERTIFICATE`	Ativa a função de mbedtls_ssl_get_peer_cert() o que permite a avaliação dos certificados, porém, há um gasto considerável de RAM.
`define MBEDTLS_AESNI_C`	Ativa suporte para instruções AES-NI em x86-64.
`define MBEDTLS_NET_C`	Ativa o TCP/UDP em conexões Ipv4 e Ipv6.
`define MBEDTLS_PADLOCK_C`	Ativa suporte para VIA Padlock no x86.
`define MBEDTLS_PSA_CRYPTO_C`	Ativa a API da "Platform Security Architecture cryptography".
`define MBEDTLS_PSA_CRYPTO-_STORAGE_C`	Ativa o suporte ao elemento de segurança da "Platform Security Architecture cryptography".

Configuração da LIBSPDM - 1

Quadro 3 – Configurações e Funções da LibSPDM - 2.

Configuração retirada	Função
`define MBEDTLS_PSA_ITS_FILE_C`	Ativa a guarda de chaves persistentes na "Platform Security Architecture cryptography".
`define MBEDTLS_TIMING_C`	Ativa a interface de tempo.
`define MBEDTLS_PLATFORM_VSNPRINTF_ALT`	Ativa para deixar o mbedtls dar suporte a função na camada de abstração.
`define MBEDTLS_PLATFORM_NV_SEED_ALT`	Permite a alteração da implementação de uma função de base.
`define MBEDTLS_PLATFORM_SETUP_TEARDOWN_ALT`	Permite a alteração da implementação de uma função de base.
`define MBEDTLS_TEST_NULL_ENTROPY`	Permite o uso da mbed tls sem qualquer tipo de fonte de entropia.
`define MBEDTLS_SSL_HW_RECORD_ACCEL`	Permite a biblioteca mudar o fluxo de execução para aceleração do hardware.
`define MBEDTLS_MEMORY_BUFFER_ALLOC_C`	Ativa a alocação de memória para o buffer.
`define MBEDTLS_PLATFORM_ZEROIZE_ALT`	Permite o uso da implementação alternativa da função mbedtls_platform_zeroize().

Configuração da LIBSPDM - 2

Quadro 4 – Configurações e Funções da LibSPDM - 3.

Configuração adicionada	Função
`define MBEDTLS_NO_UDBL_DIVISION`	Sem suporte para divisões inteiras com o dobro do tamanho (64 bits em uma plataforma de 32 bits).
`define MBEDTLS_PLATFORM_MEMORY`	Permite alocação de memória.
`define MBEDTLS_PLATFORM_NO-_STD_FUNCTIONS`	Desativa o uso de funções padrão na camada de plataforma.
`define MBEDTLS_PLATFORM-_FPRINTF_ALT`	Permite o uso de implementação alternativa de uma função.
`define MBEDTLS_PLATFORM_PRINTF_ALT`	Permite o uso de implementação alternativa de uma função.
`define MBEDTLS_PLATFORM-_SNPRINTF_ALT`	Permite o uso de implementação alternativa de uma função.
`define MBEDTLS_NO_PLATFORM_ENTROPY`	Desativa o uso de funções de entropia integradas.
`define MBEDTLS_TLS_DEFAULT-_ALLOW_SHA1_IN_KEY _EXCHANGE`	Permite a utilização da função de hash SHA1 no handshake do TLS 1.2.

Configuração da LIBSPDM - 3

Quadro 5 – Configurações e Funções da LibSPDM - 4.

Configuração adicionada	Função
`define MBEDTLS_CIPHER_MODE_CBC`	Ativa o modo CBC (Cipher Block Chaining) para cifras simétricas.
`define MBEDTLS_CIPHER_MODE_CFB`	Ativa o modo CFB (Cipher Feedback Chaining) para cifras simétricas.
`define MBEDTLS_CIPHER_MODE_CTR`	Ativa o modo CTR (Counter Block Cipher) para cifras simétricas.
`define MBEDTLS_CIPHER_MODE_OFB`	Ativa o modo OFB (Output Feedback mode) para cifras simétricas.
`define MBEDTLS_CIPHER_MODE_XTS`	Ativa o modo "Xor-encrypt-Xor com XTS (ciphertext stealing mode) para o AES.
`define MBEDTLS_CIPHER-_PADDING_PKCS7`	Adiciona suporte para modos específicos de padding na camada de cifração.
`define MBEDTLS_CIPHER-_PADDING_ONE_AND_ZEROS`	Adiciona suporte para modos específicos de padding na camada de cifração.
`define MBEDTLS_CIPHER-_PADDING_ZEROS_AND_LEN`	Adiciona suporte para modos específicos de padding na camada de cifração.
`define MBEDTLS_CIPHER-_PADDING_ZEROS`	Adiciona suporte para modos específicos de padding na camada de cifração.
`define MBEDTLS_REMOVE_ARC4-_CIPHERSUITES`	Remove o uso do algoritmo RC4 por padrão no TLS/SSL.
`define MBEDTLS_REMOVE_3DES-_CIPHERSUITES`	Remove o uso do algoritmo 3DES por padrão no TLS/SSL.

Configuração da LIBSPDM - 4

Quadro 6 – Configurações e Funções da LibSPDM - 5.

Configuração adicionada	Função
`define MBEDTLS_ECDSA_DETERMINISTIC`	Ativa ECDSA (RFC 6979) determinístico.
`define MBEDTLS_ECP_DP_SECP192R1_ENABLED`	Ativa curvas elípticas específicas com o módulo de curvas elípticas.
`define MBEDTLS_ECP_DP_SECP224R1_ENABLED`	Ativa curvas elípticas específicas com o módulo de curvas elípticas.
`define MBEDTLS_ECP_DP_SECP192K1_ENABLED`	Ativa curvas elípticas específicas com o módulo de curvas elípticas.
`define MBEDTLS_ECP_DP_SECP224K1_ENABLED`	Ativa curvas elípticas específicas com o módulo de curvas elípticas.
`define MBEDTLS_ECP_DP_SECP256K1_ENABLED`	Ativa curvas elípticas específicas com o módulo de curvas elípticas.
`define MBEDTLS_ECP_DP_BP256R1_ENABLED`	Ativa curvas elípticas específicas com o módulo de curvas elípticas.
`define MBEDTLS_ECP_DP_BP384R1_ENABLED`	Ativa curvas elípticas específicas com o módulo de curvas elípticas.
`define MBEDTLS_ECP_DP_BP512R1_ENABLED`	Ativa curvas elípticas específicas com o módulo de curvas elípticas.

Configuração da LIBSPDM - 5

Quadro 7 – Configurações e Funções da LibSPDM - 6.

Configuração adicionada	Função
`define MBEDTLS_KEY_EXCHANGE_PSK_ENABLED`	Ativa o ciphersuite mode baseado em PSK no TLS/SSL.
`define MBEDTLS_KEY_EXCHANGE_DHE_PSK_ENABLED`	Ativa o ciphersuite mode baseado em DHE-PSK no TLS/SSL.
`define MBEDTLS_KEY_EXCHANGE_ECDHE_PSK_ENABLED`	Ativa o ciphersuite mode baseado em ECDHE-PSK no TLS/SSL.
`define MBEDTLS_KEY_EXCHANGE_RSA_PSK_ENABLED`	Ativa o ciphersuite mode baseado em RSA-PSK no TLS/SSL.
`define MBEDTLS_KEY_EXCHANGE_RSA_ENABLED`	Ativa o ciphersuite mode baseado em RSA no TLS/SSL.
`define MBEDTLS_KEY_EXCHANGE_DHE_RSA_ENABLED`	Ativa o ciphersuite mode baseado em DHE-RSA no TLS/SSL.
`define MBEDTLS_KEY_EXCHANGE_ECDHE_RSA_ENABLED`	Ativa o ciphersuite mode baseado em PSK no TLS/SSL.
`define MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA_ENABLED`	Ativa o ciphersuite mode baseado em PSK no TLS/SSL.
`define MBEDTLS_KEY_EXCHANGE_ECDH_ECDSA_ENABLED`	Ativa o ciphersuite mode baseado em PSK no TLS/SSL.
`define MBEDTLS_KEY_EXCHANGE_ECDH_RSA_ENABLED`	Ativa o ciphersuite mode baseado em PSK no TLS/SSL.

Configuração da LIBSPDM - 6

Quadro 8 – Configurações e Funções da LibSPDM - 7.

Configuração adicionada	Função
`define MBEDTLS_PK_PARSE_EC_EXTENDED`	Melhorar o suporte para leitura de chaves EC usando variantes de SEC1 não permitidas pelo RFC 5915 e RFC 5480.
`define MBEDTLS_PK_RSA_ALT_SUPPORT`	Ativa o suporte para chaves privadas RSA externas na camada PK.
`define MBEDTLS_SSL_ALL_ALERT_MESSAGES`	Ativa o envio de mensagens de alerta em caso de erro encontrado como por RFC.
`define MBEDTLS_SSL_ENCRYPT_THEN_MAC`	Ativa suporte para "Encrypt-then-MAC"para o RFC7366.
`define MBEDTLS_SSL_EXTENDED_MASTER_SECRET`	Ativa suporte para o RFC7627, hash de sessão e segredo mestre extendido.
`define MBEDTLS_SSL_FALLBACK_SCSV`	Ativa suporte para RFC 7507: Fallback Signaling Cipher Suite Value (SCSV), para prevenir ataques de downgrade de protocolo.
`define MBEDTLS_SSL_CBC_RECORD_SPLITTING`	Ativa 1/n-1 "record splitting"para o modo CBC no SSLv3 and TLS 1.0.
`define MBEDTLS_SSL_RENEGOTIATION`	Ativa suporte para renegociação no TLS.
`define MBEDTLS_SSL_MAX_FRAGMENT_LENGTH`	Ativa o suporte para a extensão RFC 6066 "max$_{f}ragment_{l}ength$"$noSSL$.
`define MBEDTLS_SSL_PROTO_TLS1`	Ativa o suporte para o TLS 1.0.
`define MBEDTLS_SSL_PROTO_TLS1_1`	Ativa o suporte para o TLS 1.1 e DTLS 1.0 se o DTLS estiver ativado.
`define MBEDTLS_SSL_PROTO_TLS1_2`	Ativa o suporte para o TLS 1.2 e DTLS 1.2 se o DTLS estiver ativado.

Configuração da LIBSPDM - 7

Quadro 9 – Configurações e Funções da LibSPDM - 8.

Configuração adicionada	Função
`define MBEDTLS_SSL_PROTO_DTLS`	Ativa suporte para o DTLS (todas as versões disponíveis).
`define MBEDTLS_SSL_ALPN`	Ativa o suporte para o RFC 7301 na "Application Layer Protocol Negotiation".
`define MBEDTLS_SSL_DTLS_ANTI-_REPLAY`	Ativa o suporte para o mecanismo anti replay no DTLS.
`define MBEDTLS_SSL_DTLS-_HELLO_VERIFY`	Ativa o suporte para o "HelloVerifyRequest"nos servidores DTLS.
`define MBEDTLS_SSL_DTLS-_CLIENT_PORT_REUSE`	Ativa suporte no lado do servidor para requisições que reconectam na mesma porta.
`define MBEDTLS_SSL_DTLS-_BADMAC_LIMIT`	Ativa suporte para um limite de registros de MAC ruins.
`define MBEDTLS_SSL_SESSION_TICKETS`	Ativa suporte para tickets de sessão em RFC 5077.
`define MBEDTLS_SSL_EXPORT_KEYS`	Ativa suporte para exportação de chave de bloco e segredo mestre.
`define MBEDTLS_SSL_SERVER-_NAME_INDICATION`	Ativa o suporte para indicação de nome de servidor (SNI) RFC 6066 em SSL.
`define MBEDTLS_SSL_TRUNCATED_HMAC`	Ativa o suporte para HMAC truncado RFC 6066 em SSL.
`define MBEDTLS_VERSION_FEATURES`	Ativa verificação em tempo de execução de funções do tipo de tempo de compilação.
`define MBEDTLS_ARC4_C`	Ativa a cifra de fluxo ARC4.
`define MBEDTLS_BLOWFISH_C`	Ativa a cifra de blocos Blowfish.
`define MBEDTLS_CAMELLIA_C`	Ativa a cifra de blocos Camellia.

Configuração da LIBSPDM - 8

Quadro 10 – Configurações e Funções da LibSPDM - 9.

Configuração adicionada	Função
`define MBEDTLS_CCM_C`	Ativa o contador com o modo CBC-MAX para blocos de cifração de 128 bits.
`define MBEDTLS_CERTS_C`	Ativa os certificados de teste.
`define MBEDTLS_DEBUG_C`	Ativa as funções de debug.
`define MBEDTLS_DES_C`	Ativa o block de cifração DES.
`define MBEDTLS_MD5_C`	Ativa o algoritmo de hash MD5.
`define MBEDTLS_PKCS5_C`	Ativa as funções de PKCS5.
`define MBEDTLS_PKCS12_C`	Ativa as funções de PKCS12.
`define MBEDTLS_RIPEMD160_C`	Ativa o algoritmo de hash RIPEMD160.
`define MBEDTLS_SHA1_C`	Ativa o uso do algoritmo da função de hash SHA1.
`define MBEDTLS_SSL_CACHE_C`	Ativa uma implementação simples do cache do SSL.
`define MBEDTLS_SSL_COOKIE_C`	Ativa a implementação básicas dos cookies do DTLS para verificação de hello.
`define MBEDTLS_SSL_TICKET_C`	Ativa uma implementação para callbacks para os tickes de sessão no lado do servidor no TLS.
`define MBEDTLS_SSL_CLI_C`	Ativa o código de cliente no TLS/SSL.
`define MBEDTLS_SSL_SRV_C`	Ativa o código de servidor do TLS/SSL.
`define MBEDTLS_SSL_TLS_C`	Ativa suporte para o mecanismo de anti replay no DTLS.
`define MBEDTLS_VERSION_C`	Ativa a informação de versão em tempo de execução
`define MBEDTLS_XTEA_C`	Ativa o XTEA no bloco de cifração.
`define MBEDTLS_TLS-_DEFAULT_ALLOW_SHA1-_IN_KEY_EXCHANGE`	Permite o uso da função de hash SHA1 na assinatura de handshake no TLS 1.2.

Configuração da LIBSPDM - 9

A segunda configuração que se deve atender é a modificação de uma função da `crypto_mbedtls`, pois a função em questão já possui uma implementação que pode ser utilizada e esta já foi definida pelas configurações descritas nas tabelas anteriores. Basta retirá-la de `os_stub/cryptlib_mbedtls/sys_call/crt_wrapper_host.c`, ou seja, comentar toda a função `mbedtls_platform_zeroize` para que a sua implementação alternativa seja utilizada.

Com as duas modificações, tanto no `config.h` da `mbedtls`, como na função anteriormente descrita, utiliza-se os seguintes comandos, dentro do diretório que contém a `libspdm`, para obter o código objeto da biblioteca na arquitetura RISC-V que será utilizado posteriormente para integração dentro da BIOS e do *driver* da placa de rede Ethernet:

```
$ mkdir build
$ cd build
$ export PATH=$PATH:/opt/riscv/bin
$ cmake -DARCH=riscv64 -DTOOLCHAIN=RISCV_GNU -DTARGET=Release
    -DCRYPTO=mbedtls
$ make copy_sample_key
$ make
```

5.3 Testes e Avaliação

5.3.1 Testes da Ethernet

Os testes da placa de rede são testes simples relacionados ao funcionamento básico da placa de forma genérica. A placa de rede gerada pelo *framework* LiteX foi testada até o envio e recebimento do primeiro pacote de rede. Não há um pacote específico pois não estamos testando a pilha de rede e sim a funcionalidade do hardware. Nesse sentido, optamos por um simples pacote "ping" enviado para uma rede local. O cabo ethernet foi ligado a um computador e à placa FPGA com o processador executando um Kernel Linux, quando conseguimos enviar um pacote do FPGA para o computador, que respondeu adequadamente, fechando o ciclo de teste.

Ao se iniciar o sistema operacional contido na FPGA, todas as interfaces de comunicação de rede estão desconfiguradas, incluindo a placa Ethernet. Logo, a mesma não possui nenhum tipo de informação configurada, o que inclui não possuir endereço para que o comando "ping" envie os pacotes. Sendo assim, o primeiro passo é inicializar a placa de rede atribuindo-lhe um endereço IP. Para isso, usamos o comando (disponível em qualquer Kernel Linux) `ifconfig`, responsável por atribuir o endereço IP. Portanto, os seguintes comandos foram executados para entregar a placa de rede um endereço IP que possa ser utilizado:

- ifconfig -a
- ifconfig eth0 up
- ifconfig eth0 169.254.24.189

O primeiro comando da lista, `ifconfig -a`, é executado devido a dois motivos em mente: o primeiro para conseguir a informação de qual é o rótulo da interface da Ethernet, pois a placa FPGA possuirmais de uma interface para conexão de rede, e o segundo para descobrirmos qual a máscara de rede que a interface possui para posteriormente ser definido um IP. O segundo comando `ifconfig eth0 up` é executado logo após a informação de qual interface está sendo lidada, e garante que a interface da Ethernet (neste caso, a interface da Ethernet é chamada de `eth0`, porém, poderia ter outro nome a depender do caso) está ativada e pronta para ser configurada. Por fim, o terceiro comando `ifconfig eth0 169.254.24.189` atribui um endereço IP para a placa de rede Ethernet, que deve seguir a máscara de rede que já vem definida.

Em relação a máquina hospedeira que contém um sistema operacional Ubuntu 20.02, isto é, aquela conectada via cabeamento com a FPGA, não há configurações adicionais, basta visualizar o IP que está atribuído a interface de Ethernet. Nos testes realizados

é possível observar o endereço `169.254.24.188` atribuído a interface Ethernet, e este endereço é aquele que será utilizado para realizar o `ping` entre a máquina hospedeira e a placa FPGA. Na Figura 7 é possível ver destacado o endereço Ethernet da máquina hospedeira obtido pelo comando `ip addr show`. Este IP será utilizado para o "ping". Além disso, todas as demais conexões de rede foram desconectadas para evitar qualquer tipo de interferência nos testes.

```
(base) user@user-H110-D3A:~$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether e0:d5:5e:71:a3:37 brd ff:ff:ff:ff:ff:ff
    inet 169.254.24.188/16 brd 169.254.255.255 scope link noprefixroute enp2s0
       valid_lft forever preferred_lft forever
    inet6 fe80::40c8:cb72:e2b3:2ec9/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: wlxe46f13a6c47c: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN group default qlen 1000
    link/ether e4:6f:13:a6:c4:7c brd ff:ff:ff:ff:ff:ff
(base) user@user-H110-D3A:~$
```

Figura 7 – Endereço da interface Ethernet na máquina hospedeira

Dois tipos de testes são realizados para garantir o pleno funcionamento da Ethernet. Primeiramente foi executado um comando `ping` da placa FPGA para a máquina hospedeira e posteriormente o inverso foi aplicado. A execução de ambos os testes foram bem sucedidos. Após a configuração da interface Ethernet com os comandos descritos anteriormente, foi executado um comando `ping` da FPGA para o endereço máquina hospedeira (`169.254.24.188`) o que resultou no envio correto dos pacotes de dados para o destinatário com a resposta de `169.254.24.188 is alive!`. Logo em seguida foi realizado um outro comando `ping` para uma máquina que não existe, ou seja, com endereço não existente, para garantir de que a FPGA não encontrou a máquina hospedeira apenas devido ao cabeamento Ethernet. Isto resultou no esperado `no response from 169.254.24.187`, o que quer dizer que só havia a possibilidade de se comunicar com o endereço da máquina hospedeira, como se espera de um cabeamento de duas pontas.

A configuração da placa de rede da FPGA, através dos comandos anteriormente descritos e explicados, juntamente com a execução do primeiro teste, isto é, com a utilização do comando "ping" para a máquina hospedeira e para uma máquina inexistente podem ser visualizados na Figura 8:

O segundo teste consistiu no caminho oposto, foi realizado executando um comando de "ping" da máquina hospedeira para a FPGA. O resultado foi o esperado de que os pacotes foram chegando na FPGA com um certo tempo. Foram realizados cinco envios de pacotes, no qual também era possível perceber o aumento do tempo.

```
# ping
# ping 169.254.24.188
ping: sendto: Network is unreachable
# ifconfig eth0 up
# ifconfig eth0 169.254.24.189
# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 8E:32:F4:9C:BB:E0
          inet addr:169.254.24.189  Bcast:169.254.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:9 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:540 (540.0 B)  TX bytes:726 (726.0 B)
          Interrupt:1

lo        Link encap:Local Loopback
          LOOPBACK  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

sit0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-01-00-00-00-00-00-00-00-00-00
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

# ping 169.254.24.188
169.254.24.188 is alive!
# ping 169.254.24.187
No response from 169.254.24.187
#
```

Figura 8 – Configuração da placa Ethernet na FPGA seguido de testes com o comando de ping para máquina hospedeira

```
(base) user@user-H110-D3A:~$ ping 169.254.24.189
PING 169.254.24.189 (169.254.24.189) 56(84) bytes of data.
64 bytes de 169.254.24.189: icmp_seq=1 ttl=64 tempo=2.91 ms
64 bytes de 169.254.24.189: icmp_seq=2 ttl=64 tempo=1.52 ms
64 bytes de 169.254.24.189: icmp_seq=3 ttl=64 tempo=1.47 ms
64 bytes de 169.254.24.189: icmp_seq=4 ttl=64 tempo=1.51 ms
64 bytes de 169.254.24.189: icmp_seq=5 ttl=64 tempo=1.49 ms
```

Figura 9 – Comando `ping` realizado da máquina hospedeira para a FPGA

5.3.2 Testes do registrador de controle do SPDM através da BIOS

Dentro do protocolo do SPDM existem certos bits que devem ser guardados em um registrador de controle ao se utilizar o protocolo. Por exemplo, ao se solicitar o início de uma transferência de um arquivo com o protocolo, o periférico que realiza a requisição deve avisar qual o tipo de algoritmo criptográfico que está sendo utilizado, portanto existem

dados binários que devem ser armazenados em registradores de controle.

Inicialmente, para se ter certeza de que a criação de um novo registrador de controle pode ser controlado através da BIOS, foi proposto um simples teste, com a criação de um novo registrador CSR (*Control and Status Register*). De modo reflexivo foi construída uma nova função dentro do BIOS capaz de controlar o conteúdo deste registrador, que deve ser mostrado na saída serial e ao mesmo tempo ser visualmente identificável através dos LEDs disponíveis na FPGA.

Para realizar este teste, realizamos uma mudança na infraestrutura lógica do SoC na FPGA, ou seja, o registrador de controle do SPDM foi colocado dentro da placa de rede Ethernet. É objetivo deste trabalho implementar o SPDM dentro da placa de rede, assim este registrador foi adicionado utilizando a própria infraestrutura do *LiteX*, ou seja, alterando o código fonte da placa de rede com a adição de um simples registrador de controle. O registrador foi descrito em Python através da biblioteca Migen, utilizada pela LiteX para gerar os periféricos, e foi adicionado dentro da placa de rede Ethernet, mais especificamente dentro da sua parte física, refletindo na descrição Verilog gerada pelo *framework*.

```
class LiteEthSPDMRX(LiteXModule):
  def __init__(self):
        self._register_SPDM = CSRStorage(32, name = "SPDM_register", description = "SPDM_Control")
        self.source = source = stream.Endpoint(eth_phy_description(8))

        rx_data = Signal(2)

        converter= stream.Converter(2,8)
        converter = ResetInserter()(converter)
        self.converter = converter
        self.comb += converter.source.connect(source)
```

Figura 10 – Construção do registrador dentro da placa de rede Ethernet

Com a criação do registrador específico para o SPDM, antes de realizar modificações na BIOS é necessário saber onde o registrador foi mapeado dentro da memória. As funções de controle descritas dentro da BIOS atuam se aproveitando de funções de leitura e escrita geradas pela LiteX que já possuem um mapeamento especifico no espaço de endereçamento da memória física. Isto se aplica a todos registradores de controle dentro do SoC, que podem ser visualizados dentro do arquivo `csr.h`, usado ao se gerar o SoC. Como o registrador foi incluído dentro da parte física da placa de rede Ethernet, é esperado que esteja junto com os demais registradores da placa. A Figura 11 a seguir mostra o mapeamento realizado pela LiteX para o registrador de controle do SPDM. Nota-se que os mesmos estão dentro da parte física da placa de rede Ethernet e possuem um endereço juntamente com duas funções, uma para se escrever neste registrador e uma para se ler (marcadas na Figura 11 no quadro preto).

Antes de realizar a criação da função de controle dentro da BIOS, será necessário implementar o modo como será visualizado a mudança do conteúdo do registrador. Como anteriormente descrito, a visualização será feita primeiramente pela saída serial e posteriormente pelos LEDs da FPGA. Sendo assim, a descrição em Verilog do SoC será modificada para atender a segunda parte. Os primeiros 16 bits do sinal de saída do registrador de controle do SPDM foram ser ligados aos LEDs, pois estes já possuem um mapeamento nos pinos da FPGA feito pela *framework* LiteX. Desta forma, basta alterar a saída do registrador, ou seja, não há qualquer tipo de mudança no comportamento deste registrador, já que o que foi feito é apenas um desvio de uma cópia do sinal de saída para os LEDs. A Figura 12 a seguir mostra a descrição em RTL (*Register Transfer Level*) do registrador do SPDM após as mudanças no Verilog. A alteração foi feita dentro do ambiente de prototipação do Vivado, terminando com o registrador ligado diretamente a saída dos LEDs da FPGA. Os sinais estão destacados em azul.

Por fim, antes de realizar os testes para manipulação deste registrador, a função para realizar tal feito deve ser criada dentro da BIOS gerada pela LiteX. Para isso, modificamos os arquivos contendo os comandos do usuário que podem ser utilizados dentro da BIOS. O arquivo modificado foi o `cmd_bios.c`. Neste arquivo criamos a função `spdm_handler`, que recebe um valor qualquer escrito como um inteiro, realiza a verificação para observar se

```
391 /* ethphy */
392 #define CSR_ETHPHY_BASE (CSR_BASE + 0x1800L)
393 #define CSR_ETHPHY_CRG_RESET_ADDR (CSR_BASE + 0x1800L)
394 #define CSR_ETHPHY_CRG_RESET_SIZE 1
395 static inline uint32_t ethphy_crg_reset_read(void) {
396         return csr_read_simple((CSR_BASE + 0x1800L));
397 }
398 static inline void ethphy_crg_reset_write(uint32_t v) {
399         csr_write_simple(v, (CSR_BASE + 0x1800L));
400
401 #define CSR_ETHPHY_RX__REGISTER_SPDM_SPDM_REGISTER_ADDR (CSR_BASE + 0x1804L)
402 #define CSR_ETHPHY_RX__REGISTER_SPDM_SPDM_REGISTER_SIZE 1
403 static inline uint32_t ethphy_rx__register_spdm_SPDM_register_read(void) {
404         return csr_read_simple((CSR_BASE + 0x1804L));
405 }
406 static inline void ethphy_rx__register_spdm_SPDM_register_write(uint32_t v) {
407         csr_write_simple(v, (CSR_BASE + 0x1804L));
408 }
409 #define CSR_ETHPHY_MDIO_W_ADDR (CSR_BASE + 0x1808L)
410 #define CSR_ETHPHY_MDIO_W_SIZE 1
411 static inline uint32_t ethphy_mdio_w_read(void) {
412         return csr_read_simple((CSR_BASE + 0x1808L));
413 }
414 static inline void ethphy_mdio_w_write(uint32_t v) {
415         csr_write_simple(v, (CSR_BASE + 0x1808L));
416 }
417 #define CSR_ETHPHY_MDIO_W_MDC_OFFSET 0
418 #define CSR_ETHPHY_MDIO_W_MDC_SIZE 1
419 static inline uint32_t ethphy_mdio_w_mdc_extract(uint32_t oldword) {
420         uint32_t mask = 0x1;
421         return ( (oldword >> 0) & mask );
422 }
423 static inline uint32_t ethphy_mdio_w_mdc_read(void) {
424         uint32_t word = ethphy_mdio_w_read();
425         return ethphy_mdio_w_mdc_extract(word);
426 }
```

Figura 11 – Descrição das funções de manipulação do registrador

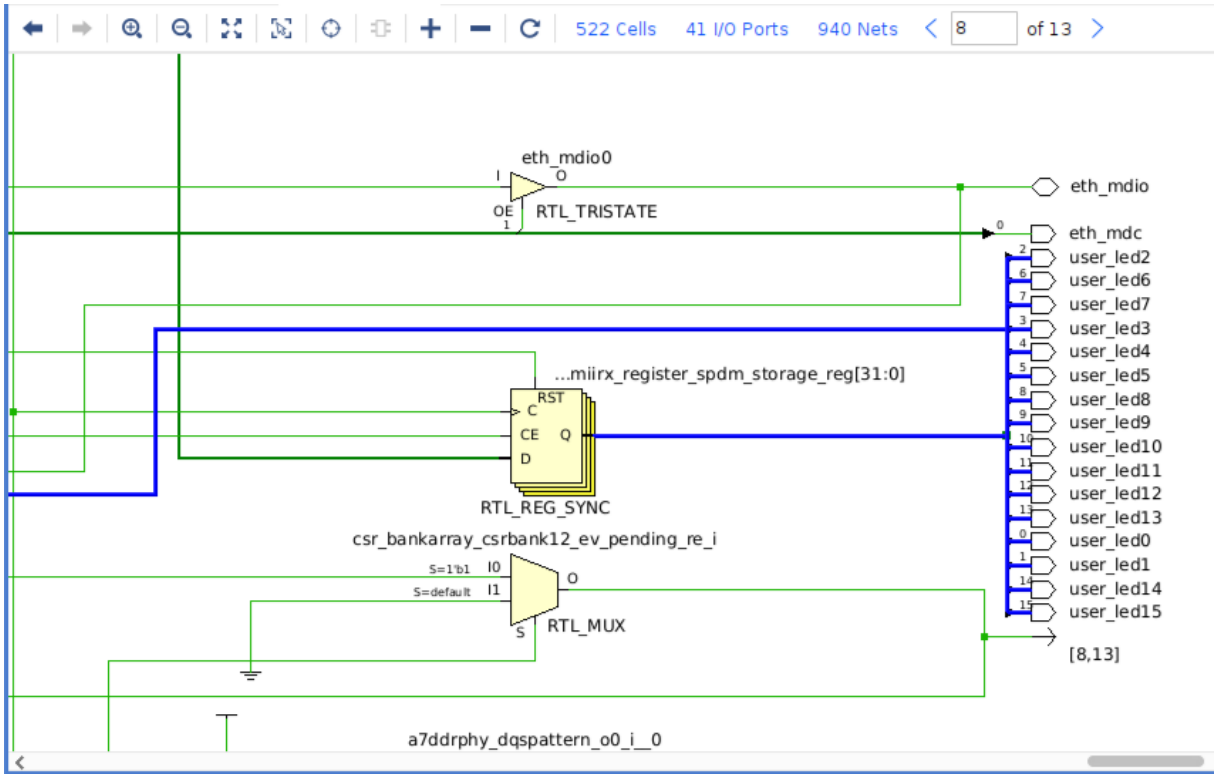

Figura 12 – Descrição em RTL do registrador de controle do SPDM ligada aos LEDs da FPGA

não foi recebido um vazio (todos bits zero). Caso tenha recebido um vazio,a função solicita outro valor, mas caso o valor seja válido o valor é escrito dentro do console na saída serial e ao final a função de escrita do registrador (Figura 13) é chamada para escrever o valor no endereço correto já definido pela LiteX. Ao fim da função é registrado a tipificação do comando pelo qual a operação é realizada, isto é, o que o usuário da BIOS deve escrever para chamar a função. Neste caso a função correspondente chama-se `spdm_csr`, e para seu uso deve-se fornecer apenas a tipificação da função junto com o valor correspondente, com a possibilidade de escrever o valor em inteiro ao invés de utilizar hexadecimal. A Figura 13 a seguir mostra a descrição da função escrita em C, sendo esta uma adaptação de uma outra função fornecida pela LiteX (também utilizada para manipular LEDs, porém, neste caso, ela foi desativada para evitar conflitos no uso dos LEDs):

```
197 #ifdef CSR_ETHPHY_BASE
198 static void spdm_handler(int nb_params, char **params)
199 {
200         char *c;
201         unsigned int value;
202
203         if (nb_params < 1) {
204                 printf("leds <value>");
205                 return;
206         }
207
208         value = strtoul(params[0], &c, 0);
209         if (*c != 0) {
210                 printf("Incorrect value");
211                 return;
212         }
213
214         printf("Settings Leds to 0x%x", value);
215         ethphy_rx__register_spdm_SPDM_register_write(value);
216 }
217
218 define_command(spdm_csr, spdm_handler, "Set SPDM value", SYSTEM_CMDS);
219 #endif
```

Figura 13 – Função para manipulação do registrador do SPDM construída dentro do *firmware*

Por fim, há a execução da função criada, a `spdm_csr`. Neste exemplo temos a escrita do valor `0x2` dentro do registrador do SPDM. Com isto, temos a prova de conceito de que os registradores criados podem ter seus valores modificados por funções externas ao hardware. Os testes sob a placa de rede Ethernet descritos na seção 5.3.1 foram repetidos para se garantir de que a mesma continua em pleno funcionamento após as modificações. As imagens a seguir mostram a utilização da função criada, sua descrição no “help” da BIOS, e finalmente a visualização na saída serial e através dos LEDs da placa:

```
--============= Console ================--

litex> help

LiteX BIOS, available commands:

leds                      - Set Leds value
flush_cpu_dcache          - Flush CPU data cache
crc                       - Compute CRC32 of a part of the address space
ident                     - Identifier of the system
help                      - Print this help
spdm_csr                  - Set SPDM value

netboot                   - Boot via Ethernet (TFTP)
serialboot                - Boot from Serial (SFL)
reboot                    - Reboot
boot                      - Boot from Memory

mem_cmp                   - Compare memory content
mem_speed                 - Test memory speed
mem_test                  - Test memory access
mem_copy                  - Copy address space
mem_write                 - Write address space
mem_read                  - Read address space
mem_list                  - List available memory regions

sdram_mr_write            - Write SDRAM Mode Register
sdram_cal                 - Calibrate SDRAM
sdram_test                - Test SDRAM
sdram_init                - Initialize SDRAM (Init + Calibration)
sdram_force_wrphase       - Force write phase
sdram_force_rdphase       - Force read phase

mdio_dump                 - Dump MDIO registers
mdio_read                 - Read MDIO register
mdio_write                - Write MDIO register

sdcard_write              - Write SDCard block
sdcard_read               - Read SDCard block
sdcard_freq               - Set SDCard clock freq
sdcard_init               - Initialize SDCard
sdcard_detect             - Detect SDCard


litex> spdm_csr
leds <value>
litex> spdm_csr 2
Settings Leds to 0x2
litex>
```

Figura 14 – Controle do Registrador SPDM pela BIOS

5.4 Desenvolvimento - Hardware com SPDM

5.4.1 Integração da LibSPDM na BIOS

Como realizado na seção de Testes e Avaliações, o SoC construído na seção de Projeto e Implementação precisa ser modificado, pois deve ser preparado para receber e

executar as funções da LibSPDM para que o protocolo possa ser utilizado na comunicação entre os diversos componentes do SoC. Como parte deste trabalho, a BIOS do SoC e o *driver* da placa de rede Ethernet serão os alvos que receberão a LibSPDM.

Em relação a BIOS, os binários da LibSPDM já compilados anteriormente precisam ser unificados, em formato de biblioteca, junto a BIOS. Neste caso, deve-se realizar duas mudanças na compilação, que se referem ao uso dos arquivos de `Makefile`. Criamos um novo `Makefile` diferente daquele padrão da LiteX, com a função de colocar os códigos fonte de `headers` dentro da biblioteca da LiteX, em formato de `link` referenciável. Isso permite que os novos comandos da BIOS consigam utilizar as funções fornecidas pela LibSPDM. O novo `Makefile` foi chamado de `libspdm_litex.mk` e pode ser visto na Figura 15 a seguir:

```
1 SPDM_DIR?= /home/user/Documentos/libspdm_docs/libspdm
2 LiteX_DIR?= /home/user/Documentos/LiteX/litex/litex/soc/software
3 SPDM_HEADERS = $(SPDM_DIR)/include
4
5 SPDM_INCLUDE_DIR =  \
6         $(SPDM_HEADERS) \
7         $(SPDM_HEADERS)/hal \
8         $(SPDM_HEADERS)/library \
9         $(SPDM_HEADERS)/industry_standard \
10        $(SPDM_DIR)/library/spdm_common_lib \
11        $(SPDM_DIR)/library/spdm_crypt_lib \
12        $(SPDM_DIR)/library/spdm_requester_lib \
13        $(SPDM_DIR)/library/spdm_responder_lib \
14        $(SPDM_DIR)/library/spdm_transport_mctp_lib \
15        $(SPDM_DIR)/library/spdm_transport_pcidoe_lib \
16        $(SPDM_DIR)/library/spdm_secured_message_lib
17
18 SPDM_HEADER_DIR_KERNEL := $(foreach incdir, $(SPDM_INCLUDE_DIR), $(addprefix libspdm/,$(notdir $(incdir))))
19
20 $(SPDM_HEADER_DIR_KERNEL):
21        cd $(LiteX_DIR) && mkdir -p libspdm/
22        cd $(LiteX_DIR) && ln -s $(filter %$(notdir $@), $(SPDM_INCLUDE_DIR)) $@
23        cd $(LiteX_DIR) && cd libspdm/ && if ls $@/*.h >/dev/null 2>&1; then for i in $@/*.h; do ln -s ../../$$i libspdm/; done; fi
```

Figura 15 – Makefile libspdm_litex.mk

Por razões de reprodutividade, deve-se ter em atenção nas duas primeiras linhas do `Makefile` (`SPDM_DIR e LiteX_DIR`), pois estas indicam o caminho no qual se encontra os arquivos da LibSPDM e da LiteX, ou seja, caso os caminhos dos arquivos estejam dispostos em um lugar diferente deve-se alterá-las com o caminho correto. Para executar este novo `Makefile` basta utilizar o seguinte comando no terminal:

```
make -f libspdm_litex.mk
```

Após a adição dos `headers` dentro das bibliotecas de software do SoC, é necessário atualizar os caminhos de localização dos `headers` e atualizar as *flags* de compilação. Isso demonstra ao compilador o caminho onde ele deve procurar os arquivos que foram adicionados. Para isso, dentro do arquivo `common.mak` que se encontra no caminho `/litex/litex/soc/software`, devem ser realizadas modificações. Além de se adicionar as variáveis de caminho onde se encontram a LibSPDM e a LiteX (`SPDM_DIR e LiteX_DIR`, nas linhas 1 e 6 da Figura 16). A primeira modificação se encontra em adicionar a variável `LIBRARIES` (linha 8 da Figura 16), que contém todas as bibliotecas da LibSPDM já compiladas, que são então adicionadas pela *flag* de compilação `-L` (linha 88 da Figura 17). Nas demais *flags* de compilação adiciona-se a *flag* de `-static`, o que instrui o compilador a adicionar as bibliotecas em tempo de compilação assim como na LibSPDM.

Por fim, se utilizando da *flag* de compilação `-I` dentro da variável `INCLUDES` (linha 58 da Figura 16) indica-se o caminho no qual se deve procurar os `headers` para evitar problemas de arquivos não encontrados. Todas as mudanças relatadas no arquivo `common.mak` podem ser visualizadas nas Figuras 16 e 17.

```
1 SPDM_DIR?= /home/user/Documentos/libspdm_docs/libspdm
2 SPDM_BUILD_DIR?= $(SPDM_DIR)/build
3 SPDM_HEADERS = $(SPDM_DIR)/include
4 SPDM_FINAL?= $(SPDM_DIR)/build/lib
5 SPDM_CRYPTO?= mbedtls
6 LiteX_DIR?= /home/user/Documentos/LiteX/litex/litex/soc/software
7
8 LIBRARIES = cmockalib rnglib memlib malloclib debuglib cryptlib_$(SPDM_CRYPTO) $(SPDM_CRYPTO) spdm_crypt_lib spdm_secured_message_lib spdm_requester_lib
  spdm_device_secret_lib_null spdm_common_lib spdm_transport_mctp_lib spdm_transport_pcidoe_lib spdm_device_secret_lib_sample spdm_responder_lib cryptlib_null
  cryptstublib_dummy debuglib_null intrinsiclib malloclib_null malloclib_simple mbedcrypto mbedx509 platform_lib platform_lib_null rnglib_null spdm_crypt_ext_lib
  spdm_transport_test_lib
9
10 ifeq ($(TRIPLE),--native--)
11 TARGET_PREFIX=
12 else
13 TARGET_PREFIX=$(TRIPLE)-
14 endif
15
16 RM ?= rm -f
17 PYTHON ?= python3
18 CCACHE ?=
19
20 ifeq ($(CLANG),1)
21 CC_normal      := $(CCACHE) clang -target $(TRIPLE) -integrated-as -static
22 CX_normal      := $(CCACHE) clang++ -target $(TRIPLE) -integrated-as -static
23 else
24 CC_normal      := $(CCACHE) $(TARGET_PREFIX)gcc -std=gnu99 -static
25 CX_normal      := $(CCACHE) $(TARGET_PREFIX)g++ -static
26 endif
27 AR_normal      := $(TARGET_PREFIX)gcc-ar
28 LD_normal      := $(TARGET_PREFIX)ld -static
29 OBJCOPY_normal := $(TARGET_PREFIX)objcopy
30
31 CC_quiet      = @echo "  CC      " $@ && $(CC_normal)
32 CX_quiet      = @echo "  CX      " $@ && $(CX_normal)
33 AR_quiet      = @echo "  AR      " $@ && $(AR_normal)
34 LD_quiet      = @echo "  LD      " $@ && $(LD_normal)
35 OBJCOPY_quiet = @echo "  OBJCOPY " $@ && $(OBJCOPY_normal)
36
37 ifeq ($(V),1)
38         CC = $(CC_normal)
39         CX = $(CX_normal)
40         AR = $(AR_normal)
41         LD = $(LD_normal)
42         OBJCOPY = $(OBJCOPY_normal)
43 else
44         CC = $(CC_quiet)
45         CX = $(CX_quiet)
46         AR = $(AR_quiet)
47         LD = $(LD_quiet)
48         OBJCOPY = $(OBJCOPY_quiet)
```

Figura 16 – Adição da biblioteca junto as flags de compilação na Common.mak

```
51 # http://scottmcpeak.com/autodepend/autodepend.html
52 # Generate *.d Makefile dependencies fragments, include using;
53 # -include $(OBJECTS:.o=.d)
54 DEPFLAGS += -MD -MP
55
56 # Toolchain options
57 #
58 INCLUDES = -I$(PICOLIBC_DIRECTORY)/newlib/libc/tinystdio \
59            -I$(PICOLIBC_DIRECTORY)/newlib/libc/include \
60            -I$(LIBBASE_DIRECTORY) \
61            -I$(SOC_DIRECTORY)/software/include \
62            -I$(SOC_DIRECTORY)/software \
63            -I$(BUILDINC_DIRECTORY) \
64            -I$(BUILDINC_DIRECTORY)/../libc \
65            -I$(CPU_DIRECTORY) \
66            -I$(SPDM_HEADERS) \
67            -I$(SPDM_HEADERS)/hal \
68            -I$(SPDM_HEADERS)/library \
69            -I$(SPDM_HEADERS)/industry_standard \
70            -I$(SPDM_HEADERS)/internal \
71            -I$(SPDM_DIR)/library/spdm_common_lib \
72            -I$(SPDM_DIR)/library/spdm_crypt_lib \
73            -I$(SPDM_DIR)/library/spdm_requester_lib \
74            -I$(SPDM_DIR)/library/spdm_responder_lib \
75            -I$(SPDM_DIR)/library/spdm_transport_mctp_lib \
76            -I$(SPDM_DIR)/library/spdm_transport_pcidoe_lib \
77            -I$(SPDM_DIR)/library/spdm_secured_message_lib
78
79 COMMONFLAGS = $(DEPFLAGS) -Os $(CPUFLAGS) -g3 -no-pie -fomit-frame-pointer -static -Wall -fno-builtin -fno-stack-protector $(INCLUDES)
80 ifeq ($(LTO), 1)
81 COMMONFLAGS += -flto
82 endif
83 ifneq ($(CPUFAMILY), arm)
84 COMMONFLAGS += -fexceptions
85 endif
86 CFLAGS = $(COMMONFLAGS) -Wstrict-prototypes -Wold-style-definition -Wmissing-prototypes -static
87 CXXFLAGS = $(COMMONFLAGS) -std=c++11 -I$(SOC_DIRECTORY)/software/include/basec++ -I$(SPDM_BUILD_DIR)/lib -fno-rtti -ffreestanding -static
88 LDFLAGS = -nostdlib -C -nodefaultlibs -Wl,--no-dynamic-linker -Wl,--build-id=none $(CFLAGS) -L$(BUILDINC_DIRECTORY) -static -L$(SPDM_FINAL) $(addprefix -l, $(LIBRARIES))
89
90 define compilexx
91 $(CX) -c $(CXXFLAGS) $(1) $< -o $@
92 endef
93
94 define compile
95 $(CC) -c $(CFLAGS) $(1) $< -o $@
96 endef
97
98 define assemble
99 $(CC) -c $(CFLAGS) -o $@ $<
00 endef
```

Figura 17 – Adição do caminho da biblioteca junto as flags de compilação na Common.mak

Após a execução do `Makefile` anteriormente criado, os *headers* da LibSPDM já estão dentro da biblioteca da LiteX para serem utilizados com os caminhos de arquivo devidamente incluídos. Em seguida, deve-se prosseguir com o segundo passo, no qual consiste em modificar o `Makefile` padrão da LiteX para realizar o *link* dos arquivos objetos da LibSPDM compilados anteriormente junto aos arquivos objeto (que serão compilados através do `Makefile`) da BIOS, sendo estes arquivos objetos adicionados como bibliotecas através de duas simples modificações. A primeira consiste em criar uma variável `LIBRARIES` e a segunda se utilizar da *flag* de compilação `-L` que indica o caminho da biblioteca compilada. A modificação no `Makefile` pode ser visualizada nas imagens abaixo, com nota para os diretórios utilizados na variável `SPDM_DIR` que deve ser o local onde a LibSPDM foi colocada:

```
include ../include/generated/variables.mak
include $(SOC_DIRECTORY)/software/common.mak

SPDM_DIR?= /home/user/Documentos/libspdm_docs/libspdm
SPDM_FINAL?= $(SPDM_DIR)/build/lib
SPDM_HEADERS = $(SPDM_DIR)/include
SPDM_CRYPTO?= mbedtls

LIBRARIES = cryptlib_$(SPDM_CRYPTO) $(SPDM_CRYPTO) cmockalib rnglib_null memlib malloclib debuglib spdm_crypt_lib spdm_secured_message_lib spdm_requester_lib
  spdm_common_lib spdm_transport_mctp_lib spdm_transport_pcidoe_lib spdm_device_secret_lib_sample spdm_responder_lib mbedcrypto platform_lib spdm_crypt_ext_lib
  spdm_transport_test_lib mbedx509 debuglib_null

# Permit TFTP_SERVER_PORT override from shell environment / command line
ifdef TFTP_SERVER_PORT
CFLAGS += -DTFTP_SERVER_PORT=$(TFTP_SERVER_PORT)
endif

OBJECTS = boot-helper.o       \
          bswapsi2.o \
          boot.o                    \
          helpers.o                 \
          cmd_bios.o          \
          cmd_mem.o                 \
          cmd_boot.o          \
          cmd_i2c.o                 \
          cmd_spiflash.o      \
          cmd_litedram.o      \
          cmd_liteeth.o             \
          cmd_litesdcard.o  \
          cmd_litesata.o    \
          sim_debug.o         \
          main.o

ifneq "$(or $(BIOS_CONSOLE_NO_AUTOCOMPLETE),$(BIOS_CONSOLE_LITE))" ""
CFLAGS += -DBIOS_CONSOLE_NO_AUTOCOMPLETE
else
OBJECTS += complete.o
endif

ifdef BIOS_CONSOLE_NO_HISTORY
CFLAGS += -DBIOS_CONSOLE_NO_HISTORY
endif

ifdef BIOS_CONSOLE_DISABLE
CFLAGS += -DBIOS_CONSOLE_DISABLE
endif

ifdef BIOS_CONSOLE_LITE
CFLAGS += -DBIOS_CONSOLE_LITE
OBJECTS += readline_simple.o
else
```

Figura 18 – Adiação dos binários da biblioteca LibSPDM

Ao final do `Makefile` pode-se observar a adição do *flag* de compilação `-L`, utilizado para compor a unificação entre as duas bibliotecas.

```
%.elf: crt0.o $(LIBS:%=%.a)
        $(CC) $(LDFLAGS) -T $(BIOS_DIRECTORY)/$(LSCRIPT) -N -o $@ \
                crt0.o \
                $(OBJECTS) \
                $(PACKAGES:%=-L../%) \
                -Wl,--whole-archive \
                -Wl,--gc-sections \
                -Wl,-Map,$@.map \
                $(LIBS:lib%=-l%) -L$(SPDM_FINAL) $(addprefix -l, $(LIBRARIES))

```

Figura 19 – Flag de compilação -L adicionada no Makefile

Além das modificações nos `Makefiles`, é necessário a adição de seis arquivos de bibliotecas externas, adicionados para resolver problemas de funções de referências. Os arquivos foram adicionados juntamente com os demais arquivos de BIOS da LiteX, que usualmente ficam no caminho de arquivos `/litex/litex/soc/software/bios`. Os novos arquivos são de duas bibliotecas abertas, a Compiler RT (COMPILER..., 2023) que contém os arquivos: `bswapsi2.c`, `int_endianness.h`, `int_lib.h`, `int_types.h` e `int_util.h` e a `src` (SRC, 2023) que contém a `stdint.h`. Tais bibliotecas são necessárias pois na BIOS não temos o suporte de um sistema operacional, portanto a LibSPDM não pode ser executada sem o devido suporte. As bibliotecas suprem o mínimo necessário para sua utilização.

Estas bibliotecas, antes de serem utilizadas pela BIOS, precisam de um endereçamento de memória para seus símbolos, ou seja, um espaço de memória alocado separado da pilha e do código do programa para que estes códigos possam utilizar deste espaço para suas funções. sendo assim, há a necessidade de realizar uma modificação no *linker* da BIOS, adicionando um espaço de memória, chamado de `heap`. Pode-se observar na Figura 20 abaixo os espaços de memória que estão sendo utilizados pelo SoC. Estes espaços não podem ser ocupados pelo símbolo `heap`, mais quaisquer daqueles espaços que não são utilizados podem ser ocupados.

```
1 MEMORY {
2         opensbi : ORIGIN = 0x80000000, LENGTH = 0x00200000
3         plic : ORIGIN = 0x0c000000, LENGTH = 0x00400000
4         clint : ORIGIN = 0x02000000, LENGTH = 0x00010000
5         rom : ORIGIN = 0x10000000, LENGTH = 0x00200000
6         sram : ORIGIN = 0x11000000, LENGTH = 0x00040000
7         main_ram : ORIGIN = 0x80000000, LENGTH = 0x08000000
8         ethmac : ORIGIN = 0x30000000, LENGTH = 0x00002000
9         csr : ORIGIN = 0x12000000, LENGTH = 0x00010000
10 }
```

Figura 20 – Espaços de memória no SoC

Como há a possibilidade de escolha, a faixa de endereçamento escolhido foi o de `0x40000000` até `0x40800000`, devido a ser um espaço vazio e com certa distância em relação aos demais, evitando possíveis problemas de sobreposição da memória. Esta faixa de endereços, juntamente com o símbolo de `heap` devem ser adicionados dentro do arquivo `linker.ld`, uma vez que este realiza o trabalho de *link* dentro da BIOS. Tal tarefa pode ser visualizada na Figura 21 abaixo, com o destaque para as linhas de 72 até 79, que contém a faixa de endereço escolhida para o símbolo:

```
68
69          /* Use the top of RAM and downwards for the stack: */
70          __stack_top = 0x00000000;
71
72          /* Added heap_start and heap_end */
73          _heap_start = .;
74                  __heap_start = .;
75                  . +=0x40000000 ;
76
77          _heap_end = .;
78                  __heap_end = .;
79                  . +=0x40800000 ;
80
81          .bss :
82          {
83                  . = ALIGN(8);
84                  _fbss = .;
85                  *(.dynsbss)
86                  *(.sbss .sbss.* .gnu.linkonce.sb.*)
87                  *(.scommon)
88                  *(.dynbss)
89                  *(.bss .bss.* .gnu.linkonce.b.*)
90                  *(COMMON)
91                  . = ALIGN(8);
92                  _ebss = .;
93                  _end = .;
94          } > sram
```

Figura 21 – Modificação do Linker.ld para comportar o símbolo Heap

Por fim, antes de criar as funções para o uso do SPDM, é necessário uma simples modificação no tamanho da memória do SoC. O SPDM, ao ser utilizado, precisa realizar alocação de memória, tanto para inicializar um contexto de troca de mensagens entre o *requester* e o *responder*, quanto para guardar as mensagens em *buffers*. As áreas alocadas persistem até o final da comunicação, é necessário que a quantidade de memória alocada seja suficiente. A memória de base da ROM foi aumentada de `0x20000` para `0x200000` e a memória SRAM de `0x2000` para `0x40000`. Estas mudanças foram realizadas no arquivo `soc_core.py` para que as configurações de tamanho da memória se tornem o padrão do SoC. O arquivo que deve ser modificado se encontra no caminho `/litex/litex/soc/integration`. Não há qualquer risco de mudança na base dos endereçamentos, ou seja, o `heap` anteriormente adicionado não precisa ser modificado.

```
271     # ROM parameters
272     soc_group.add_argument("--integrated-rom-size", default=0x200000, type=auto_int,
    smaller).")
273     soc_group.add_argument("--integrated-rom-init", default=None,    type=str,
274
275     # SRAM parameters
276     soc_group.add_argument("--integrated-sram-size", default=0x40000, type=auto_int,
277
```

Figura 22 – Mudança nos tamanhos padrão da memória SRAM e ROM.

Com as configurações de compilação devidamente aplicadas para receber a LibSPDM, pode-se iniciar a construção das funções de SPDM para a BIOS. Neste caso será construído funções para o *requester* e para o *responder*, ou seja, a BIOS deve ser capaz de participar de um canal de comunicação com SPDM seja qual for o seu papel.

Todas as funções construídas para o SPDM serão escritas em um arquivo separado, ou seja, dentro do caminho onde se encontra os demais arquivos da BIOS, usualmente `/litex/litex/soc/software/bios`. Este novo arquivo foi nomeado de `spdmfuncs` e possui apenas duas funções acessíveis, uma que inicia o *requester* e outra que inicia o *responder*. A Figura 23 mostra a declaração das funções descritas em um arquivo de *header* nomeado de `spdmfuncs.h`.

```
1 #ifndef SPDMFUNCS_H
2 #define SPDMFUNCS_H
3
4 void spdm_responder(void);
5 void spdm_requester(void);
6
7 #endif
```

Figura 23 – Funções da BIOS declaradas

Estas funções principais e suas secundárias estão declaradas em um arquivo `spdmfuncs.c`, que devido a sua complexidade não mostraemos aqui (as funções estão disponíveis no repositório Git (SPDM. . . , 2023)). A Figura 24 mostra o início da declaração da função de *requester*, utilizada neste documento como exemplo:

```
310 void spdm_requester(void)
311 {
312         void *spdm_bios_context;
313         void *m_spdm_bios_context;
314
315         size_t request_size_buffer;
316         void *request_scratch_buffer;
317
318         size_t spdm_context_size;
319
320         libspdm_data_parameter_t parameter;
321
322         uint8_t data8 = 0x0;
323
324         libspdm_return_t status;
325
326         spdm_version_number_t spdm_version = 0x2;
327
328         uint8_t m_use_version = 0;
329         uint32_t *session_id = 0;
330
331         spdm_context_size = libspdm_get_context_size();
332         printf("Using %zu \n", spdm_context_size);  // prints as bytes
333
334         m_spdm_bios_context = (void *)malloc(spdm_context_size); //spdm_context_size
335
336         if (m_spdm_bios_context == NULL) {
337                 printf("Erro");
338                 return NULL;
339         }
340         else
341            printf("Ok \n");
342
343         spdm_bios_context = m_spdm_bios_context;
344
345         libspdm_init_context(spdm_bios_context);
346
347         printf("Context Initialized \n");
```

Figura 24 – Função de SPDM Requester dentro da BIOS

Adicionar um arquivo que contém as funções não é suficiente para o pleno funcionamento do protocolo, ainda é necessário alterar novamente o `Makefile` demonstrado na Figura 18 para que as novas funções sejam transformadas em código objeto e incluídas durante o tempo de compilação. A modificação é simples, basta realizar duas adições. A primeira consiste em alterar uma biblioteca na variável `LIBRARIES` (linha 9) de onde será retirada a biblioteca `spdm_device_secret_lib_sample` em troca do código objeto `spdm_device_secret_lib_null`. Isto é necessário para evitar erros de referências não encontradas. A segunda modificação se encontra na linha 30 da Figura 25 onde podemos ver a adição do código objeto `spdmfuncs.o`, garantindo que, ao ser gerado o arquivo binário final, as funções SPDM sejam incluídas.

```
LIBRARIES = cryptlib_$(SPDM_CRYPTO) $(SPDM_CRYPTO) cmockalib rnglib_null memlib malloclib debuglib spdm_crypt_lib spdm_secured_message_lib spdm_requester_lib
  spdm_common_lib spdm_transport_mctp_lib spdm_transport_pcidoe_lib spdm_device_secret_lib_null spdm_responder_lib mbedcrypto platform_lib spdm_crypt_ext_lib
  spdm_transport_test_lib mbedx509 debuglib_null

# Permit TFTP_SERVER_PORT override from shell environment / command line
ifdef TFTP_SERVER_PORT
CFLAGS += -DTFTP_SERVER_PORT=$(TFTP_SERVER_PORT)
endif

OBJECTS = boot-helper.o       \
          bswapsi2.o \
          boot.o                      \
          helpers.o                   \
          cmd_bios.o          \
          cmd_mem.o                   \
          cmd_boot.o          \
          cmd_i2c.o                   \
          cmd_spiflash.o      \
          cmd_litedram.o      \
          cmd_liteeth.o               \
          cmd_litesdcard.o  \
          cmd_litesata.o    \
          sim_debug.o         \
          spdmfuncs.o \
          main.o
```

Figura 25 – Makefile com adição da compilação das funções de SPDM

Por fim, com todos os parâmetros de compilação devidamente ajustados, as novas funções foram adicionadas como comandos da BIOS e assim podem ser chamadas por comandos pelo terminal da entrada serial. Nos arquivos de comandos, no caminho `/litex/litex/soc/software/bios/cmds`, dentro dos comandos descritos em `cmds_bios.c`, foram adicionadas duas modificações. A primeira modificação realizada é o caminho das funções descritas em `spdmfuncs.h`. Para isto basta adicionar uma linha de `include` junto as demais já existentes, assim, basta escrever:

```
#include <bios/spdmfuncs.h>
```

Com as funções devidamente adicionadas, será criado comandos que se utilizam delas, o que na prática será somente uma chamada de função já existente. A Figura 26 mostra a construção destas duas funções de comandos:

```
/** SPDM Commands **/

#ifdef CSR_ETHPHY_BASE
static void spdm_responder_handler(char **params)
{
        spdm_responder();
}
define_command(spdm_responder, spdm_responder_handler, "Set BIOS to use SPDM as responder", SYSTEM_CMDS);
#endif

#ifdef CSR_ETHPHY_BASE
static void spdm_requester_handler(char **params)
{
        spdm_requester();
}
define_command(spdm_requester, spdm_requester_handler, "Set BIOS to use SPDM as requester", SYSTEM_CMDS);
#endif

```

Figura 26 – Funções SPDM adicionadas como comandos na BIOS

Com todas as modificações de compilação e com as novas funções devidamente aplicadas, basta recompilar o SoC como descrito na seção 5.2.3.

5.4.2 Integração da LibSPDM no driver da Ethernet

Após a implementação da LibSPDM na BIOS do SoC, deve-se seguir para a integração da mesma biblioteca dentro do Kernel e, portanto, no *driver* da placa de rede, logo, mesmo ao se utilizar parte do sistema construído na seção 5.2.2. Ainda assim, será necessário realizar modificações tanto em quesitos de compilação na LibSPDM como em configurações no *Buildroot*.

A primeira alteração se refere a realizar mudanças nas *flags* de compilação. Considerando a Seção 5.2.5, as *flags* devem ser trocadas por aquelas fornecidas pelo próprio *Buildroot* para que se tenha uma compatibilidade no momento de execução. A figura 27 mostra as novas opções adicionadas, especialmente em dois campos da variável `ADD_COMPILE_OPTIONS`, nas linhas 353 e linha 361:

```
353         ADD_COMPILE_OPTIONS(-D__KERNEL__ -fmacro-prefix-map=./= -Wall -Wundef -Wno-trigraphs -fno-strict-aliasing -fno-common -fshort-wchar -fno-PIE -Werror=implicit-
    function-declaration -Werror=implicit-int -Werror=return-type -Wno-format-security -funsigned-char -std=gnu11 -mabi=lp64 -march=rv64imac -mno-save-restore -
    DCONFIG_PAGE_OFFSET=0xff60000000000000 -mcmodel=medany -fno-omit-frame-pointer -fno-asynchronous-unwind-tables -fno-unwind-tables -mno-riscv-attribute -Wa,-mno-arch-attr -
    mstrict-align -fno-delete-null-pointer-checks -Wno-frame-address -Wno-format-truncation -Wno-format-overflow -Wno-address-of-packed-member -O2 -fno-allow-store-data-races
    Wframe-larger-than=2048 -fstack-protector-strong -Wno-main -Wno-unused-but-set-variable -Wno-unused-const-variable -fno-omit-frame-pointer -fno-optimize-sibling-calls -fno
    stack-clash-protection -Wdeclaration-after-statement -Wvla -Wno-pointer-sign -Wcast-function-type -Wno-stringop-truncation -Wno-stringop-overflow -Wno-restrict -Wno-maybe-
    uninitialized -Wno-array-bounds -Wno-alloc-size-larger-than -Wimplicit-fallthrough=5 -fno-strict-overflow -fno-stack-check -fconserve-stack -Werror=date-time -
    Werror=incompatible-pointer-types -Werror=designated-init -Wno-packed-not-aligned -Wno-attribute-alias -mstack-protector-guard=tls -mstack-protector-guard-reg=tp -mstack-
    protector-guard-offset=1080 -fno-function-sections -fno-data-sections)
354         if(CMAKE_BUILD_TYPE STREQUAL "Debug")
355            ADD_COMPILE_OPTIONS(-g)
356         endif()
357         if(GCOV STREQUAL "ON")
358         ADD_COMPILE_OPTIONS(--coverage -fprofile-arcs -ftest-coverage)
359         endif()
360         SET(OPENSSL_FLAGS -include base.h -Wno-error=maybe-uninitialized -Wno-error=format -Wno-format -Wno-error=unused-but-set-variable -Wno-cast-qual)
361         ADD_COMPILE_OPTIONS(-DBROOT_RV64) # bignum.c problem with bswap
362         SET(CMOCKA_FLAGS -std=gnu99 -Wpedantic -Wall -Wshadow -Wmissing-prototypes -Wcast-align -Werror=address -Wstrict-prototypes -Werror=strict-prototypes -Wwrite-
    strings -Werror=write-strings -Werror-implicit-function-declaration -Wpointer-arith -Werror=pointer-arith -Wdeclaration-after-statement -Werror=declaration-after-statement
    Wreturn-type -Werror=return-type -Wuninitialized -Werror=uninitialized -Werror=strict-overflow -Wstrict-overflow=2 -Wno-format-zero-length -Wmissing-field-initializers -
    Wformat-security -Werror=format-security -fno-common -Wformat -fno-common -fstack-protector-strong -Wno-cast-qual)
363
364         SET(CMAKE_EXE_LINKER_FLAGS "-flto -Wno-error -no-pie" )
365         if(GCOV STREQUAL "ON")
366         SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS}  --coverage -lgcov -fprofile-arcs -ftest-coverage")
367         endif()
368
369         SET(CMAKE_C_LINK_EXECUTABLE "<CMAKE_LINKER> <LINK_FLAGS> <OBJECTS> -o <TARGET> -Wl,--start-group <LINK_LIBRARIES> -Wl,--end-group")
```

Figura 27 – Flags de compilação para a LibSPDM no Kernel Linux

Com as devidas *flags* de compilação adicionadas para o *Buildroot* deve-se ainda realizar mudanças em um código disponível na LibSPDM, no arquivo `bignum.c` disponível no caminho `libspdm/os_stub/mbedtlslib/mbedtls/library`. As mudanças são em suma apenas duas, a primeira se concentra em adicionar uma definição, as adições da linha `#if !defined (BROOT_RV64)` e `#endif /*BROOT_RV64*/` nas linhas 725 e 751 da Figura 28, que impede o uso da função `bswap` que não possui referência nos arquivos de compilação. Não há impacto nas funções da LibSPDM.

```
716 #if defined(__BYTE_ORDER__)
717
718 /* Nothing to do on bigendian systems. */
719 #if ( __BYTE_ORDER__ == __ORDER_BIG_ENDIAN__ )
720     return( x );
721 #endif /* __BYTE_ORDER__ == __ORDER_BIG_ENDIAN__ */
722
723 #if ( __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__ )
724
725 #if !defined(BROOT_RV64)
726 /* For GCC and Clang, have builtins for byte swapping. */
727 #if defined(__GNUC__) && defined(__GNUC_PREREQ)
728 #if __GNUC_PREREQ(4,3)
729 #define have_bswap
730 #endif
731 #endif
732
733 #if defined(__clang__) && defined(__has_builtin)
734 #if __has_builtin(__builtin_bswap32)  &&                  \
735     __has_builtin(__builtin_bswap64)
736 #define have_bswap
737 #endif
738 #endif
739
740 #if defined(have_bswap)
741     /* The compiler is hopefully able to statically evaluate this! */
742     switch( sizeof(mbedtls_mpi_uint) )
743     {
744         case 4:
745             return( __builtin_bswap32(x) );
746         case 8:
747             return( __builtin_bswap64(x) );
748     }
749 #endif
750 #endif /* __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__ */
751 #endif /* BROOT_RV64 */
752 #endif /* __BYTE_ORDER__ */
753
```

Figura 28 – Mudanças na função de bignum.c

A segunda mudança no arquivo `bignum.c` refere-se à mudança de escrita de um `for`, que pode ser visualizado na linha 1997 da Figura 29.

```
1992 static int mpi_select( mbedtls_mpi *R, const mbedtls_mpi *T, size_t T_size, size_t idx )
1993 {
1994     int ret = MBEDTLS_ERR_ERROR_CORRUPTION_DETECTED;
1995
1996     //for( size_t i = 0; i < T_size; i++ )
1997     size_t i = 0;
1998     for( i = 0; i< T_size; i++ )
1999     {
2000         MBEDTLS_MPI_CHK( mbedtls_mpi_safe_cond_assign( R, &T[i],
2001                          (unsigned char) mbedtls_ct_size_bool_eq( i, idx ) ) );
2002     }
2003
2004 cleanup:
2005     return( ret );
2006 }
```

Figura 29 – Segunda mudança na função bignum.c

Com a LibSPDM devidamente configurada para a compilação em *Buildroot*, volta-se ao diretório padrão onde se encontra a LibSPDM e realiza-se novamente a compilação (será realizado uma compilação em outro diretório, assim, não se perde a compilação realizada para a BIOS da LiteX na seção 5.2.5). Seguem os seguintes comandos:

```
$ mkdir build_buildroot
$ cd build_buildroot
$ export PATH=$PATH:/opt/riscv/bin
$ export PATH="/PATH/TO/buildroot-2023.05.1/output/host/bin:$PATH"
$ cmake -DARCH=riscv64 -DTOOLCHAIN=RISCV_GNU -DTARGET=Release -DCRYPTO=mbedtls
$ make copy_sample_key
$ make
```

Após a devida compilação da LibSPDM, deve-se partir para as configurações do *Buildroot*. Primeiramente será trocado o tipo de compilador utilizado na construção do Kernel, pois as versões de compiladores das bibliotecas e do Kernel devem ser compatíveis entre si. Para isto ativamos dentro do *Buildroot* a opção de *external toolchain*. Dentro do ambiente de configurações (arquivo `.config`) do *Buildroot* haverá as seguintes configurações habilitadas:

```
BR2_TOOLCHAIN_EXTERNAL_CUSTOM=y
BR2_TOOLCHAIN_EXTERNAL_PREINSTALLED=y
BR2_TOOLCHAIN_EXTERNAL_PATH="/opt/riscv/"
BR2_TOOLCHAIN_EXTERNAL_GLIBC=y
BR2_PACKAGE_HAS_TOOLCHAIN_EXTERNAL=y
BR2_PACKAGE_PROVIDES_TOOLCHAIN_EXTERNAL="toolchain-external-custom"
BR2_TOOLCHAIN_EXTERNAL_PREFIX="$(ARCH)-unknown-linux-gnu"
BR2_TOOLCHAIN_EXTERNAL_CUSTOM_PREFIX="$(ARCH)-unknown-linux-gnu"
BR2_TOOLCHAIN_EXTERNAL_GCC_12=y
```

```
BR2_TOOLCHAIN_EXTERNAL_HEADERS_5_10=y
BR2_TOOLCHAIN_EXTERNAL_CUSTOM_GLIBC=y
BR2_TOOLCHAIN_EXTERNAL_HAS_SSP=y
BR2_TOOLCHAIN_EXTERNAL_HAS_SSP_STRONG=y
BR2_TOOLCHAIN_EXTERNAL_CXX=y
BR2_TOOLCHAIN_EXTERNAL_FORTRAN=y
BR2_PACKAGE_HOST_GDB_ARCH_SUPPORTS=y
```

Ao entrar dentro do ambiente de configuração do *Buildroot*, no qual pode igualmente manipular o arquivo de configuração, temos o seguinte resultado:

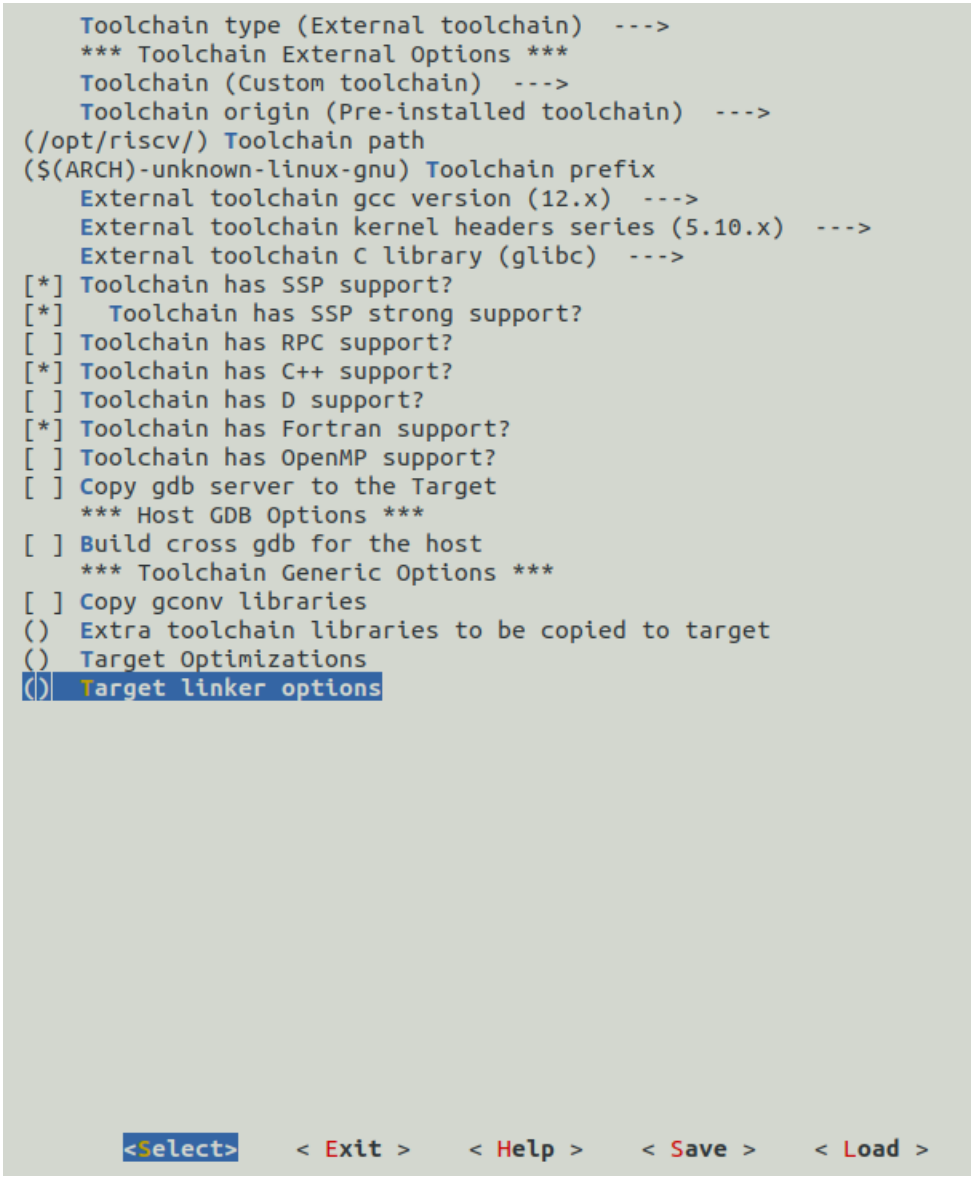

Figura 30 – Configuração de toolchain externa dentro do Buildroot

Passado das configurações de compilador, pode ser necessário igualmente habilitar opções referentes a criptografia para que os algoritmos que podem vir a ser utilizados

em um *requester* ou *responder*. Estas configurações devem ser habilitadas caso se deseje utilizar o protocolo de chaves ou o uso de certificados. As configurações que devem ser habilitadas são as seguintes:

```
CONFIG_ASYMMETRIC_KEY_TYPE
CONFIG_ASYMMETRIC_PUBLIC_KEY_SUBTYPE
CONFIG_X509_CERTIFICATE_PARSER
CONFIG_SYSTEM_TRUSTED_KEYRING
CONFIG_SYSTEM_TRUSTED_KEYS
CONFIG_TCG_TPM
CONFIG_KEYS
CONFIG_CRYPTO_MANAGER
CONFIG_CRYPTO_RSA
CONFIG_CRYPTO_SHA256
```

Posteriormente às configurações no *Buildroot* estarem devidamente aplicadas, as próximas modificações são referentes somente aos arquivos de compilação do Kernel Linux que foi contruído dentro do *Buildroot*. Todas as seguintes modificações dentro do Kernel são realizadas dentro do caminho `buildroot-2023.05.01/output/build/linux-6.1.26`.

Existem três mudanças de compilação que devem ser realizadas para o pleno funcionamento da LibSPDM dentro do Kernel, a primeira destas se refere a adicionar dentro do caminho de `lib` um novo diretório no qual será chamado de `spdm`. Este novo diretório irá conter, um novo `Makefile`, responsável por adicionar os arquivos binários (já compilados anteriormente) e realizar a compilação para código objeto de um arquivo extra, este novo arquivo será chamado de `spdm_glue.c`, responsável por introduzir funções necessários ao SPDM. O arquivo de `Makefile` pode ser visto nas figuras 31 e 32, com notas para as duas primeiras linhas, que precisam ser, respectivamente, o caminho do diretório onde se encontra a LibSPDM e os arquivos compilados da mesma:

```
SPDM_DIR = ~/Documentos/libspdm_docs/libspdm
SPDM_BUILD_DIR = $(SPDM_DIR)/build_buildroot
SPDM_CRYPTO = mbedtls
SPDM_HEADERS = $(SPDM_DIR)/include

SPDM_INCLUDE := -Iinclude/spdm -Iinclude/spdm/hal -Iinclude/spdm/internal -Iinclude/spdm/include

CFLAGS_spdm_glue.o = $(SPDM_INCLUDE)

SPDM_INCLUDE_DIR =  \
        $(SPDM_HEADERS) \
        $(SPDM_HEADERS)/hal \
        $(SPDM_HEADERS)/internal \
        $(SPDM_HEADERS)/library \
        $(SPDM_HEADERS)/industry_standard \
        $(SPDM_DIR)/library/spdm_common_lib \
        $(SPDM_DIR)/library/spdm_requester_lib \
        $(SPDM_DIR)/library/spdm_secured_message_lib

SPDM_HEADER_DIR_KERNEL := $(foreach incdir, $(SPDM_INCLUDE_DIR), $(addprefix include/spdm/,$(notdir $(incdir))))

$(SPDM_HEADER_DIR_KERNEL):
        mkdir -p include/spdm
        ln -s $(filter %$(notdir $@), $(SPDM_INCLUDE_DIR)) $@
        if ls $@/*.h >/dev/null 2>&1; then for i in $@/*.h; do ln -s ../../$$i include/spdm/; done; fi

SPDM_LIBS = \
        $(SPDM_BUILD_DIR)/lib/libcryptlib_mbedtls.a \
        $(SPDM_BUILD_DIR)/lib/libdebuglib_null.a \
        $(SPDM_BUILD_DIR)/lib/libmalloclib_simple.a \
        $(SPDM_BUILD_DIR)/lib/libmbedcrypto.a \
        $(SPDM_BUILD_DIR)/lib/libmbedtls.a \
        $(SPDM_BUILD_DIR)/lib/libmbedx509.a \
        $(SPDM_BUILD_DIR)/lib/libmemlib.a \
        $(SPDM_BUILD_DIR)/lib/libplatform_lib_null.a \
        $(SPDM_BUILD_DIR)/lib/librnglib_null.a \
        $(SPDM_BUILD_DIR)/lib/libspdm_common_lib.a \
        $(SPDM_BUILD_DIR)/lib/libspdm_crypt_lib.a \
        $(SPDM_BUILD_DIR)/lib/libspdm_crypt_ext_lib.a \
        $(SPDM_BUILD_DIR)/lib/libspdm_device_secret_lib_null.a \
        $(SPDM_BUILD_DIR)/lib/libspdm_requester_lib.a \
        $(SPDM_BUILD_DIR)/lib/libspdm_secured_message_lib.a \
        $(SPDM_BUILD_DIR)/lib/libspdm_transport_mctp_lib.a\
        $(SPDM_BUILD_DIR)/lib/libspdm_transport_pcidoe_lib.a
```

Figura 31 – Makefile para LibSPDM - 1

```

SPDM_DIR_SUFIX = _spdmlib
SPDM_OBJ  := $(foreach libfile, $(SPDM_LIBS), $(shell ar t $(libfile) | sed s-^-$$(basename $(libfile))$(SPDM_DIR_SUFIX)/-))
SPDM_TARGETS := $(addprefix $(obj)/, $(SPDM_OBJ))

spdm_glue-objs := spdm_glue.o $(SPDM_OBJ)
obj-y += spdm_glue.o

$(SPDM_TARGETS): $(SPDM_LIBS) $(SPDM_HEADER_DIR_KERNEL)
        mkdir -p $$(dirname $@)
        cd $$(dirname $@); ar x $(SPDM_BUILD_DIR)/lib/$$(basename $$(dirname $@ | sed s/$(SPDM_DIR_SUFIX)//)) $$(basename $@)
```

Figura 32 – Makefile para LibSPDM - 2

Após adicionar o novo `Makefile` para incluir os códigos objetos da LibSPDM, deve-se garantir que o Kernel irá adicionar o código objeto de `spdm_glue`. Para que o novo código objeto seja adicionado como objeto dentro do Kernel Linux deve-se modificar o `Makefile` que se encontra no caminho `buildroot-2023.05.01/output/build/linux-6.1.26/lib`, adicionando uma linha ao final para indicar o objeto e a localização do mesmo nos diretórios. Esta nova linha de código dentro do `Makefile` pode ser escrita como:

```
obj-y += spdm/
```

Por fim, antes de realizar uma nova compilação do Kernel Linux, o caminho onde se encontram os `headers` da LibSPDM devem ser adicionados para que durante

o tempo de compilação, os arquivos sejam encontrados para poderem ser utilizados pelo *driver* da Ethernet. O `Makefile` que deve ser modificado se encontra no caminho `buildroot-2023.05.01/output/build/linux-6.1.26/arch/riscv`. As modificações que se deve realizar são simples, basta criar duas variáveis, `SPDM_DIR` e `SPDM_HEADERS` (linhas 9 e 10 da 33, no qual contém, respectivamente, os caminhos de onde se encontra a LibSPDM e o caminho dos *headers*, além disto, deve-se incluir uma nova variável chamada de `INCLUDES` (linha 12 da 33), esta contém a *flag* de compilação -I no qual indica o caminho que o compilador deve procurar ao realizar o seu trabalho. A figura 33 mostra o que foi descrito:

```
1 # This file is included by the global makefile so that you can add your own
2 # architecture-specific flags and dependencies.
3 #
4 # This file is subject to the terms and conditions of the GNU General Public
5 # License.  See the file "COPYING" in the main directory of this archive
6 # for more details.
7 #
8
9 SPDM_DIR?= /home/user/Documentos/libspdm_docs/libspdm
10 SPDM_HEADERS = $(SPDM_DIR)/include
11
12 INCLUDES = -I$(SPDM_HEADERS) \
13            -I$(SPDM_HEADERS)/hal \
14            -I$(SPDM_HEADERS)/library \
15            -I$(SPDM_HEADERS)/industry_standard \
16            -I$(SPDM_HEADERS)/internal \
17            -I$(SPDM_DIR)/library/spdm_common_lib \
18            -I$(SPDM_DIR)/library/spdm_crypt_lib \
19            -I$(SPDM_DIR)/library/spdm_requester_lib \
20            -I$(SPDM_DIR)/library/spdm_responder_lib \
21            -I$(SPDM_DIR)/library/spdm_transport_mctp_lib \
22            -I$(SPDM_DIR)/library/spdm_transport_pcidoe_lib \
23            -I$(SPDM_DIR)/library/spdm_secured_message_lib
24
```

Figura 33 – Makefile para inclusão de headers

Após criar a variável com a *flag* de compilação -I que contém os caminhos dos *headers*, deve-se adicionar a nova variável como comando ao compilador. A figura 34 em sua linha 122 demonstra como deve ser adicionado dentro de outra variável `KBUILD_CFLAGS`, desta vez, uma variável que será passada diretamente ao compilador.

```
105 # Avoid generating .eh_frame sections.
106 KBUILD_CFLAGS += -fno-asynchronous-unwind-tables -fno-unwind-tables
107
108 # The RISC-V attributes frequently cause compatibility issues and provide no
109 # information, so just turn them off.
110 KBUILD_CFLAGS += $(call cc-option,-mno-riscv-attribute)
111 KBUILD_AFLAGS += $(call cc-option,-mno-riscv-attribute)
112 KBUILD_CFLAGS += $(call as-option,-Wa$(comma)-mno-arch-attr)
113 KBUILD_AFLAGS += $(call as-option,-Wa$(comma)-mno-arch-attr)
114
115 KBUILD_CFLAGS_MODULE += $(call cc-option,-mno-relax)
116 KBUILD_AFLAGS_MODULE += $(call as-option,-Wa$(comma)-mno-relax)
117
118 # GCC versions that support the "-mstrict-align" option default to allowing
119 # unaligned accesses.  While unaligned accesses are explicitly allowed in the
120 # RISC-V ISA, they're emulated by machine mode traps on all extant
121 # architectures.  It's faster to have GCC emit only aligned accesses.
122 KBUILD_CFLAGS += $(call cc-option,-mstrict-align) $(INCLUDES)
123
124 ifeq ($(CONFIG_STACKPROTECTOR_PER_TASK),y)
125 prepare: stack_protector_prepare
126 stack_protector_prepare: prepare0
127         $(eval KBUILD_CFLAGS += -mstack-protector-guard=tls            \
128                                 -mstack-protector-guard-reg=tp                \
129                                 -mstack-protector-guard-offset=$(shell        \
130                         awk '{if ($$2 == "TSK_STACK_CANARY") print $$3;}' \
131                                         include/generated/asm-offsets.h))
132 endif
```

Figura 34 – Adição da flag de compilação

Ao fim, basta substituir o *driver* da placa de rede Ethernet `litex_liteeth.c` e recompilar o Kernel Linux:

```
$ make linux-rebuild
```

6 Resultados e Considerações Finais

6.1 Conclusões do Projeto de Formatura

Como proposto na seção de objetivos, este trabalho teve como principal meta a descoberta e medição do desempenho do uso de memória do SPDM em hardware. Consideramos que a implementação foi construída com sucesso, através do SoC com a devida implementação da LibSPDM na BIOS e no *driver* da placa de rede, onde conseguimos realizar medições sobre o uso de memória.

O SPDM, para realizar a sua comunicação, precisa de dois elementos chave tanto para o *requester* como para o *responder*. Estes elementos são o contexto e o *buffer*. O contexto armazena informações de comunicação como algoritmos disponíveis para serem utilizados, versão do protocolo e as capacidades de cada parte. O segundo (*buffer* é utilizado para armazenar as trocas de mensagens entre o *requester* e o *responder*. Ambos são construídos com alocações de memória padrões, ou seja, se utilizam da função de `malloc`, disponível na biblioteca padrão da linguagem C.

Embutindo as funções dentro da BIOS, foi possível observar que os valores de armazenamento, medidos em bytes, são alocados de maneiras constantes como pode ser observado na Tabela 6.1.

Tabela 1 – Memória na BIOS.

Funcionalidade	Tamanho em Bytes
Contexto - Requester	23208
Contexto - Responder	23208
Buffer - Requester	18432
Buffer - Responder	18432

Os valores, tanto para o contexto como para o *buffer*, não se alteram. Para esta demonstração foram estabelecidas quatro conexões simultâneas na mesma placa FPGA (duas de *requester* e duas de *responder*). Todas as conexões se utilizam da mesma quantidade de memória disponível e não se observou qualquer variação na mesma.

```
litex> spdm_responder
Memory used for responder context (in bytes): 23208
Context Initialzed
Memory used for responder buffer (in bytes): 18432

LIBSPDM_DATA_SPDM_VERSION - 0x0
LIBSPDM_DATA_SECURED_MESSAGE_VERSION - 0x0
LIBSPDM_DATA_CAPABILITY_CT_EXPONENT - 0x0
LIBSPDM_DATA_CAPABILITY_FLAGS - 0x0
LIBSPDM_DATA_MEASUREMENT_SPEC - 0x0
LIBSPDM_DATA_MEASUREMENT_HASH_ALGO - 0x0
LIBSPDM_DATA_BASE_ASYM_ALGO - 0x0
LIBSPDM_DATA_BASE_HASH_ALGO - 0x0
LIBSPDM_DATA_DHE_NAME_GROUP - 0x0
LIBSPDM_DATA_AEAD_CIPHER_SUITE - 0x0
LIBSPDM_DATA_REQ_BASE_ASYM_ALG - 0x0
LIBSPDM_DATA_KEY_SCHEDULE - 0x0
LIBSPDM_DATA_OTHER_PARAMS_SUPPORT - 0x0
LIBSPDM_DATA_HEARTBEAT_PERIOD - 0x0

Message size spdm_responder_receive_message 4672
SPDM Message from requester: 0x0

libspdm_responder_dispatch_message - 0x80010001

litex> spdm_responder
Memory used for responder context (in bytes): 23208
Context Initialzed
Memory used for responder buffer (in bytes): 18432

LIBSPDM_DATA_SPDM_VERSION - 0x0
LIBSPDM_DATA_SECURED_MESSAGE_VERSION - 0x0
LIBSPDM_DATA_CAPABILITY_CT_EXPONENT - 0x0
LIBSPDM_DATA_CAPABILITY_FLAGS - 0x0
LIBSPDM_DATA_MEASUREMENT_SPEC - 0x0
LIBSPDM_DATA_MEASUREMENT_HASH_ALGO - 0x0
LIBSPDM_DATA_BASE_ASYM_ALGO - 0x0
LIBSPDM_DATA_BASE_HASH_ALGO - 0x0
LIBSPDM_DATA_DHE_NAME_GROUP - 0x0
LIBSPDM_DATA_AEAD_CIPHER_SUITE - 0x0
LIBSPDM_DATA_REQ_BASE_ASYM_ALG - 0x0
LIBSPDM_DATA_KEY_SCHEDULE - 0x0
LIBSPDM_DATA_OTHER_PARAMS_SUPPORT - 0x0
LIBSPDM_DATA_HEARTBEAT_PERIOD - 0x0

Message size spdm_responder_receive_message 4672
SPDM Message from requester: 0x0

libspdm_responder_dispatch_message - 0x80010001

litex>
```

Figura 35 – Estabelecimento de dois *responders*

Em relação às mensagens SPDM transmitidas durante uma conexão, há variações quanto ao uso de memória qua acompanham o tipo de protocolo utilizado na camada

de transporte. Neste trabalho se utilizou a camada do tipo `pci doe`, um protocolo de tamanho fixo de bits. Uma outra possibilidade era se utilizar do protocolo `mctp` mas, neste caso podem haver mudanças no tamanho dos dados transmitidos.

Em relação ao uso do SPDM dentro do Kernel Linux, em especial o *responder* que foi implementado neste trabalho, os resultados obtidos foram semelhantes. O SPDM não realiza diferenças sobre quais os componentes estão em sua comunicação, portanto o Kernel utilizou a mesma quantidade de memória para inicialização do *buffer* e do contexto. Além disto, mesmo ao se utilizar menores capacidades do protocolo, a reserva de memória é a mesma.

```
# [   17.922764] Memory used for requester context (in bytes): 23208
[   17.927730] Ok
[   17.932164] Context Initialized
[   17.934194] Memory used for requester buffer (in bytes): 18432
[   17.940234] LIBSPDM_DATA_SPDM_VERSION - 0x0
[   17.944696] LIBSPDM_DATA_SECURED_MESSAGE_VERSION - 0x0
[   17.949386] LIBSPDM_DATA_CAPABILITY_CT_EXPONENT - 0x0
[   17.954794] LIBSPDM_DATA_CAPABILITY_FLAGS - 0x0
[   17.958934] LIBSPDM_DATA_MEASUREMENT_SPEC - 0x0
[   17.963816] LIBSPDM_DATA_BASE_ASYM_ALGO - 0x0
[   17.967794] LIBSPDM_DATA_BASE_HASH_ALGO - 0x0
[   17.972478] LIBSPDM_DATA_DHE_NAME_GROUP - 0x0
[   17.976472] LIBSPDM_DATA_AEAD_CIPHER_SUITE - 0x0
[   17.981070] LIBSPDM_DATA_REQ_BASE_ASYM_ALG - 0x0
[   17.986032] LIBSPDM_DATA_KEY_SCHEDULE - 0x0
[   17.989850] LIBSPDM_DATA_OTHER_PARAMS_SUPPORT - 0x0
```

Figura 36 – Uso de memória para o SPDM dentro do Kernel Linux

6.2 Contribuições

Devido a natureza do trabalho, as contribuições se estendem na integração do protocolo SPDM dentro de um dispositivo físico, ou seja, em termos técnicos temos a criação de um SoC com capacidade de executar nativamente, a partir de seu próprio hardware, um protocolo de autenticação de *firmware*. Nas capacidades cientificas as contribuições se referem às medições de desempenho do protocolo. Ao se utilizar um sistema embarcado para tal feito, retira-se a necessidade de um ambiente emulado como, por exemplo o QEMU, e parte-se para um sistema computacional real que pode ter seu desempenho de uso de memória medido de fato e não estimado através de um modelo. Visto que ambientes como sistemas embarcados possuem uma baixa quantidade de memória e uma menor velocidade de processamento de seus dados, a medição precisa torna-se uma necessidade.

6.3 Perspectivas de Continuidade

A implementação em hardware para o protocolo SPDM não foi realizada para toda a execução do protocolo, principalmente devido a necessidade de tempo. O protocolo foi implementado e avaliado somente até a sua terceira etapa, negociação de algoritmos. Para utilização real o protocolo ainda possui outras funcionalidades, como por exemplo requisição de certificados e troca de chaves, que precisam ser implementadas para uma total avaliação de seu desempenho. Além disto, o SPDM neste trabalho foi realizado para questões da BIOS e do *driver* da placa de rede Ethernet. Em um sistema que utiliza SPDM com o proósito de proteção desde o momento zero (conhecido como *boot* seguro), todos os componentes usados até a carga e execução do Kernel precisam ser autenticados. Por exemplo, o *bootloader* contido em uma ROM que suporte SPDM ou uma porta USB ativa durante o *boot* podem ser autenticados pelo SPDM. Os dois periféricos citados como exemplo possuem ataques documentados na literatura científica, portanto o SPDM pode ser uma alternativa para evitá-los se implementado na sua completude.

Por fim, além das métricas retiradas sobre o uso de memória no protocolo, há a expectativa futura de se obter métricas em relação ao tempo de execução, o número de ciclos de *clock* e até mesmo o uso de energia necessária para comportar todo o ciclo do protocolo.

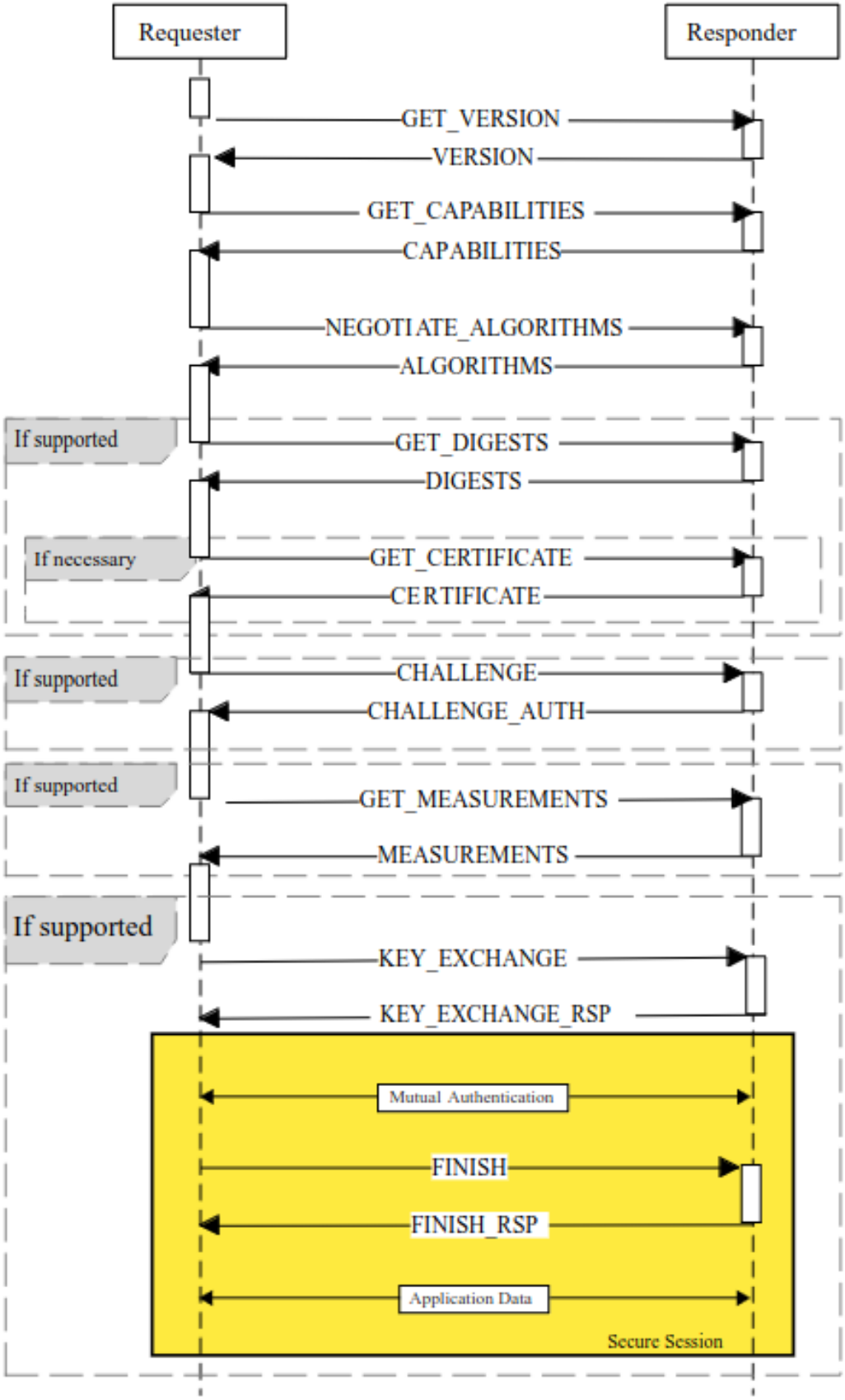

Figura 37 – Possível estrutura de um trabalho futuro

Referências

ALVES, R. C.; ALBERTINI, B. C.; SIMPLICIO, M. A. Securing hard drives with the security protocol and data model (spdm). In: IEEE. *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. [S.l.], 2022. p. 446–447. Citado 2 vezes nas páginas 13 e 14.

ASANOVIć, K. et al. *The Rocket Chip Generator*. [S.l.], 2016. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>. Citado na página 17.

BERE, G. et al. Blockchain-based firmware security check and recovery for smart inverters. In: *2021 IEEE Applied Power Electronics Conference and Exposition (APEC)*. [S.l.: s.n.], 2021. p. 675–679. Citado na página 15.

BUILDROOT-DOCS. 2020. Disponível em: <https://buildroot.org/downloads/manual/manual.html#customize>. Citado na página 19.

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 na página 15.

COMPILER RT. 2023. Disponível em: <https://github.com/llvm-mirror/compiler-rt>. Citado na página 52.

CUI, A.; COSTELLO, M.; STOLFO, S. When firmware modifications attack: A case study of embedded exploitation. 2013. Citado na página 15.

DTMF. *Security Protocol and Data Model (SPDM) Specification*. 2022. Disponível em: <https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.2.1.pdf>. Citado 2 vezes nas páginas 13 e 29.

KERMARREC, F. et al. *LiteX: an open-source SoC builder and library based on Migen Python DSL*. 2020. Disponível em: <https://osda.gitlab.io/19/1.1.pdf>. Citado na página 17.

LIBSPDM. 2023. Disponível em: <https://github.com/DMTF/libspdm>. Citado na página 13.

LINUX on Rocket. 2023. Disponível em: <https://github.com/litex-hub/linux-on-litex-rocket>. Citado 2 vezes nas páginas 7 e 28.

OPENSBI. 2020. Disponível em: <https://github.com/riscv-software-src/opensbi/releases/tag/v0.8>. Citado 2 vezes nas páginas 19 e 26.

QEMU. 2020. Disponível em: <https://www.qemu.org/docs/master/system/riscv/virt.html?highlight=riscv>. Citado na página 19.

RISC-V GNU Toolchain. 2023. Disponível em: <https://github.com/riscv-collab/riscv-gnu-toolchain>. Citado na página 22.

SPDM Hardware Git. 2023. Disponível em: <https://github.com/GustavsC/SPDM-Hardware-Implementation>. Citado na página 55.

SRC. 2023. Disponível em: <https://github.com/openbsd/src>. Citado na página 52.

THE -march, -mabi, and -mtune arguments to RISC-V Compilers. 2017. Disponível em: <https://www.sifive.com/blog/all-aboard-part-1-compiler-args>. Citado na página 21.

THU, M. M. et al. Bus electrocardiogram: Vulnerability of soc-fpga internal axi bus to electromagnetic side-channel analysis. In: *2023 International Symposium on Electromagnetic Compatibility – EMC Europe*. [S.l.: s.n.], 2023. p. 1–6. Citado na página 13.

WANG, X. et al. Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. [S.l.: s.n.], 2015. p. 544–551. Citado na página 15.