PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
ENGENHARIA DE COMPUTAÇÃO

AMBIENTE SEGURO PARA TROCA DE INFORMAÇÕES ENTRE PROCESSADORES EM SISTEMAS EMBARCADOS BASEADOS NA ARQUITETURA ARM

RAFAEL SCHILD REUSCH

Monografia apresentada como requisito parcial à obtenção do grau de Engenheiro de Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Fernando Gehm Moraes

Porto Alegre
2020

AMBIENTE SEGURO PARA TROCA DE INFORMAÇÕES ENTRE PROCESSADORES EM SISTEMAS EMBARCADOS BASEADOS NA ARQUITETURA ARM

RESUMO

Sistemas embarcados estão presentes em praticamente todos os dispositivos de nosso dia-a-dia, não limitando-se a computadores e telefones celulares. Temos dispositivos inteligentes em nossas casas, na nossa cidade e nos serviços prestados pelo governo. Com o crescente aumento da utilização de sistemas embarcados, há uma crescente necessidade de investimentos na área de segurança. Dentre os processadores utilizados em sistemas embarcados destacam-se os produzidos pela empresa ARM, os quais utilizam a extensão de segurança denominada *TrustZone*, habilitando a proteção de segmentos de memória, permitindo a implementação de um ambiente seguro de execução no processador. Nesse contexto, esse TCC compreende o estudo e implementação de uma plataforma cliente-servidor com suporte a segurança. Para atender a este objetivo implementa-se mecanismos de autenticação e comunicação criptografada. A implementação de uma plataforma com um servidor e dois clientes, utilizando as ferramentas de modelagem OVP, permitiu validar os mecanismos de regiões seguras em memória em processadores ARM, o processo de autenticação e a troca de mensagens criptografadas.

Palavras-Chave: ARM, TrustZone, Segurança, Criptografia, Autenticação.

PROPOSAL FOR A SECURE ENVIRONMENT FOR DATA EXCHANGE BETWEEN ARM-BASED PROCESSORS IN EMBEDDED SYSTEMS

ABSTRACT

Embedded systems are present in virtually every device in our lives, not limited to computers and cell phones. We have smart devices in our homes, in our city, and in government services. With the increasing use of embedded systems, there is an increasing need for investments in the security area. Among the processors used in embedded systems, those produced by ARM stand out, which use the security extension called textit TrustZone, enabling the protection of memory segments, allowing the implementation of a secure execution environment in the processor. In this context, this End-of-Term work comprises the study and implementation of a client-server platform with security support. To meet this goal, authentication and encrypted communication mechanisms are implemented. The implementation of a platform with a server and two clients, using OVP modeling tools, allowed validating the *TrustZone* mechanisms, the authentication process, and the exchange of encrypted messages.

Keywords: ARM, TrustZone, Security, Encryption, Authentication.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AMBA – *Advanced Microcontroller Bus Architecture*

API – *Application Programming Interface*

APB – *Advanced Peripheral Bus*

AXI – *Advanced eXtensible Interface*

CPU – *Central Processing Unit*

GIC – *Generic Interrupt Controller*

IP – *Intellectual Property*

OS – *Operating System*

OVP – *Open Virtual Plataforms*

RAM – *Random-Access Memory*

RISC – *Reduced Instruction Set Computer*

SOC – *System on Chip*

TCC – *Trabalho de Conclusão de Curso*

TZASC – *TrustZone Address Space Controller*

TZMA – *TrustZone Memory Address*

TZPC – *TrustZone Protection Controller*

DMA – *Direct Memory Access*

ARM – *Advanced RISC Machine*

AES – *Advanced Encryption Standard*

OVP – *Open Virtual Plataform*

1. INTRODUÇÃO

Sistemas embarcados estão presentes em praticamente todos os dispositivos de nosso dia-a-dia, não limitando-se a computadores e telefones celulares. Temos dispositivos inteligentes em nossas casas, na nossa cidade e nos serviços prestados pelo governo. Com a tendência da interconexão entre todos estes dispositivos (e.g., IoT, *smart cities*, *smart government*, indústria 4.0), o crescimento do número destes equipamentos é inevitável. A Figura 1.1 ilustra a tendência do número de dispositivos interconectados no período 2012 a 2020 [Burhan et al., 2018].

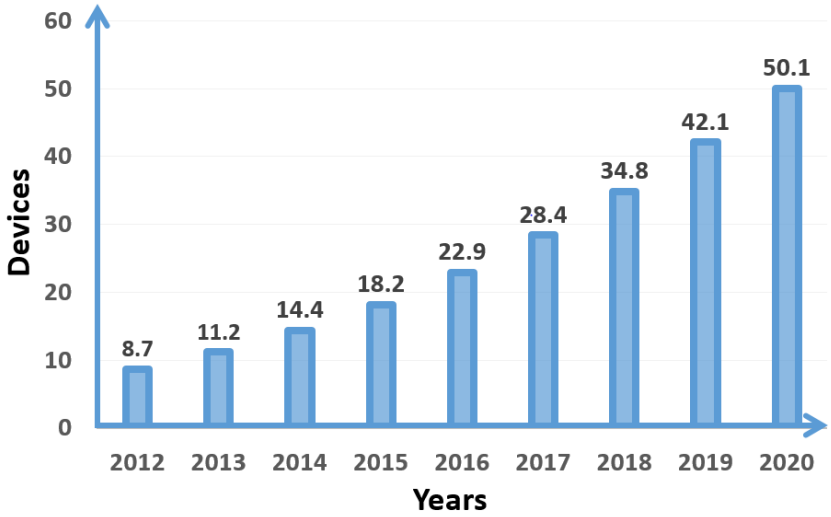


Figura 1.1: Número de dispositivos IoT interconectados, em bilhões ao longo dos anos (fonte: [Burhan et al., 2018]).

Um importante fator a considerar com o crescimento no número de dispositivos é a segurança da informação. Dispositivos embarcados podem conter informações sensíveis relacionadas a dados pessoais (e.g., operações bancárias, dados de saúde), dados corporativos e dados governamentais. Estes dados são muito atraentes para *hackers* que buscam benefícios financeiros baseados em ataques de extorsão digital, conhecidos como *ransomware*, ou também realizando roubo de informações, como por exemplo, credenciais de acesso a sistemas bancários. Com a crescente densidade de informações armazenadas em sistemas embarcados é de extrema relevância que existam barreiras de proteção em hardware e software para prevenir esses ataques.

Os processadores ARM são muito utilizados em sistemas embarcados, em particular as arquiteturas ARM Cortex-M3/M4 [Bakos, 2016]. A ARM disponibiliza para seus processadores a tecnologia de segurança no nível de hardware e software denominada *TrustZone* [Ngabonziza et al., 2016]. Essa tecnologia permite compartimentalizar a memória em duas regiões, a segura e a insegura. Isso permite que os processadores processem dados sensíveis, os quais encontram-se armazenados na região segura. Hélio Fuques

[Fuques, 2019] em seu TCC explorou a tecnologia *TrustZone*, utilizando para isto as ferramentas providas pela Imperas [IMPERAS, 2019]. Seu trabalho consistiu na modelagem de uma plataforma multiprocessada com uso de *TrustZone* para proteção de dados sensíveis, porém sem uso de autenticação e criptografia.

A Figura 1.2 ilustra a plataforma multiprocessada proposta em [Fuques, 2019]. A parte esquerda da figura contém processadores ditos "inseguros", os quais são denominados *clientes* neste trabalho, e a parte direita da figura o processador "seguro", denominado *servidor* neste trabalho. A interconexão entre estes ambientes é realizada pelo periférico *NonSecToSec*. Este ambiente representa, por exemplo, nodos de uma rede IoT (processadores "inseguros") que comunicam-se com um processador "seguro", o qual pré-processa dados antes de enviá-los para uma nuvem. Este procedimento é denominado de *edge computing*, o qual é um paradigma de computação distribuída que aproxima a computação e o armazenamento de dados do local onde são necessários, para melhorar os tempos de resposta e economizar largura de banda [Satyanarayanan, 2017].

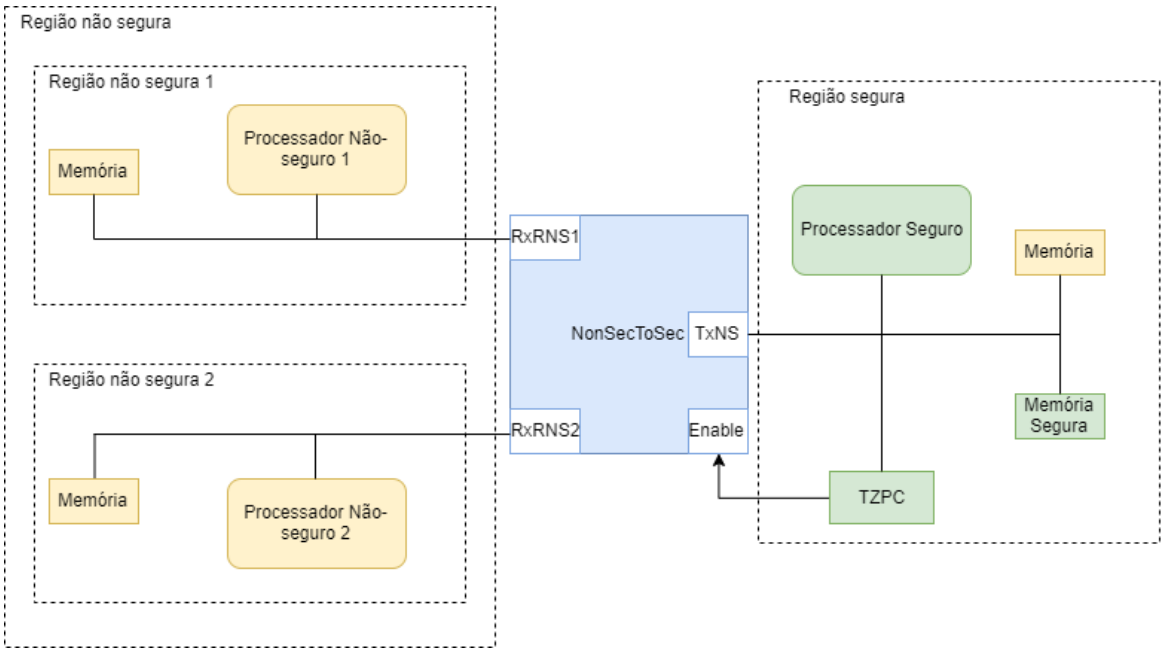


Figura 1.2: Implementação da plataforma modelada com OVP [Fuques, 2019].

1.1 Motivação

A motivação deste trabalho é dupla. O primeiro fator de motivação é a utilização dos conhecimentos adquiridos ao longo do curso de Engenharia de Computação na área de arquitetura de computadores e programação de periféricos, assim como realizar estudos na área de segurança da informação e criptografia, temas pouco abordados durante a graduação.

O segundo fator de motivação é a importância da ubiquidade dos sistemas embarcados em nosso dia-a-dia, e a necessidade de possuir conhecimentos, tanto no nível de hardware quanto do nível de software, para melhor projetar estes sistemas posteriormente na vida profissional.

1.2 Objetivos

O objetivo estratégico deste TCC é prover segurança à um sistema, não apenas no armazenamento da informação no processador servidor, mas também proveja segurança na troca de informação entre os clientes e o servidor, de forma bidirecional.

Os objetivos específicos incluem (utilizamos abaixo a terminologia “seguro” e “não seguro” como sinônimos de servidor e cliente, respectivamente)

1. Hélio Fuques [Fuques, 2019] desenvolveu uma plataforma que permitia a comunicação segura entre um processador ARM “seguro” e processadores ARM “não seguros”, através do periférico *NonSecToSec*. O primeiro objetivo específico compreende tornar a comunicação do periférico *NonSecToSec* bidirecional. Hoje, este periférico habilita a comunicação no sentido ambiente "não seguro" para "seguro". O desafio nesta etapa encontra-se em dominar os mecanismos de exceção do processador, assim como no domínio dos mecanismos de privilégios que os processadores ARM utilizam.
2. Utilizar um mecanismo de autenticação baseado no protocolo Diffie-Hellman Curva Elíptica [Paar and Pelzl, 2010] de tal forma que cada processador "não seguro" tenha uma chave de sessão única com o processador "seguro".
3. Utilizar o mecanismo de criptografia AES [Paar and Pelzl, 2010] para a troca de informações entre um determinado processador da região "não segura" com o processador "seguro", utilizando a chave de sessão.

O atendimento a este conjunto de objetivos específicos visa proteger o sistema multiprocessado de ataques que possuam como intenção o roubo de informações sensíveis.

1.3 Organização do documento

Este TCC está organizado como segue. O Capítulo 2 introduz conceitos básicos relacionados à autenticação e criptografia. O Capítulo 3 apresenta a arquitetura do processador ARM, os aspectos de hardware e software da extensão de segurança *TrustZone*, assim como detalha o projeto de referência [Fuques, 2019]. O Capítulo 4 apresenta a arquitetura desenvolvida. O Capítulo 5 apresenta a validação e os resultados obtidos.

2. SEGURANÇA DA INFORMAÇÃO

Este capítulo apresenta conceitos básicos relacionados à segurança da informação, assim como a algoritmos de criptografia e autenticação. Estes conceitos são apresentados, dada a necessidade dos mesmos no trabalho desenvolvido.

2.1 Criptografia

Acredita-se que os conceitos de criptografia [Davies, 1997] originaram-se junto com a escrita. Em conjunto com as emergentes disputas de poder notou-se a necessidade de estabelecer uma comunicação segura entre um grupo seleto de pessoas. Pode-se dizer que antigamente a criptografia era uma forma de encriptação, sendo basicamente a conversão de uma mensagem legível para uma mensagem aparentemente ilegível para os destinatários não desejáveis.

A criptografia moderna possui como um dos seus principais pilares a matemática. Conforme o avanço da matemática, novos métodos de criptografia e autenticação foram sendo criados [Davies, 1997]. O princípio da criptografia de chave pública foi apresentado pela primeira vez por Martin Hellman, Ralph Merkle e Whitfield Diffie na Universidade de Stanford em 1976. Em 1978, Ron Rivest, Adi Shamir e Leonard Adleman publicaram o sistema criptográfico de chave pública RSA (Rivest – Shamir – Adleman), sendo amplamente utilizado para transmissão segura de dados até os dias de hoje [Rivest et al., 1978].

2.1.1 Conceitos Básicos

Desde sua divulgação pública, o algoritmo RSA tornou-se um dos métodos de chave pública mais utilizado na área de criptografia assimétrica. Utilizando conceitos matemáticos o algoritmo realiza a criação de duas chaves, a pública e a privada. Essas chaves são utilizadas principalmente na criptografia de pequenos volumes de dados, por exemplo, na comunicação para geração de outras chaves, assim como em certificados digitais.

O objetivo desse método não é substituir cifradores simétricos, pois o algoritmo é diversas vezes mais lento que um AES (*Advanced Encryption Standard)*, por exemplo. Por causa dessa limitação, ambos AES e RSA são muito utilizados em conjunto, onde a responsabilidade da criptografia de chave pública é participar da troca de chaves para obtenção de uma chave de sessão, simétrica. Uma vez estabelecida esta chave de sessão, a comunicação é feita utilizando cifradores simétricos, de maior eficiência em termos de computação.

2.1.2 AES

O AES é um padrão que define o algoritmo de criptografia simétrica mais utilizado nos dias de hoje, empregado em diversas aplicações como *Wi-Fi*, protocolos de rede segura, *secure shell* (SSH) e também em serviços de telefonia via internet como Skype. Este padrão foi criado a partir da constatação da insegurança do algoritmo DES (*Data Encryption Standard*) (e 3DES) [Paar and Pelzl, 2010]. A seleção deste novo padrão de criptografia AES foi feita pelo Instituto Estadunidense de Padrões e Tecnologia, NIST. [Daemen and Rijmen, 2000]

O algoritmo AES foi considerado pelo NIST como muito eficiente, permitindo a utilização de chaves de até 256 *bits* [NIST, 2001]. Dada sua segurança, o governo dos Estados Unidos anunciou que a NSA iria utilizá-lo para criptografia de documentos confidenciais. Anteriormente, documentos confidenciais dos EUA só utilizavam métodos de criptografia secretos, nunca um algoritmo disponível ao público.

O AES utiliza uma metodologia semelhante ao DES, baseado em rodadas (*rounds*). Diferentemente do DES, o AES não utiliza redes de Feistel, já que esta não permite que todo bloco de dados seja criptografado em uma única rodada. O resultado é um número reduzido de rodadas necessárias para que todo processo de criptografia seja realizado. Nesse padrão, são utilizadas chaves de 128, 192 ou 256 *bits* com tamanho do bloco de dados fixo em 128 *bits*. A correlação entre tamanho da chave e o número de rodadas necessárias é apresentada na Tabela 2.1.

Tabela 2.1: Tabela relacionando o número de rodadas com o tamanho da chave providenciada.

Tamanho da chave	Número de rodadas
128 *bits*	10
192 *bits*	12
256 *bits*	14

O AES opera sobre um arranjo bidimensional de *bytes* com 4x4 posições (128 bits), denominado de *estado*. Para criptografar, cada rodada do AES (exceto a última) consiste em quatro estágios, conforme apresentado na Figura 2.1 (Fonte: [Paar and Pelzl, 2010]).

- **Adição de chaves** (*Key Addition Layer*) – cada *byte* do estado é combinado com a subchave própria da rodada *(RoundKey)*, sendo cada subchave derivada da chave principal usando um algoritmo de escalonamento de chaves.
- **Substituição de bytes (S-Box)** (*Byte Substitution Layer*). É uma etapa de substituição não linear, onde cada *byte* é substituído por outro, de acordo com uma tabela de referência.

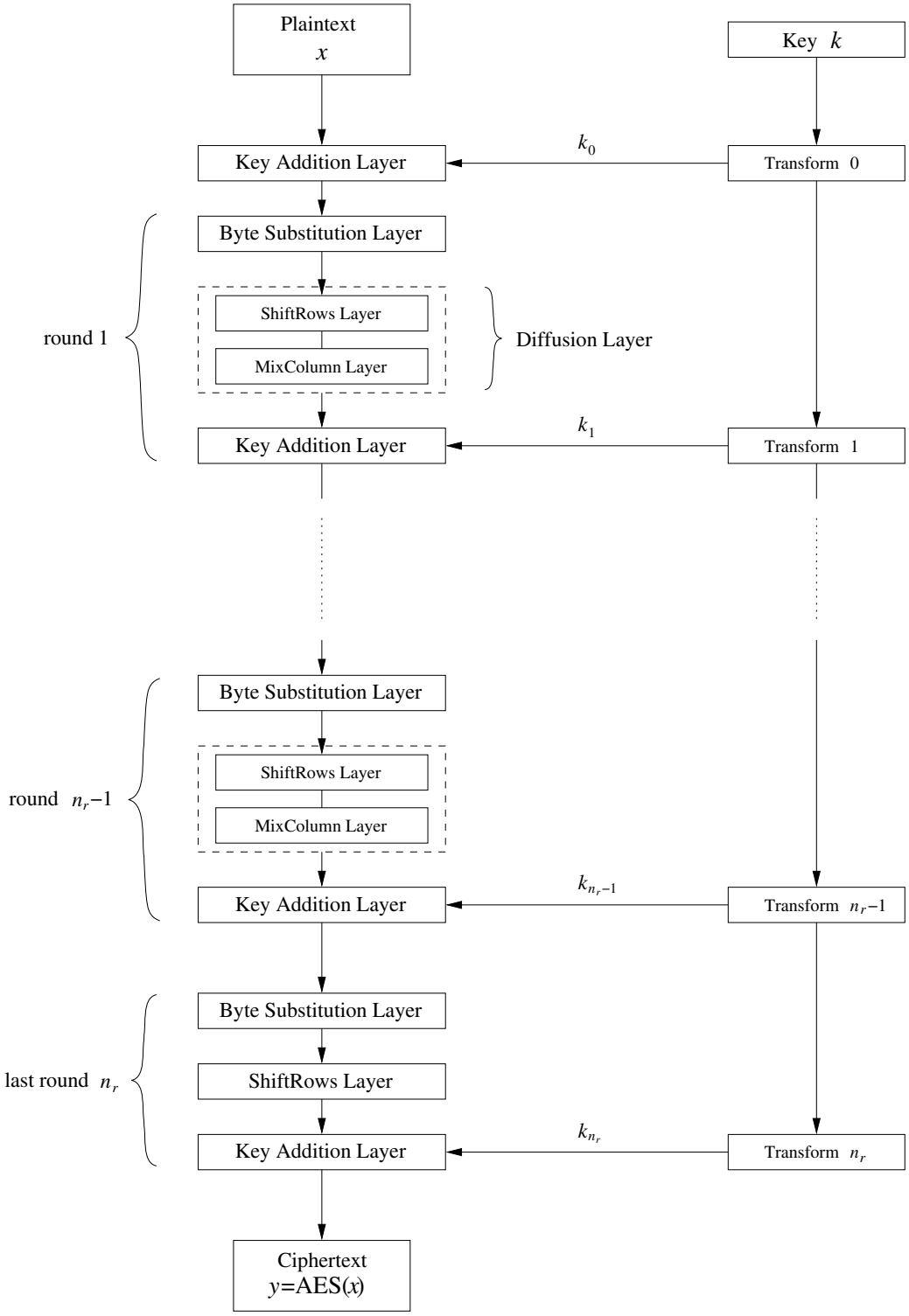


Figura 2.1: Estrutura dos blocos presentes na implementação do AES.

- **Camada de difusão** *(Diffusion Layer*. Nessa etapa, existem duas sub-camadas nas quais implementam operações lineares realizando diversas trocas na posição dos *bytes* dentro da matriz.

    1. **Camada de mudança de linhas** (*ShiftRows layer*). A etapa de *ShiftRows* opera sobre as linhas do estado, deslocando os *bytes* em cada linha de um determinado

número de posições. A primeira linha fica inalterada. Cada *byte* da segunda linha é deslocado à esquerda de uma posição. Similarmente, a terceira e quarta fileiras são deslocadas de duas e de três posições respectivamente.

2. **Camada de mistura de colunas** (*MixColumns layer*). Na etapa de *MixColumns*, os quatro *bytes* de cada coluna do estado são combinados usando uma transformação linear inversível.

A Figura 2.2 foi obtida do livro *Understanding Cryptography* e exemplifica simplificadamente as operações feitas nas camadas mencionadas anteriormente.

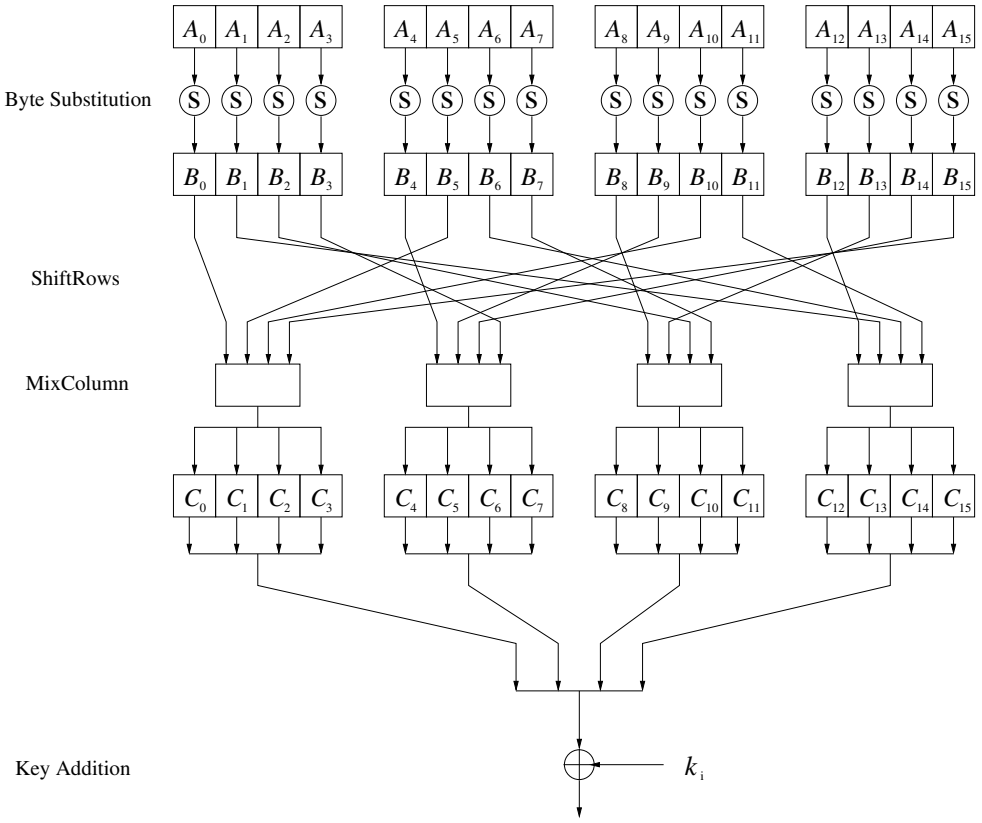


Figura 2.2: Estágios em um *round* do AES. (Fonte:[Paar and Pelzl, 2010]).

Após a realização de todas as rodadas necessárias, o dado de entrada está criptografado, e somente com a chave *k*, utilizada para realizar a encriptação, o dado poderá ser decriptografado. Até o momento não existem ataques melhores que os de força bruta conhecidos contra o AES.

2.2 Diffie-Hellman Key Exchange utilizando Curvas Elípticas

O protocolo de trocas de chaves *Diffie-Hellman Key Exchange* foi proposto em 1976 e solucionou um grande problema da criptografia simétrica que era à distribuição

de chaves, permitindo que dois dispositivos chegassem em uma chave secreta comum, a chave de sessão, comunicando-se em um canal não-seguro [Diffie and Hellman, 1976]. Ao contrário do nome do algoritmo, em nenhum momento a chave é realmente trocada. Ambas as partes realizam operações pré-definidas utilizando variáveis públicas e privadas, e assim chegam em uma chave de sessão. Esse protocolo é essencial para utilização do AES, já que este não possui nenhuma diretiva padrão relacionada a distribuição de chaves.

O protocolo pode ser feito com um pequeno número de mensagens, resultando em um bom desempenho. A troca de chaves proposta por Diffie-Hellman consiste de duas etapas:

- **Configuração** - Fase onde parâmetros públicos iniciais são decididos, utilizados na próxima fase para obtenção da chave de sessão;
- **Troca de chaves** - Nessa fase, ambas partes trocam entre si resultados de computações realizadas localmente. Em um segundo momento, cada uma das partes deve realizar uma operação com o resultado recebido da outra parte. O resultado final dessas operações é a chave de sessão.

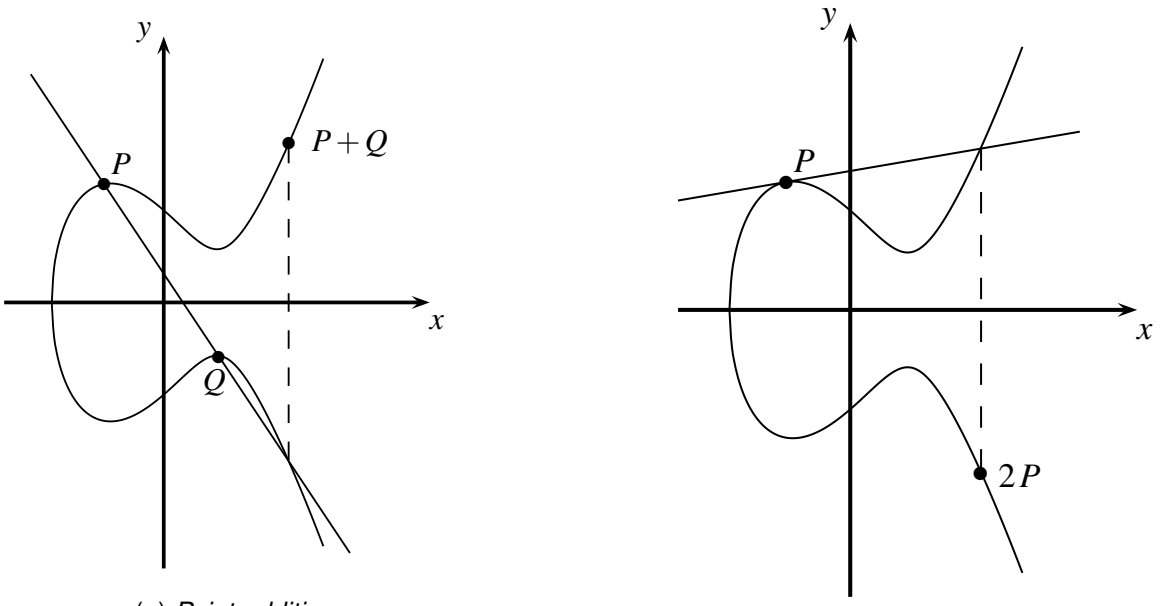
O DHKE (*Diffie–Hellman key exchange*) convencional utiliza como base funções exponenciais utilizando números primos. Essas operações são unidirecionais, ou seja, com o resultado delas não é possível descobrir quais foram os operandos utilizados. Neste trabalho é utilizado uma variação desse protocolo, no qual utiliza-se curvas elípticas para obtenção da chave de sessão. A utilização desse tipo deste tipo de curva permite um nível de segurança semelhante ao RSA, porém, utilizando chaves de menor tamanho, e melhor desempenho computacional. Neste trabalho, este método será utilizado em conjunto com o protocolo DHKE.

A escolha da curva elíptica a ser utilizada não é um processo simples, pois a segurança do processo de criptografia é dependente desta curva. Hoje, existem diversas curvas que são consideradas criptograficamente fortes, sendo padronizadas pelo NIST.

A Figura 2.3 ilustra duas operações sobre a curva $y^2 = x^3 - 3x + 3$, com números reais [Paar and Pelzl, 2010]. Estas operações são: soma (*point addition*) e *point doubling*. Estas operações são a base para o *point multiplication*, operação utilizada no pelo ECDH (*Elliptic-curve Diffie–Hellman*).

Os passos para realização do ECDH são muito semelhantes ao DH convencional. O primeiro passo a realizar é a escolha de um número primo, uma curva elíptica e uma coordenada presente na curva. A etapa de troca de chaves acontece da mesma forma que o método convencional. A Figura 2.4 ilustra esse processo.

Utilizando suas respectivas chaves privadas, que nesse caso serão números inteiros grandes, são calculados utilizando a técnica de *Point Multiplication*, suas respectivas chaves públicas, que no caso são coordenadas nessa curva. Após a troca desses parâ-



(a) *Point addition.*

(b) Point doubling.

Figura 2.3: Operações sobre curvas elípticas [Paar and Pelzl, 2010].

Elliptic Curve Diffie–Hellman Key Exchange (ECDH)

Alice		**Bob**
choose $k_{prA} = a \in \{2,3,\ldots,\#E-1\}$		choose $k_{prB} = b \in \{2,3,\ldots,\#E-1\}$
compute $k_{pubA} = aP = A = (x_A, y_A)$		compute $k_{pubB} = bP = B = (x_B, y_B)$
	$\xrightarrow{A}$	
	$\xleftarrow{B}$	
compute $aB = T_{AB}$		compute $bA = T_{AB}$

Joint secret between Alice and Bob: $T_{AB} = (x_{AB}, y_{AB})$.

Figura 2.4: Troca de chaves Diffie-Hellman utilizando curvas elípticas [Paar and Pelzl, 2010].

metros públicos entre ambas as partes, representado pelas letras *A* e *B* na figura 2.4, é executado novamente a técnica *Point Multiplication* utilizando dessa vez os parâmetros recebidos junto com sua respectiva chave privada. O resultado dessa operação é a chave de sessão que será utilizada pelo AES.

3. ARQUITETURA ARM COM SUPORTE A TRUSTZONE

A arquitetura ARM *(Advanced RISC Machine)*, como a própria sigla menciona, é uma arquitetura RISC *(Reduced Instruction Set Computer)*, a qual utiliza um conjunto de instruções reduzido e simples, permitindo que os processadores operem em frequências mais altas que arquiteturas CISC *(Complex Instruction Set Computing)* [Bakos, 2016].

Os processadores ARM são muito utilizados em *smarphones*, *tablets*, assim como em dispositivos *wearables* como *smartwatches*. Devido ao seu conjunto de instruções reduzidas, o número de transistores é também reduzido, permitindo reduzir o tamanho do *die*, facilitando a utilização em dispositivos miniaturizados.

Comumente, as arquiteturas RISC possuem três grupos principais de instruções: (*i*) controle; (*ii*) acesso à memória (*load*/*store*); (*iii*) aritméticas.

3.1 TrustZone

A tecnologia TrustZone é uma extensão opcional de hardware disponível nas arquiteturas ARM. O aspecto de segurança do Trustzone é baseado na ideia de dividir o SOC *(System on Chip)* em duas regiões, a “segura” e a “normal” (também referenciada com “insegura”). A região segura contém as aplicações nas quais o processador opera em modo seguro, e a região normal as aplicações sem restrições de segurança. Barreiras de *hardware* estão presentes para impedir que aplicações que estão sendo executadas na região normal tenham acesso a itens sensíveis da região segura, como partições protegidas de memória.

3.1.1 Arquitetura de Hardware

O ARM possui diversos componentes físicos de segurança responsáveis por impor as restrições de segurança necessárias para preservar a divisão das regiões. A lista abaixo apresenta os periféricos necessários ao suporte de TrustZone [Ngabonziza et al., 2016].

- **Advanced eXtensible Interface Bus (AXI)**. É o barramento de comunicação principal do processador ARM. Esse barramento utiliza um *bit* chamado de NS *(Non Secure)*, o qual indica se a operação subsequente será feita no modo seguro ou inseguro. Esse *bit* é definido com o endereço a ser acessado.
- **Advanced Peripheral Bus (APB)**. Este barramento é responsável pela comunicação com periféricos de menor desempenho. Com o objetivo de permitir a compatibili-

dade com dispositivos de terceiros, nenhum mecanismo de segurança é implementado neste barramento.

- **AXI to APB Bridge**. Este componente é responsável por realizar a ponte entre os dois barramentos, AXI e APB. Considerando que o APB não realiza verificações de segurança, esta ponte é responsável por controlar as permissões e negar requisições não autorizadas provenientes do barramento APB.

- **Cache Controller**. Esse componente observa o *bit* NS, presente na décima terceira posição dos endereços. Considerando que existirão dois endereços de memória idênticos, uma no mundo seguro e a outra no inseguro, será de responsabilidade do *Cache Controller* analisar esse *bit* e decidir se a requisição tem como alvo o mundo seguro ou o não seguro.

- **Direct Memory Access (DMA)**. Sua principal responsabilidade é transferir dados da memória física, evitando a sobrecarga no processador quando há volumes de dados maiores a serem enviados/recebidos. Esse componente utiliza o barramento AXI e suporta o tratamento de eventos seguros e inseguros simultaneamente. Ele também previne que memórias seguras sejam acessadas através de requisições não seguras.

- **TrustZone Address Space Controller (TZASC)**. Esse componente possibilita uma classificação das memórias conectadas ao barramento AXI. Ele permite particionar uma única unidade de memória, não precisando existir unidades dedicadas para cada uma das regiões.

- **TrustZone Memory Adapter (TZMA)**. Semelhante ao TZASC porém ele realiza o particionamento de memórias estáticas *on-chip*, como uma ROM ou SRAM. O TZMA permite que uma única memória de até 2 MB seja particionada em duas regiões, onde a parte inferior é segura e a parte superior não segura.

- **Generic Interrupt Controller (GIC)**. Esse controlador é muito semelhante ao componente *Cache Controller*, ele é responsável por tratar interrupções seguras e não seguras. A partir da análise das permissões de acesso, o controlador impossibilita o acesso ao processador seguro através de interrupções não seguras.

- **Trustzone Protection Controller (TZPC)**. Esse é um controlador que é utilizado para permitir que periféricos que estão localizados no APB possam transitar entre as regiões seguras e inseguras. No ARM o número máximo de periféricos simultâneos sendo controlados pelo TZPC é 8.

A *bridge* possui um sinal de entrada, TZPCDECPROT, oriundo de cada periférico localizado no barramento. Este sinal é usado para determinar se o periférico está configurado como seguro ou não seguro. A *bridge* rejeitará transações não seguras para intervalos

de endereços de periféricos seguros. Esses sinais de entrada podem ter valor fixo determinado em tempo de projeto, ou podem ser controlados dinamicamente por meio de um periférico confiável, como o *TrustZone Protection Controller* (TZPC), para permitir a alteração dinâmica do estado de segurança em tempo de execução.

A Figura 3.1 mostra uma *bridge* AXI-para-APB controlando 4 periféricos. O TZPC está configurado como seguro, os temporizadores (*Timers*) e relógio de tempo real (RTC) como não seguros, e a interface de teclado e mouse (KMI) tem sua configuração de segurança programável [ARM, 2010]. O *software* executando em modo seguro pode programar o TZPC para alternar o KMI de seguro para não seguro, e vice-versa.

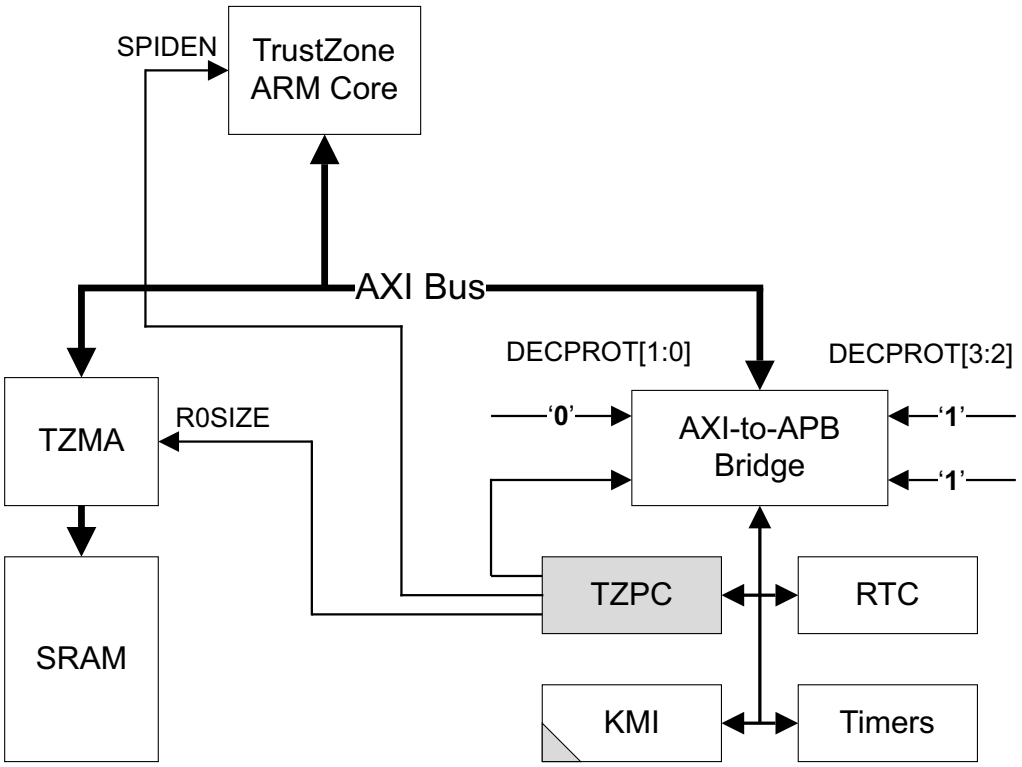
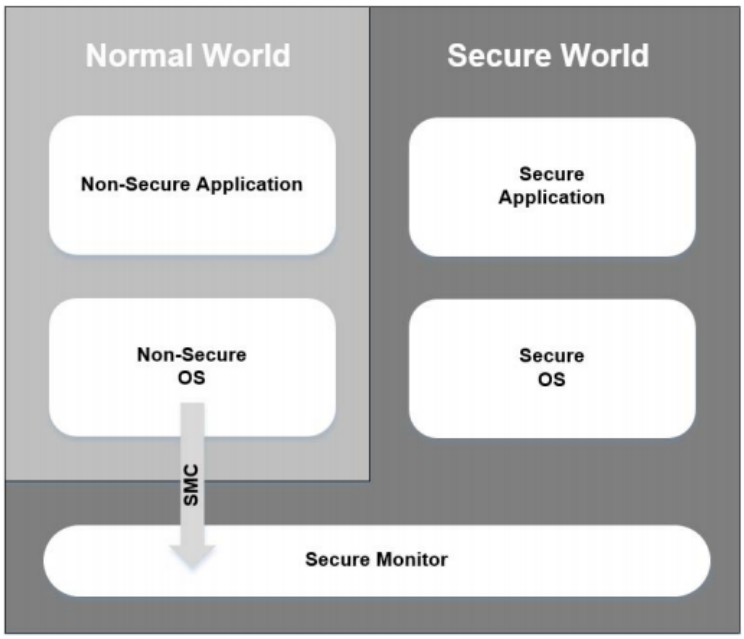


Figura 3.1: Arquitetura ilustrando diversos componentes para o funcionamento da Trustzone na arquitetura ARM (Fonte: [ARM, 2010])

3.1.2 Arquitetura de Software

O artigo *TrustZone Explained: Architectural Features and Use Cases* detalha uma arquitetura de *software* que permite a execução de sistemas operacionais dedicados para cada uma das regiões seguras do TrustZone [Ngabonziza et al., 2016]. Essa é uma solução robusta, pois permite a execução de diversas aplicações simultaneamente. A comunicação entre essas regiões (*worlds*) pode ser feita através da utilização de um modo monitor, que funciona semelhante a uma função de troca de contexto. A região não segura só pode utilizar esse modo caso uma exceção seja gerada por uma das três possibilidades: (*i*)

interrupção de *hardware*; (*ii*) sinal de abortar externo; (*iii*) instrução SMC (*Secure Monitor Call*). A Figura 3.2 ilustra essa arquitetura.



(Fonte: [Ngabonziza et al., 2016])

Figura 3.2: Arquitetura de *software* da TrustZone em um Cortex-A.

3.1.3 Exceções

A arquitetura ARM utiliza uma *vector table*, apresentada na Tabela 3.1. Nessa tabela cada exceção é relacionada com um endereço indicando sua função de tratamento. Quando uma exceção ocorre o *program counter* é movido para o endereço de tratamento da exceção ocorrida [ARM, 2011].

Tabela 3.1: *Vector table* e sua descrição [ARM, 2019].

PC Offset	Exceção	Descrição
0x04	*Undefined Instruction*	Ocorre caso o processador tenta executar uma instrução não definida.
0x08	*SMC / SVC Handler*	É executado quando uma instrução SMC ou SVC é executada pelo processador.
0x10	*Data Abort Handler*	Acontece quando ocorre algum erro no manuseio de dados, como por exemplo endereço de leitura incorreto.
0x18	*IRQ Handler*	Funciona como um *callback* caso algum periférico deseja notificar o processador.
0x1C	*FIQ Handler*	Função chama quando o periférico tenta notificar o processador com prioridade superior

3.1.4 Alterando o modo de segurança do processador ARM

Para alterar o modo de segurança do processador entre seguro e não seguro, é necessário realizar uma escrita no registrador NS do SCR (*Secure Configuration Register*). Essa operação só pode ser realizada caso o processador esteja em modo seguro. Para isso é necessário que uma exceção seja gerada através da instrução SMC (*Secure Monitor Call*) para que o processador entre em modo seguro de forma temporária. Durante a execução do tratamento da exceção previamente gerada, é possível escrever no registrador NS, realizando a troca do modo de segurança do processador.

O artigo *SMC Calling Convention* [ARM, 2016] sugere utilizar registradores como argumentos da instrução SMC. O Código 3.1 é um exemplo da função que realiza o tratamento da exceção gerada pelo SMC. Nesse código, o *handler* irá alterar o modo de segurança do processador baseado no argumento *r2* [Fuques, 2019].

Código 3.1: Código do tratamento de uma SMC [Fuques, 2019].

```
1  register r2 asm("r2");
2  void SMC_Handler_Main( )
3  {
4      int smc_number = r2;
5      printf("NS - Entered in Super Monitor Call Handler!!! Operation is %d\n", smc_number);
6      switch (smc_number)
7      {
8      case TO_SECURE:
9          printf("Writing 0x0 to ns bit\n");
10         WR_SCR(0);
11         break;
12     case TO_UNSECURE:
13         printf("Writing 1 to ns bit\n");
14         WR_SCR(1);
15         break;
16     default:
17         asm(
18             "MSR SPSR_cxsf, #0x13\n"
19             "LDR PC, =run\n" // SVC_entry points to the first
20         );
21         break;
22     }
23 }
24
25 int main()
```

```
26 {
27 ...
28 // enter in non-secure mode
29 asm("mov r2, #2\n"
30     "SMC #0\n")
31 ...
32 }
```

3.2 Projeto de Referência

Helio Fuques [Fuques, 2019] em seu TCC explorou a tecnologia TrustZone em uma arquitetura ARM. Em seu trabalho ele propôs a implementação de um periférico atuando como ponte de comunicação entre duas regiões, não segura e segura, apresentado na Figura 3.3.

A plataforma proposta por Fuques é composta por dois processadores Cortex-A8 na região não segura, representando sistemas suscetíveis a ataques externos, almejando armazenar dados sigilosos em uma memória segura. Outro processador Cortex-A8 é posicionado na região segura e tem acesso direto a uma memória segura. O periférico providencia a conexão entre essas duas regiões. O componente TZPC, previamente detalhado, é responsável por realizar o controle de acessos ao periférico *NonSecToSec*.

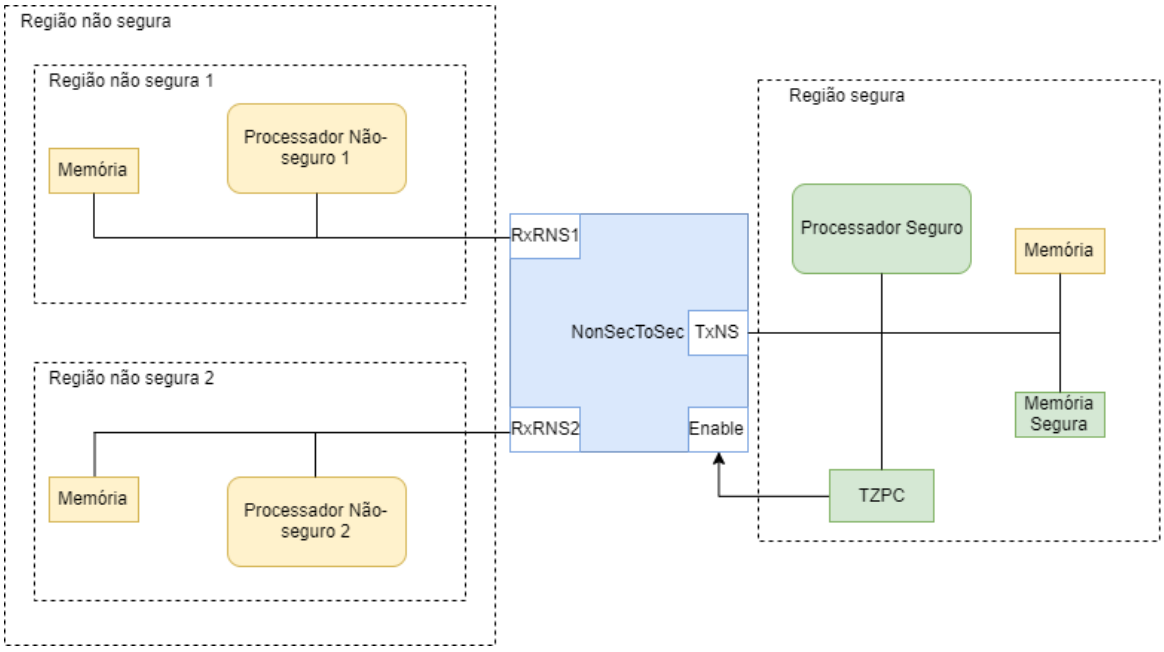


Figura 3.3: Implementação da plataforma multiprocessada modelada com OVP (fonte: [Fuques, 2019]).

3.2.1 Periférico NonSecToSec

A principal função do periférico *NonSecToSec* é administrar o envio de mensagens de uma região de processadores não seguros para um processador seguro unidirecionalmente. No evento de um envio, o periférico deve gerar uma interrupção no processador da região segura. O processador executa a interrupção, e determina se deve armazenar a informação em uma memória segura, a qual só poderá ser acessada pelo modo seguro do processador. No final do tratamento dessa interrupção o processador deverá realizar a recuperação de contexto para retomar as atividades que estavam sendo feitas.

Na implementação feita por Fuques [Fuques, 2019], o periférico possui três barramentos para transmissão de dados. Dois barramentos dedicados para comunicação entre periférico e os respectivos processadores não-seguros. Um barramento dedicado para comunicação com o processador seguro. Também está presente uma porta *enable*. Este sinal é operado pelo periférico TZPC, informando para o periférico *NonSecToSec* que o processador seguro está pronto para ler os dados do barramento. A Tabela 3.2 detalha as portas presentes no módulo.

Tabela 3.2: Portas e Barramentos do Periférico NonSecToSec (Fonte: [Fuques, 2019]).

Nome	Tipo	Descrição
enable	netport de entrada	Porta de 1 *bit* que habilita a transferência de mensagens da região não segura para a região segura.
interruptRequest	netport de saída	Porta de 1 *bit* que fica em nível lógico alto quando uma nova mensagem está disponível para ser enviada para a região segura.
rxNonSecurePort1	barramento 32 *bits*	Esse barramento é responsável por receber as mensagens de um processador não seguro. Ele possui registradores para ser requisitada a escrita, avisar a permissão da escrita e transferir dados
rxNonSecurePort2	barramento 32 *bits*	Barramento com o mesmo propósito do rxNonSecurePort1.
readSecurePort	barramento 32 *bits*	Barramento responsável por transferir as mensagens para o processador seguro. Ele possui registradores para implementar o protocolo de comunicação e transferir os dados.

3.2.2 Região não-segura

Para realizar o envio de mensagens para a região segura através do periférico *NonSecToSec*, cada um dos processadores presentes na região não segura recebeu funções que habilitaram essa comunicação. As funções criadas foram (*i*) *requireToSend* e (*ii*)

sendMessage. A função *requireToSend* realiza uma escrita no registrador RG_TX_REQ, notificando o periférico que um processador deseja enviar um dado a ele. Após o periférico responder uma resposta positiva, a função *sendMessage* escreve no barramento de 32 *bits* *rxNonSecurePort* a mensagem a ser enviada. O Código 3.2 implementa essas funções [Fuques, 2019].

Código 3.2: Funções utilizadas para habilitar o envio de dados para região segura.

```
1  // @brief: Require permission to peripheral to send a message.
2  // Blocking method.
3  void requireToSend()
4  {
5      printf("RNS: Requiring write permission\n");
6      (*(char*) RG_TX_REQ) = 1;
7      while(*RG_TX_REQ_ACK != 1);
8      printf("RNS: Ready to send!\n");
9  }
10
11 // @brief: Send given message filling the header with parameter sender and size.
12 void sendMessage(int size, int sender,char* data)
13 {
14     *RG_TX_WRITE_HEADER = sender;
15
16     *RG_TX_WRITE_HEADER = size;
17
18     int i = 0;
19     for(i = 0; i < size; i++ )
20     {
21        *RG_TX_WRITE_DATA = data[i];
22     }
23     printf("RNS: Sent a message!\n");
24 }
```

3.2.3 Região segura

Para receber as mensagens provenientes dos processadores não seguros, o *software* presente no processador deve: *(i)* habilitar interrupções; *(ii)* transicionar entre modo seguro e não seguro; *(iii)* mapear a memória ao periférico TZPC; *(iv)* preencher a *vector table*. O processador seguro possui três principais modos de operação:

- *boot* - Realiza a configuração inicial do processador. Mapeia a *Translation Lookaside Table* (TLB) e as instruções da *vector table*.
- *run* - Simula a operação comum de um processador, através da execução do cálculo da sequência de Fibonacci (pode ser qualquer aplicação).
- *interruption handler* - Implementa o tratamento da interrupção gerada pelo periférico *NonSecToSec*. O diagrama na Figura 3.4 ilustra o passo a passo do tratamento de uma interrupção.

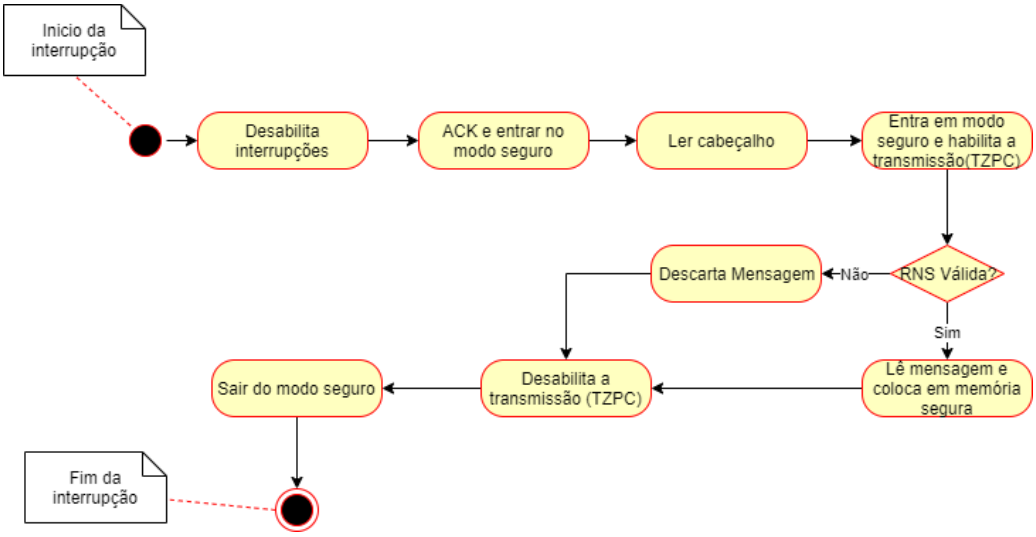


Figura 3.4: Diagrama de blocos ilustrando o tratamento de uma interrupção utilizando TrustZone. (Fonte: [Fuques, 2019]).

3.2.4 Execução do Projeto de Referência

Nesta execução, cada um dos processadores da região não segura tenta enviar duas mensagens para o processador seguro.

Observa-se na primeira etapa da Figura 3.5 o processo de envio de uma mensagem do processador não seguro número 2 para o periférico *NonSecToSec*. Na segunda etapa ilustrada na figura, é possível visualizar o armazenamento da mensagem no periférico assim como uma nova tentativa de escrita proveniente do mesmo processador, a qual foi negada pois o periférico ainda não repassou os dados da transmissão anterior para o processador seguro. Na terceira etapa, pode ser visto que o processador seguro recebeu a interrupção causada pela mensagem presente na porta de saída do periférico.

A Figura 3.6 ilustra o processo de tratamento da interrupção feita pelo processador seguro. Na etapa número 4, é feita a escrita no *bit* NS, através do *Secure Monitor Call* (SMC), realizando a troca do modo de segurança. Agora no modo seguro, na etapa 5, o

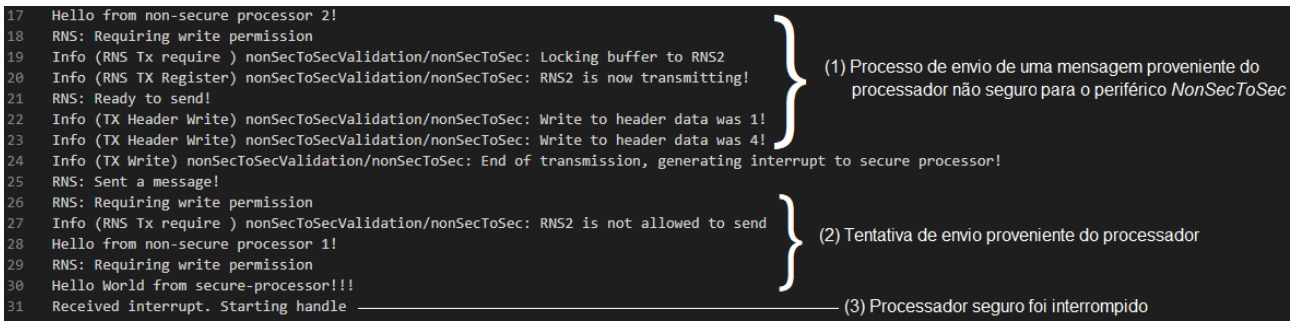


Figura 3.5: Comunicação do processador não seguro com o periférico (fonte: Autor).

processador realiza a leitura da mensagem proveniente do periférico, concluindo o processo na etapa 6. Na sétima e última etapa, o processador seguro termina a interrupção e realiza a recuperação de contexto.

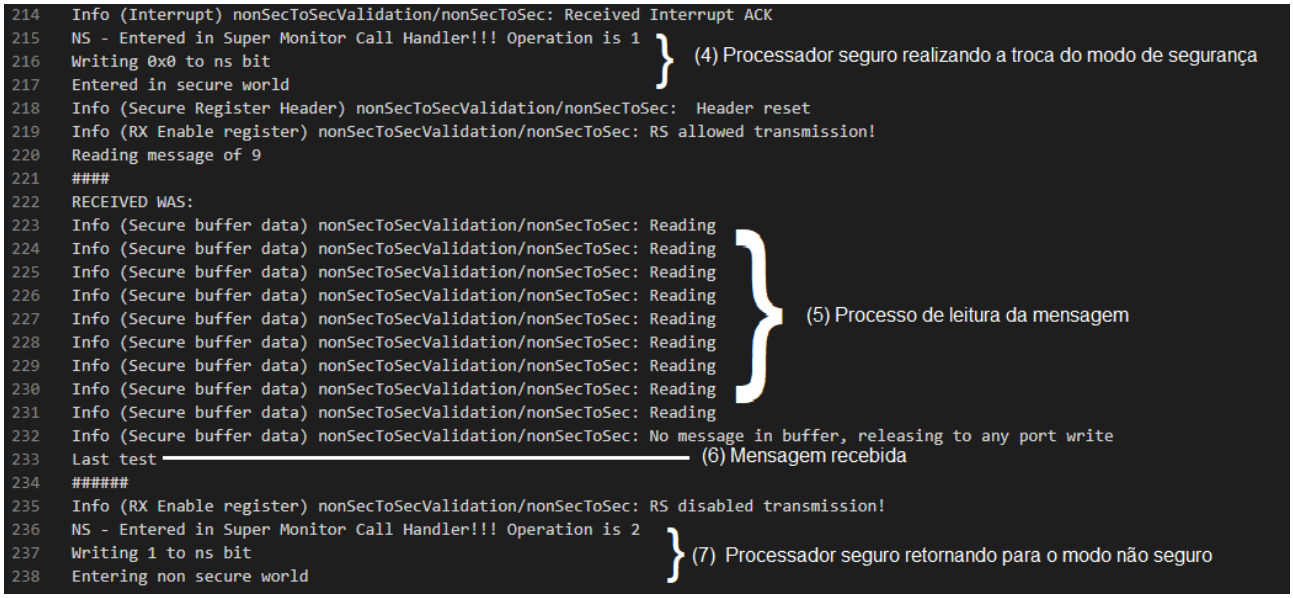


Figura 3.6: Tratamento da interrupção pelo processador seguro (fonte: Autor).

Por fim, a Figura 3.7 ilustra duas operações de leitura na memória segura. A primeira tentativa, com o processador no modo seguro, se concretizou corretamente, mostrando o caractere lido. Na segunda tentativa, com o processador no modo não seguro, a tentativa de leitura foi negada, gerando uma exceção.

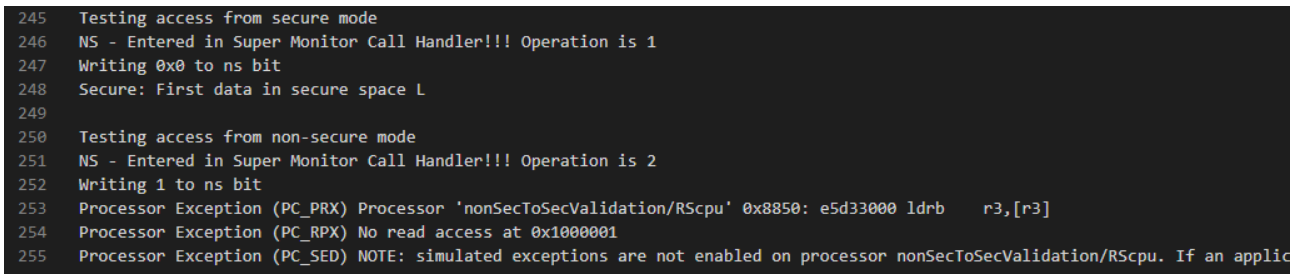


Figura 3.7: Teste de acesso à memória segura (fonte: Autor).

4. ARQUITETURA CLIENTE-SERVIDOR COM SUPORTE A SEGURANÇA

Este capítulo detalha a implementação da arquitetura cliente-servidor, com suporte à segurança. A Seção 4.1 detalha a plataforma desenvolvida. A Seção 4.2 apresenta o periférico que permite a comunicação bidirecional entre servidor e clientes. A Seção 4.3 detalha as camadas de software responsáveis pela autenticação e criptografia. A Seção 4.4 conclui este capítulo apresentado as aplicações que estão executando nos clientes e no servidor.

4.1 Plataforma de Hardware

Esta seção apresenta o desenvolvimento da plataforma que permite a troca de mensagens entre dois clientes e um servidor, utilizando zonas seguras, autenticação e criptografia.

A plataforma foi criada utilizando o conjunto de ferramentas de modelagem e simulação providas pela Imperas [IMPERAS, 2019]. A descrição do hardware da plataforma é feita através de um arquivo TCL que contém instâncias de todos os módulos. A plataforma desenvolvida contém dois processadores Cortex-A8 atuando como clientes, cada um possuindo acesso a um barramento que permite conexão direta à sua memória. No lado do servidor, existe outro processador Cortex-A8 acompanhado de duas memórias e o módulo TZPC *(TrustZone Platform Controller)*. A Figura 4.1 apresenta a plataforma.

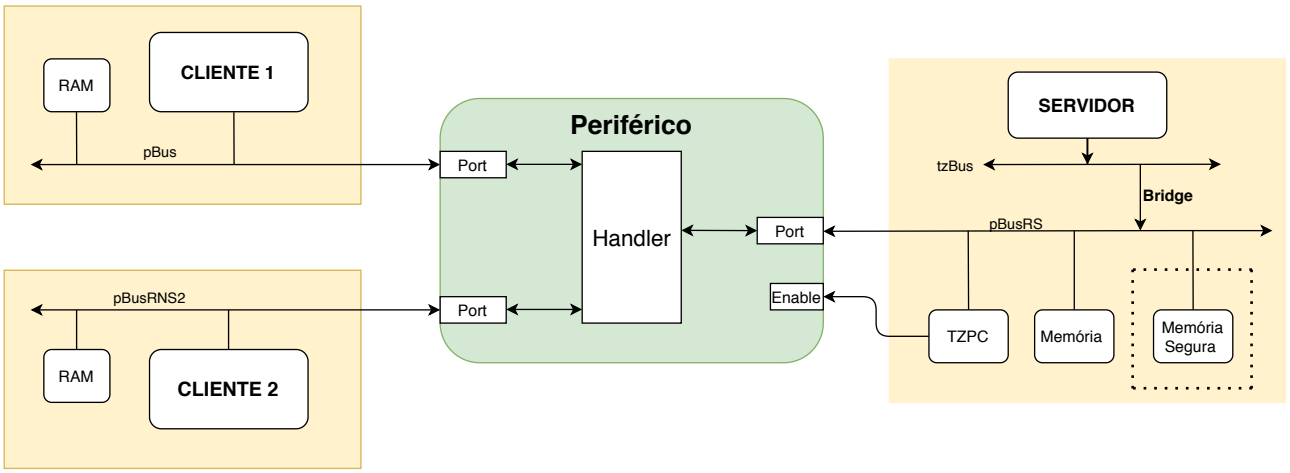


Figura 4.1: Implementação da plataforma multiprocessada modelada no presente TCC.

O periférico originalmente denominado *NonSecToSec* denomina-se no presente trabalho *secBridge*. Neste ambiente, o periférico *secBridge* é responsável por realizar a co-

municação entre clientes e servidor de forma bidirecional, aplicando algoritmos de controle de fluxo, garantindo que a comunicação ocorra de forma correta.

O trecho de Código 4.1 apresenta a descrição do servidor, realizando a instância do processador ARM e de duas memórias.

Código 4.1: TCL utilizado pelo iGen modelando a arquitetura ARM presente no servidor.

```
1  ihwaddprocessor -instancename RScpu -type arm -vendor arm.ovpworld.org
2      -endian little  -semihostname armNewlib -semihostvendor arm.ovpworld.org
3
4  ihwsetparameter   -handle  RScpu
5                    -name variant
6                    -value Cortex-A8
7                    -type enum
8
9  ihwconnect       -instancename RScpu -busmasterport INSTRUCTION -bus tzBus
10 ihwconnect       -instancename RScpu -busmasterport DATA        -bus tzBus
11 ihwconnect       -instancename RScpu -netport irq -net irq_
12
13 ihwaddmemory -instancename ramRS -type ram
14 ihwconnect   -instancename ramRS -busslaveport sp1 -bus pBusRS
15              -loaddress 0x00000000 -hiaddress 0x001fffff
16
17 ihwaddmemory -instancename secureRam -type ram
18 ihwconnect   -instancename secureRam -busslaveport sp1 -bus pBusRS
19              -loaddress 0x01000000 -hiaddress 0x01ffffff
```

O processador utilizado no servidor é descrito nas linhas 1 e 4 utilizando os comandos *iwaddprocessor* e *ihwsetparameter*, configurado através dos seguintes parâmetros:

- *instancename*: nome dado a instância do processador – *RScpu*;
- *type*: tipo do processador
- *vendor*: fornecedor do processador;
- *endian*: tipo de ordenação utilizado pelo processador;
- *semihostname*: biblioteca no qual o processador está inserido;
- *semihostvendor*: fornecedor da biblioteca;
- *value*: modelo do processador;

- *variant*: versão do processador.

Na plataforma do servidor foram criadas duas memórias com intervalos de endereços distintos, uma memória ”comum” e outra ”segura” . A descrição desta memória é feita da linha 13 até a 18, utilizando os comandos *ihwaddmemory* e *ihwconnect*. Os parâmetros *loaddress* e *hiaddress* são utilizados para definir o intervalo de endereços disponíveis para cada memória. Neste ambiente, a memória segura não compartilha nenhum endereço com a memória comum.

Na linha 11 é descrito uma porta no processador nomeada *irq*, que é específica para o recebimento de um aviso de interrupção. Este sinal é escrito pelo periférico *secBridge* quando este possui uma mensagem a ser enviado para o servidor. Com esse aviso o servidor inicia o tratamento da interrupção realizando a troca para o modo seguro para ler os dados disponíveis na porta do periférico.

Como pode ser visto nas linhas 9 e 10 do Código 4.1, o processador Cortex-A8 não está presente no mesmo barramento que seus periféricos – *tzBuz*. Isso é feito para adicionar outra camada de segurança contra acessos não autorizados a endereços de memória protegidos. A conexão com o barramento externo é feita através de diversas *bridges*. Um exemplo de duas *bridges* é apresentado no Código 4.2.

Código 4.2: Descrição de duas bridges de acesso a memória

```
1  ihwaddbridge -instancename bNSRS
2
3  ihwconnect   -instancename bNSRS -busslaveport  sp -bus tzBus \
4               -loaddress 0x10011001000 -hiaddress 0x10011001002
5  ihwconnect   -instancename bNSRS -busmasterport mp -bus pBusRS \
6               -loaddress 0x11001000 -hiaddress 0x11001002
7
8  ihwaddbridge -instancename bRSNS
9
10 ihwconnect   -instancename bRSNS -busslaveport  sp -bus tzBus \
11              -loaddress 0x10011002000 -hiaddress 0x10011002007
12 ihwconnect   -instancename bRSNS -busmasterport mp -bus pBusRS \
13              -loaddress 0x11002000 -hiaddress 0x11002007
```

A *bridge bNSRS*, criada na linha 1, é responsável por realizar a conexão de dois barramentos, *tzBus* e *pBusRS*. Sua função é limitar o acesso do servidor, enquanto trabalhando no modo não seguro, de uma forma que ele consiga apenas ler os primeiros *bytes* deste endereço, que nesse caso representam o cabeçalho da mensagem. Essa política de segurança exige que o processador altere seu modo de operação para o modo seguro antes de realizar a leitura do *payload* da mensagem.

A segunda *bridge* – *bRSNS*, descrita na linha 8, é muito semelhante à primeira, realizando a conexão do barramento *tzBus* ao *pBusRS*, porém nessa última, o intervalo de endereços de memória disponíveis são diferentes. A *bridge bRSNS* diferentemente da *bridge* descrita anteriormente, disponibiliza para o servidor a possibilidade desse enviar dados para os clientes, alocando uma região de memória dedicada para isso.

Para acessos à memória segura, outra *bridge* foi criada. O Código 4.3 apresenta esta *bridge*.

Código 4.3: Descrição de uma *bridge* para acesso à memória segura.

```
1 ihwaddbridge -instancename secure2
2 ihwconnect   -instancename secure2 -busslaveport  sp -bus tzBus \
3              -loaddress 0x00000000 -hiaddress 0xffffffff
4 ihwconnect   -instancename secure2 -busmasterport mp -bus pBusRS  \
5              -loaddress 0x00000000 -hiaddress 0xffffffff
```

A *bridge secure2* realiza o oposto das *bridges* explicadas anteriormente. Ela concede acesso total a todos endereços de memória quando o processador está trabalhando em modo seguro. Dessa forma, todas operações de leitura e escrita na memória segura utilizam essa *bridge*.

4.2 Periférico secBridge

O periférico inicialmente proposto por Fuques [Fuques, 2019] foi modificado com a objetivo de prover comunicação bidirecional entre clientes e servidor. Para isso, modificações foram feitas nas máquinas de estado do periférico.

Como a plataforma criada neste trabalho é composta por dois clientes se comunicando com um único servidor, espera-se que ocorra disputa na comunicação entre os clientes e o servidor. Por esse motivo, na região entre o servidor e o periférico foi adicionado uma segunda porta de comunicação, permitindo uma comunicação *full-duplex*. Já na região entre cada cliente e o periférico, a comunicação é *half-duplex*, ou seja, o cliente não pode transmitir e receber ao mesmo tempo. Para isso foi adicionado um registrador *sendtoRNS*, que permite o periférico controlar o modo de transmissão de cada clientes. A Figura 4.2 ilustra essa topologia.

As portas do periférico são efetivamente um conjunto de registradores mapeados em memória, ou seja, cada porta possui um intervalo de endereços fixos. Dessa forma outros dispositivos com acesso a essa memória podem realizar operações de leitura e escrita nestes registradores. A descrição de uma porta e um registrador é mostrada no Código 4.4.

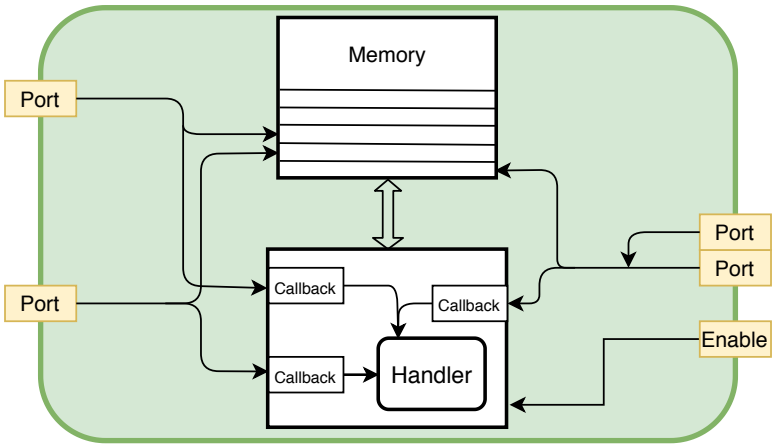


Figura 4.2: Diagrama de blocos da implementação do periférico *secBridge*.

Código 4.4: Declaração de uma porta e um registrador mapeado em memória.

```
1  imodeladdbusslaveport -name rxNonSecurePort1
2
3  imodeladdaddressblock -name regs    \
4                        -port readSecurePort \
5                        -offset 0x0   \
6                        -width 8      \
7                        -size 128
8
9  imodeladdmmregister  -name sendtoRNS1 \
10                      -addressblock rxNonSecurePort1/rxRegs \
11                      -readfunction  dataReadyRNS1 \
12                      -offset 0x5
```

O comando *-imodeladdbusslaveport* é utilizado para declaração de uma porta. Utilizando o comando *imodeladdressblock* é possível atribuir a porta a um bloco de memória, para posteriormente serem declarados registradores. Esse comando aceita parâmetros como *width* e *size* para especificar a quantidade de endereços disponíveis para cada posição de memória. No periférico implementado neste trabalho foram declarados diversos registradores mapeados em memória utilizando o comando *imodeladdmmregister*. Este comando recebe os seguintes parâmetros:

- *name*: nome dado à instância da porta;
- *addressblock*: em qual bloco de memória previamente criado o registrador é mapeado;
- *writefunction/readfunction*: nome da função chamada caso uma operação de leitura/escrita seja feita nessa porta;

- *offset*: valor do *offset* em relação ao endereço do bloco de memória;
- *access*: define as permissões que clientes conectados a esta porta possuem: *w* para escrita e *r* para leitura. No Código 4.4, este parâmetro não está presente, pois caso não seja especificado, o registrador aceita ambas operações.

Como ilustra a Figura 4.2, a combinação de portas e registradores é baseada em funções *callbacks*. No momento que alguma operação de leitura ou escrita é feita no registrador, imediatamente uma função *callback* é executada no periférico.

O Código 4.5 representa uma função *callback* de leitura. Desta forma, o valor de retorno da função é o resultado da operação de leitura do registrador. Por isso, esta função deve ser otimizada, já que a latência de leitura à memória está diretamente relacionada com o tempo de execução dessa função. O Código 4.5 apresenta a função que é chamada quando o Cliente 1 realiza uma leitura no registrador *sendtoRNS1*.

Código 4.5: *Callback* de leitura do registrador *sendtoRNS1*.

```
1 PPM_REG_READ_CB(dataReadyRNS1)
2 {
3     if (bufferRSUsed == true && RSmsgHeader.target == 0 ){
4         bhmMessage("I","RS to RNS","RNS1 is ready to read message");
5         return 1;
6     } else {
7         return 0;
8     }
9 }
```

Essa função retorna o valor 1 quando: *(i)* o periférico possui uma mensagem para enviar para algum cliente; *(ii)* o destinatário é o Cliente 1. A implementação destas *callbacks* não podem ser genéricas pois a plataforma exige que cada registrador tenha uma *callback* dedicado a ele. Dessa forma, existe uma outra função praticamente idêntica a essa no código, porém dedicada a leitura do registrador *dataReadyRNS2*, que é lida pelo Cliente 2.

Depois da declaração das portas e escrita dos *callbacks*, o periférico deve ser inserido na plataforma e conectado aos respectivos barramentos para comunicação com outras entidades. O Código 4.6 mostra a inserção do periférico na plataforma.

Código 4.6: Declaração do periférico *secBridge* na plataforma.

```
1 ihwaddperipheral -instancename secBridge -type nonSecToSec -modelfile peripheral/pse
2
3 ihwconnect       -instancename secBridge -netport enable -net tzpcdecprot0_0
```

```
4  ihwconnect       -instancename secBridge -netport newMessageAvailable -net irq_
5
6  ihwsetparameter    -handle       nonSesecBridgecToSec -name portSize       \
7        -value   0x1000 -type uns64
8
9  ihwconnect      -instancename secBridge -busslaveport readSecurePort -bus pBusRS \
10                 -loaddress 0x11001000 -hiaddress 0x1100107F
11
12 ihwconnect      -instancename secBridge -busslaveport writeSecurePort -bus pBusRS \
13                 -loaddress 0x11002000 -hiaddress 0x1100207F
14
15 ihwconnect      -instancename secBridge -busslaveport rxNonSecurePort1 \
16                  -bus pBus -loaddress 0x11001000 -hiaddress 0x1100107f
17
18 ihwconnect      -instancename secBridge -busslaveport rxNonSecurePort2 \
19                  -bus pBusRNS2 -loaddress 0x11001000 -hiaddress 0x1100107f
```

Utilizando o comando *ihwaddperipheral*, o periférico *secBridge* é instanciado na plataforma criada neste trabalho. Em seguida, o comando *ihwconnect* é utilizado com o objetivo de realizar conexões físicas entre portas e barramentos. É possível verificar da linha 9 até a 19 as conexões das quatro portas de comunicação principais aos respectivos barramentos. Os parâmetros dessa função são:

- *instancename*: nome dado a instância na qual a porta (*busslaveport*) está presente;
- *busslaveport*: nome da porta declarada na descrição de portas do periférico;
- *bus*: indica qual barramento a porta deverá se conectar;
- *loaddress/hiaddress*: intervalo de endereços de memória que protegem o periférico de escritas/leituras em área de memória não disponíveis para o mundo externo.

4.3 Software de suporte para prover segurança à plataforma

Esta Seção apresenta a camada de *software* responsável por agregar segurança à plataforma. O presente trabalho utilizou três diferentes bibliotecas:

- **tweetNaCl**: implementação da criptografia de curvas elípticas em conjunto com o protocolo de troca de chaves Diffie-Hellman [Bernstein et al., 2014].
- **SHA256.c** Biblioteca *open source* baseado no algoritmo de *hash* SHA-2 [Conte, 2012].

- **AES.c** Biblioteca *open source* que realiza a implementação completa de dois tipos de AES, EBC e EBD [Kokke, 2019] .

A Figura 4.3 apresenta a forma na qual estas bibliotecas trabalham em conjunto.

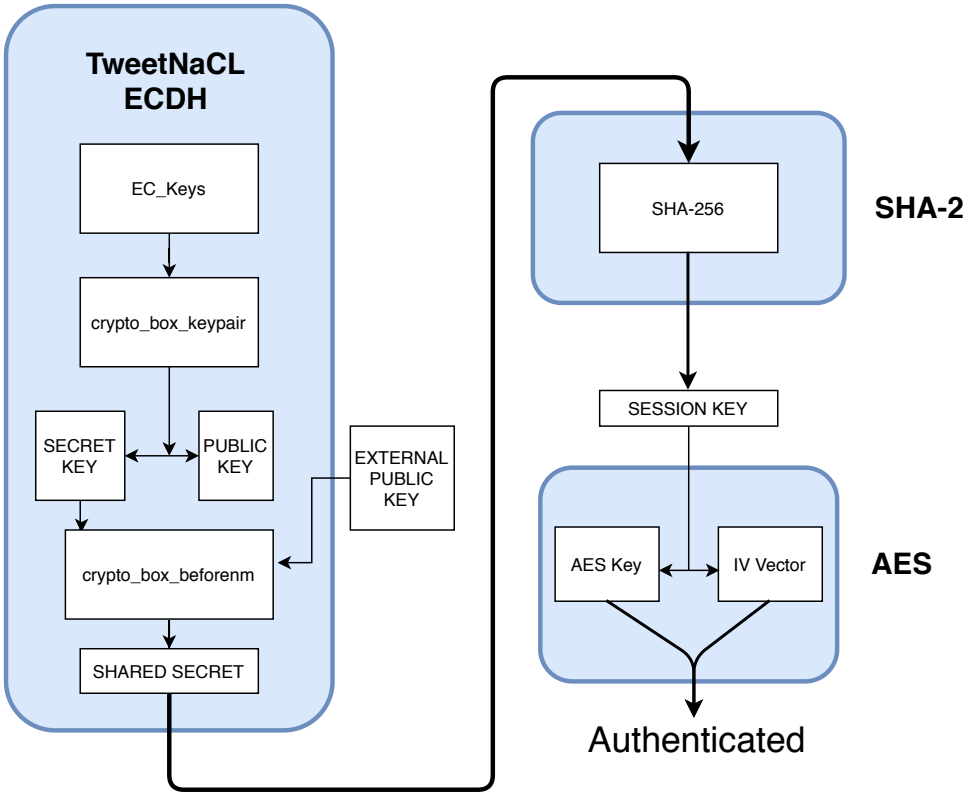


Figura 4.3: Diagrama de blocos ilustrando a obtenção da chave simétrica de sessão para a execução da proteção das mensagens por criptografia.

A primeira etapa, após a inicialização do sistema (*boot*), corresponde à geração de dois números aleatórios, que sejam criptograficamente seguros. Em seguida, a função *crypto_box_keypair*, da biblioteca *TweetNaCl*, é utilizada para gerar as chaves privada (*secret key*) e pública (*public key*), através do uso de curvas elípticas. Na terceira etapa, utilizando a chave privada em conjunto com a chave pública da outra parte, é realizado o processamento do segredo compartilhado, através da função *crypto_box_beforenm*. Em seguida, a chave de sessão é derivada do segredo compartilhado utilizando uma biblioteca de *hash*, nesse caso, SHA-256. Por último, a chave de sessão de 256 *bits* é dividida em duas partes. A parte mais significativa é utilizada como chave do AES e a parte menos significativa é utilizada como vetor de inicialização no algoritmo do AES. [Paar and Pelzl, 2010]

Como a plataforma contém dois clientes comunicando-se com um único servidor, esse terá que armazenar ambas as chaves em memória. Dessa forma, antes de qualquer operação de criptografia o servidor deve verificar qual é o cliente que ele está se comunicando, utilizando a chave de acordo. Para isso, a memória segura foi mapeada em duas

áreas, como mostra o Código 4.7.

Código 4.7: Áreas de memória para armazenamento de dados seguros.

```
1 #define SECURE_MEMORY_REGION ((volatile unsigned char *) (0x01000000))
2 #define SECURE_KEY_REGION ((volatile unsigned char *) (0x01000100))
```

A primeira área de memória, *SECURE_MEMORY_REGION* é dedicada para o armazenamento temporário de mensagens recebidas dos clientes. A segunda área *SECURE_KEY_REGION* é exclusiva para armazenamento das chaves de sessão provenientes do Diffie-Hellman. Foram criadas duas funções que realizam operações nesta memória, *memoryKeyWrite* e *memoryKeyRead*. As funções estão descritas no Código 4.8.

Código 4.8: Funções criadas para realizar leitura e escrita das chaves na memória segura.

```
1  void memoryKeyWrite(int keyid){
2      int memoffset = keyid * PUB_KEY_LEN;
3
4      ENTER_SECURE_USING_MONITOR();
5
6      memcpy(SECURE_KEY_REGION+memoffset,sessionKey, PUB_KEY_LEN);
7
8      ENTER_NON_SECURE_USING_MONITOR();
9  }
10 void memoryKeyRead(int keyid){
11
12     int memoffset = keyid * PUB_KEY_LEN ;
13     int offset;
14     ENTER_SECURE_USING_MONITOR();
15
16     for(offset = 0; offset < PUB_KEY_LEN ; offset++)
17        keyFromMemory[offset] = *(SECURE_KEY_REGION+memoffset+offset);
18
19    ENTER_NON_SECURE_USING_MONITOR();
20    printf("\n");
21
22 }
```

Ambas funções são semelhantes. No início das funções, nas linhas 4 e 14, é realizado a troca do modo de operação do processador, utilizando a função *ENTER_SECURE_USING_MONITOR()*. Agora no modo seguro, o processador tem acesso à memória segura,

e nas linhas 6 e 17, realizar a operação requisitada. Por fim, o processador volta ao modo anterior utilizando a função *ENTER_NON_SECURE_USING_MONITOR()*.

4.3.1 Protocolo ECDH

Na implementação do protocolo de trocas de chaves Diffie-Hellmann utilizando curvas elípticas optou-se pela utilização da biblioteca *TweetNaCl* pela documentação disponível e simplicidade de utilização. Esta é uma biblioteca criptográfica *open source*, que utiliza algoritmos padronizados e curvas que são conhecidas e criptograficamente seguras [Bernstein, 2006]. Ao mesmo tempo, esse pacote criptográfico é compacto e providencia abstrações em etapas que são complexas, como o pacote de funções *crypto_box*.

Para a implementação do protocolo Diffie-Hellman, a biblioteca utiliza a curva elíptica 25519 para o cálculo das chaves privadas e públicas. Essa é uma curva que foi projetada exclusivamente para ser utilizada em conjunto com o protocolo Diffie-Hellman. Ela é uma das curvas mais rápidas de ser processada e ao mesmo tempo considerada dentre as mais seguras conhecidas atualmente [Bernstein et al., 2014]

O presente trabalho utilizou as funções do pacote *crypto_box*. Esse pacote permite realizar desde a geração de chaves utilizando curva elípticas até realização de assinaturas digitais complexas. No entanto, as funções do pacote *crypto_box* possuem uma limitação. São funções chamadas sob demanda, ou seja, mesmo que a chave de sessão já tenha sido gerada, ao tentar criptografar um dado via um AES disponível na biblioteca, a mesma irá processar novamente as chaves, causando uma perda de eficiência no algoritmo. No ambiente deste trabalho, onde a plataforma oferece um processamento limitado, a utilização desta abstração se provou inviável.

O artigo "*TweetNaCl: A crypto library in 100 tweets*" sugere utilizar uma função intermediária interna da biblioteca com a intenção de pré-processar as chaves de sessão e manualmente armazená-las em memória, evitando-se a necessidade do processador calcular esta chave novamente [Bernstein et al., 2014]. O Código 4.9 ilustra a utilização da função *crypto_box_beforenm*.

Código 4.9: Cálculo do segredo compartilhado utilizando funções da biblioteca *TweetNaCl*.

```
1 generateSecretParameters();
2 crypto_box_keypair(EC_keys.pk, EC_keys.sk);
3 crypto_box_beforenm(sharedSecret,hisPublicKey,EC_keys.sk);
```

Esse trecho de código ilustra o passo-a-passo para a obtenção de uma chave de sessão utilizando a biblioteca TweetNacl. Inicialmente a chave secreta (privada) deve

ser gerada por uma fonte de números aleatórios criptograficamente segura. Com a chave secreta utilizamos a função *crypto_box_keypair* para obter a chave pública através de uma derivação da sua chave privada, utilizando uma curva elíptica. Com esses dados, utilizamos a função intermediaria *crypto_box_beforenm* para obter um segredo compartilhado, no qual futuramente será derivada a chave de sessão [Bernstein et al., 2014]. A Figura 4.4 ilustra o passo a passo deste procedimento.

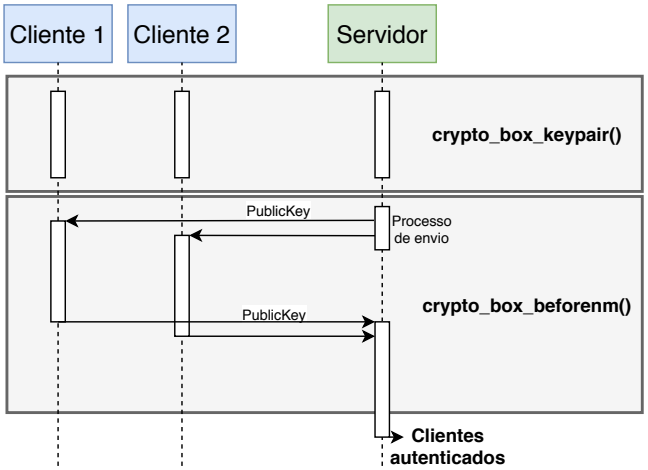


Figura 4.4: Diagrama sequencial demonstrando a autenticação de clientes na plataforma criada.

4.3.2 SHA-256

O SHA-256 é parte do conjunto de funções *hash* criptográficas SHA-2. Esse conjunto foi inicialmente projetado pela NSA (Agência de Segurança Nacional dos EUA). [NIST, 2012] Funções *hash* são um conjunto de operações matemáticas aplicadas a um dado com a intenção de adicionar confusão. Elas possuem três características que se destacam:

- São unidirecionais. É inviável descobrir a mensagem inicial a partir da cifra. A única forma de ataque conhecida é a força bruta.
- São determinísticas. A mesma mensagem sempre resultará no mesmo *hash*.
- Efeito avalanche: Uma pequena modificação na mensagem, resulta em um *hash* totalmente diferente. Essa é uma característica muito desejável por algoritmos criptográficos.

A Figura 4.5 ilustra a execução do *hash* SHA-256 sobre duas *strings* diferentes, porém o tamanho da saída do SHA-256 é sempre fixa em 256, isso é um resultado de um *padding* feito internamente por essa operação.

Figura 4.5: Exemplo funcional da função de *hash* SHA-256.

Nesse trabalho, foi utilizado a função *hash* SHA-256 da biblioteca *cryto-algoritms*, proposta por Brad Conte [Conte, 2012]. Na arquitetura implementada esta função é utilizada em conjunto com o protocolo ECDH. O livro *Understanding Cryptography* sugere derivar a chave de sessão a partir do segredo compartilhado através da utilização de uma função SHA-256.

4.3.3 AES

Motivado pela limitação do pacote criptográfico *tweetNaCl*, esse trabalho utiliza a biblioteca *Tiny-AES* como algoritmo de criptografia simétrica [Kokke, 2019]. Esse pacote de funções criptográficas simétrica realiza uma implementação simples, porém robusta para três modos de operação do AES:

- ECB (Electronic Code Book) - é o mais simples de todos. Devido a suas deficiências, como a incapacidade de ocultar padrões de dado, geralmente não é utilizado.
- CTR (Counter) - cada bloco de texto claro é *XORed* com um contador criptografado. O contador é incrementado para cada bloco subsequente.
- CBC (Cypher Block Chaining) - a cada bloco de texto simples é aplicada uma função *XOR* junto com o bloco cifrado anterior antes do texto ser criptografado. Desta forma, cada bloco cifrado fica dependente de todos os blocos de texto simples processados até este momento. Além disso, para que cada mensagem seja única, mesmo de um mesmo texto original, um vector de inicialização único deve ser utilizado no primeiro bloco, preferencialmente gerado aleatoriamente.

A escolha dessa biblioteca foi feita pela simplicidade e segurança oferecidas, além de consumir apenas 200 *bytes* de RAM. Uma limitação desta biblioteca é que ela não possui função *padding*, dessa forma, apenas blocos de dados múltiplos de 16 *bytes* são aceitos como entrada. Nesse trabalho a implementação projetada força o bloco de dados a sempre respeitar essa regra, não exigindo nenhum tipo de ajuste extra.

O modo CBC foi escolhido por sua facilidade de implementação quando comparado ao CTR, e sua maior segurança quando comparado ao ECB. Como mencionando, o modo CDC requer um IV (*Initialization Vector*). Esse vetor é uma entrada de tamanho fixo que necessita ser um número criptograficamente seguro, idealmente aleatório. Esse passo é essencial para alcançar a segurança semântica, um conceito chave da criptografia utilizando AES. A obtenção desse vetor e da chave é feita através da derivação da chave de sessão. A Figura 4.6 apresenta um diagrama de blocos da implementação.

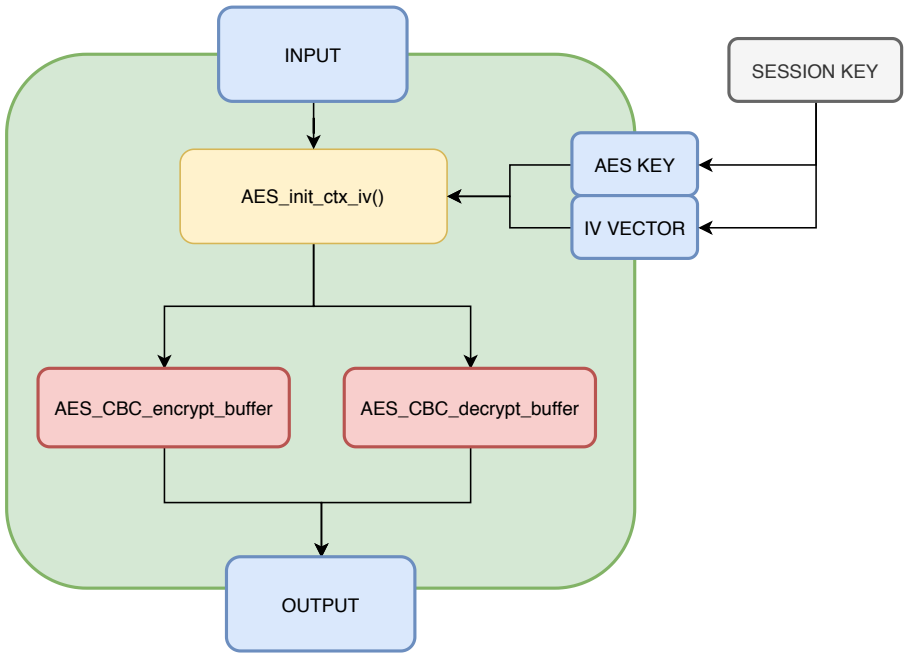


Figura 4.6: Funcionamento do AES no modo CDC.

Dois passos principais precisam ser feitos para a correta operação do AES.

- Obtenção do IV e da chave: ambas são derivadas da chave de sessão, através de um método de divisão;
- *AES-init-ctx-iv()*: esta função inicializa a biblioteca com dois parâmetros: a chave e o vetor de inicialização;
- *AES_CBC_encrypt_buffer e AES_CBC_decrypt_buffer*: estas funções realizam a encriptação ou decriptação do *buffer* enviado como parâmetro. Como o AES já foi inicializado com a chave, enviar a chave como parâmetro nessa função não é necessário.

O Código 4.10 apresenta as funções utilizadas no servidor desde a leitura das chaves da memória até o envio da mensagem criptografada.

Código 4.10: Processo de utilização do AES no servidor.

```
1     // Read Keys from secure memory
2     memoryKeyRead(target);
```

```
3
4       // Obtaining IV and AES Key
5       memcpy(AESkey,keyFromMemory,PUB_KEY_LEN/2);
6       memcpy(nounce,keyFromMemory+16,PUB_KEY_LEN/2);
7
8       // AES Setup and encryption
9       AES_init_ctx_iv(&ctx, AESkey, nounce);
10      AES_CBC_encrypt_buffer(&ctx, messageOut, 32);
11
12     // Sending message do Client
13      requireToSend();
14      sendMessage(target, messageOut);
```

Esse código é executado toda vez que o servidor deseja enviar uma mensagem para o cliente. A única exceção para isso é quando o servidor está autenticando os clientes, momento no qual a comunicação é feita em claro. Um algoritmo muito semelhante está implementado na função de envio de ambos os clientes. Na linha 2 do trecho de Código, a função *memoryKeyRead* realiza a leitura da chave de sessão guardada na memória segura. Em seguida, a função de inicialização citada anteriormente é feita, preparando o AES para uma futura encriptação. Logo depois a função *AES_CBC_encrypt_buffer* realiza a criptografia do *buffer messageOut*. Com o dado já criptografado, o Servidor avisa o periférico de que deseja enviar uma mensagem a ele utilizando a função *requireToSend()*, após a resposta positiva do periférico, o servidor efetivamente envia a mensagem ao seu destinatário utilizando a função *sendMessage*.

4.4 Aplicações Cliente e Servidor

Nesse trabalho foi criada uma aplicação que implementa um exemplo de comunicação entre clientes e servidor de forma segura. A aplicação é dividida em duas partes principais, detalhadas na Figura 4.7:

- **Autenticação**: etapa na qual o servidor gera as chaves de sessão em conjunto com seus clientes. Nesta etapa os três módulos de segurança, ECDH, SHA e AES são utilizados.
- **Comunicação**: momento no qual clientes e servidor podem realizar a troca de mensagens utilizando criptografia simétrica.

A aplicação adota um modelo de comunicação mestre-escravo, onde o cliente, sendo o escravo, só envia mensagens ao servidor caso receba uma mensagem de requi-

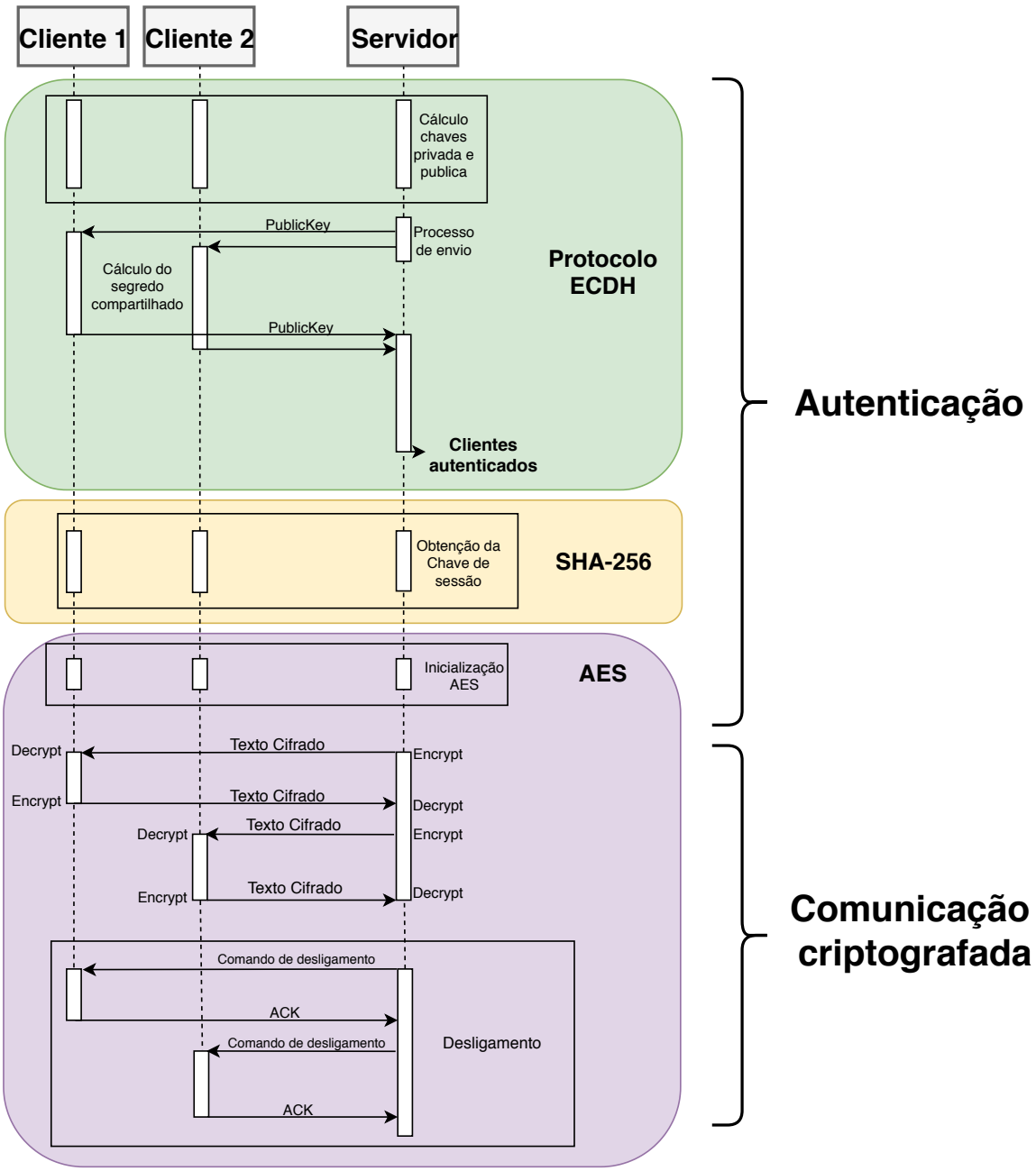


Figura 4.7: Protocolo de comunicação da plataforma criada.

sição. Isso não é uma limitação da plataforma nem do *software*, sendo a forma como a aplicação foi desenvolvida para garantir o correto sincronismo entre a troca de mensagens.

A Figura 4.8 ilustra o caminho feito pela mensagem enviada pelo servidor até o cliente. Este diagrama ilustra o processo de troca de mensagens entre o servidor e o cliente 1 através do periférico *secBridge*. A comunicação é feita a partir de comandos de *request*. Para qualquer ação de envio ou de recebimento feito através do periférico existe uma etapa de *handshaking*. Esse processo é ilustrado em todos os passos da comunicação.

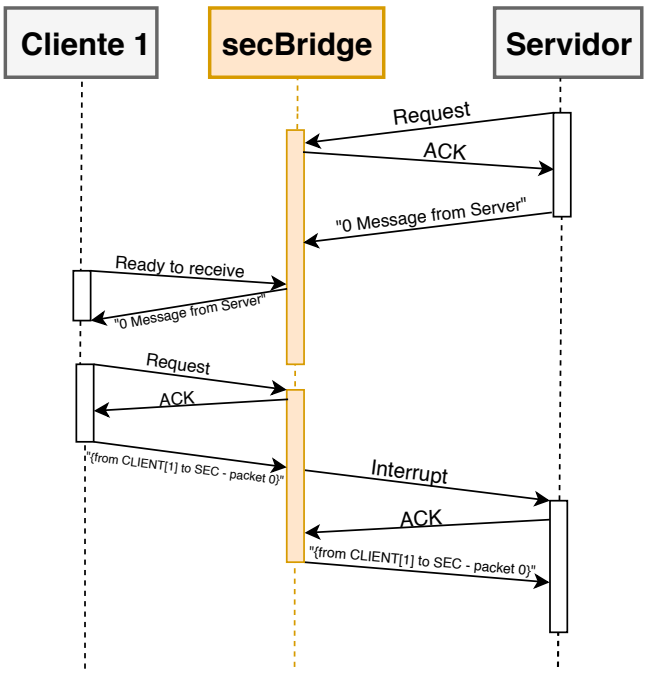


Figura 4.8: Diagrama mostrando a interação entre servidor, periférico e cliente. O *payload* de todas as mensagens trafega criptografado.

4.4.1 Servidor

Para o correto funcionamento das interrupções e da troca entre os modos de segurança, o servidor deve realizar diversos procedimentos ao iniciar sua execução. O Código 4.11 apresenta os passos para inicialização do servidor.

Código 4.11: Código da inicialização do servidor.

```
1  int main()
2  {
3      received = 0;
4
5      CPU_INIT();
6      setupTranslationTable();
7      enableTLB();
8      setSVCHandler();
9      computeKeys();
10     REGISTER_ISR(irq, irq_handler, (void *)NULL);
11
12     ENTER_NON_SECURE_USING_MONITOR();
13
14     if( iteration != NB_ITERATIONS) {
```

```
15              ENABLE_INTERRUPTS();
16              run();
17      }
18
19      exit(0);
20  }
```

Fazem parte do procedimento de *boot* as seguintes funções:

- *CPU_INIT()*: configura o modo de operação inicial do ARM.
- *setupTranslationTable()* e *enableTLB()*: essa etapa configura a TLB (*Translation lookaside buffer*) e logo em seguida a habilita. A TLB é utilizada para mapear seções de memória para o TZPC.
- *setSVCHandler()*: função dedicada para o registro da função que tratará interrupções SVC (*Super Visor Call*).
- *computeKeys()*: nessa etapa é feita geração da chave privada e o cálculo da chave pública utilizando curvas elípticas.
- *REGISTER_ISR()*: esta função aponta a função que será chamada quando uma interrupção é feita através da porta *irq*.

Em seguida, para executar a aplicação, o processador altera seu modo de segurança para não seguro utilizando a função *ENTER_NON_SECURE_USING_MONITOR*. Após esta mudança de modo, a função *run()* é chamada, a qual executa *NB_ITERATIONS*. O trecho de Código 4.12 corresponde a esta função.

Código 4.12: Código simplificado da função *run()*.

```
1  void run (){
2
3      // Envia a chave publica para os processadores não seguros
4      // executado um única apenas uma vez.
5      if (!sendKey_lock){
6          sendKey(0);
7          sendKey(1);
8          sendKey_lock = 1;
9      }
10     for(  ; iteration < NB_ITERATIONS ; )  {
11
```

```
12          // envia pacote para um processador não seguro - **código simplificado**
13          sendProcess(0, cmd );
14          while( received==0 );    // espera resposta da interrupção (SEMÁFORO)
15
16          DISABLE_INTERRUPTS();
17
18          if (sessionKeyState[0] == 0 || sessionKeyState[1] == 0 )
19          {
20              memoryDataRead(received);     // this step compute the session key
21              computeSessionKey();
22              memoryKeyWrite(messageSender);
23              sessionKeyState[messageSender] = 1;
24          } else {
25              printf("RS received a ciphered message from RNS
26                    %d\nStarting AESdecryption...\n\n",messageSender+1);
27              decryptMessage();
28          }
29
30          ENABLE_INTERRUPTS();
31          received = 0;
32          ++iteration;
33      }
34      puts("###### R U N       E N D E D");
35      enterNonSecure();    // non secure
36  }
```

A função *run()* inicia distribuindo as chaves públicas aos clientes (linhas 6 e 7). Este processo é realizado uma única vez.

Na sequência, o servidor envia de forma alternada a cada cliente solicitação de mensagem (o código está simplificado por questões de legibilidade), através da função *sendProcess()* – linha 13. Esta mensagem é transmitida encriptada, via o periférico *secBrige*, para o cliente selecionado. O comando seguinte atua como um semáforo (variável *received*, linha 14), ativado pela função de tratamento de interrupção (detalhada no Código 4.14).

Após a liberação do semáforo, O servidor inicia o tratamento da mensagem recebida. Este tratamento é feito com as interrupções desabilitadas, de forma a evitar que dados cheguem e não sejam tratados. Inicialmente, o servidor verifica se as chaves públicas dos clientes já foram recebidas (a primeira mensagem enviada pelos clientes é sempre sua chave pública) - linhas 20 a 23. Caso a mensagem recebida seja uma chave pública, o servidor inicia o protocolo Diffie-Hellman utilizando a função *computeSessionKey()*. Após

esse processamento o servidor armazena a chave de sessão na memória segura através da função *memoryKeyWrite()*.

Caso a mensagem recebida seja um texto criptografado, o programa redireciona a execução para a função *decryptMessage()*. A mensagem recebida é armazenada pelo tratador de interrupção na memória segura do processador. O Código 4.13 mostra a implementação da função *decryptMessage()*.

Código 4.13: Função que descriptografa a mensagem recebida.

```
1  void decryptMessage(){
2      //Memory Read
3      memoryKeyRead(messageSender);
4      memoryDataRead(received);
5
6      //Key derivation
7      memcpy(AESkey,keyFromMemory,PUB_KEY_LEN/2);
8      memcpy(IV,keyFromMemory+16,PUB_KEY_LEN/2);
9
10     // AES operations
11     AES_init_ctx_iv(&ctx, AESkey, IV);
12     AES_CBC_decrypt_buffer(&ctx, dataFromMemory, 32);
13 }
```

Nas primeiras linhas da função é feita a leitura da memória segura para obter a chave de sessão especifica para o canal de comunicação a ser utilizado. Em seguida, é feito a leitura do endereço de dados dessa mesma memória para obter a mensagem recebida. Depois, a chave do AES e o IV são derivados da chave de sessão. Dessa forma as operações de configuração e encriptação do AES podem ser feitas. O resultado esperado dessa função é o *array dataFromMemory* contendo a mensagem em claro.

No final do processo, as interrupções voltam a ser habilitadas, o semáforo é desativado, e o contador de interações é incrementado (linhas 30-32).

A função de tratamento de interrupção é apresentada de forma simplificada no Código 4.14. Ao iniciar os tratamentos de interrupções o servidor deve salvar o contexto do processador[1], em seguida as interrupções são desabilitadas, e alterna-se o processador para modo seguro (linhas 3-6). Na sequência, lê-se do periférico *secBrigde* o cabeçalho da mensagem enviada pelo cliente (linhas 8-9). A linha 15 habilita a comunicação com o periférico *secBrigde*, e a mensagem é lida e armazenada na área de memória segura. O semáforo que libera a execução da função *run()* encontra-se na linha 21 - variável *received*.

[1]No presente trabalho o salvamento de contexto não está implementado devido à ocorrências de exceção no modo seguro.

No final do processo, o processador volta ao modo não seguro.

Código 4.14: Código simplificado da função *irq_handler()*.

```
1  static void irq_handler(void *ud)
2  {
3      // Save the processor state  - code omitted
4      DISABLE_INTERRUPTS();
5      ackInterruptHandler();
6      ENTER_SECURE_USING_MONITOR();
7
8      for(i = 0; i< HEADER_SIZE; i++)
9           buffer[i] = *(RG_READ_HEADER_DATA);
10
11     header.sender =  buffer[0];         // client number
12     header.messageSize =  buffer[4];
13     messageSender = header.sender;
14
15     ENABLE_BRIDGE_COMMUNICATION();
16     receiveMessage(header.messageSize);
17     DISABLE_BRIDGE_COMMUNICATION();
18
19     ENTER_NON_SECURE_USING_MONITOR();
20
21     received =  buffer[4];     /// received <- message size, sendo também o SEMÁFORO
22
23     ENABLE_INTERRUPTS();
24
25     // restore context  - code omitted
26 }
```

O Código completo do servidor encontra-se disponível em: https://github.com/rafareusch/tccreusch.git.

4.4.2 Cliente

Nessa aplicação o cliente é o escravo da comunicação, então ele sempre atua de acordo com os comandos enviados pelo servidor. O Código 4.15 implementa a função

main() do cliente, responsável por inicializar os módulos de segurança e executa a aplicação.

Código 4.15: Código demonstrando a inicialização do programa do cliente.

```
1  int main()
2  {
3      enterNonSecure();
4      printf("Hello from non-secure processor %d!\n",RNS_ID);
5
6      requireToRead(wd);
7
8      computeSessionKey();
9
10     AES_init()
11
12     example1();
13
14     return(0);
15 }
```

A implementação da inicialização do cliente é a praticamente uma versão oposta da presente no servidor, porém, como o cliente recebe a chave pública do servidor logo no início da execução, é possível efetuar todo processo de autenticação de uma única vez. O cliente utiliza a função *requireToRead()* para alertar o periférico *secBridge* que ele está pronto para receber a chave pública do servidor. Em seguida, a chave de sessão é calculada utilizando a função *computeSessionKey()*. Depois a função *AES_init()* irá pré-processar a chave de sessão para obter a chave simétrica e o vetor de inicialização, melhorando a eficiência da aplicação. Ao final deste processo, o cliente executa a função *example1()* na qual implementa a comunicação criptografada com o servidor. Esta implementação pode ser vista no Código 4.16. Como a biblioteca do AES é compartilhada entre o servidor e os clientes, as funções de criptografia implementadas são muito semelhantes, porém no caso do cliente, é preciso uma menor quantidade de passos para configuração do AES pois ele só possui uma chave de sessão, diferentemente do servidor que possui uma para cada cliente.

Código 4.16: Função que implementa a comunicação com o servidor.

```
1  void example1(){
2      int fim, i=0;
3
4      for(;;) {
```

```
5           requireToRead(wd);
6           decryptMessage();
7
8           if (wd[0]=='1')  fim=1;  else fim=0;
9
10          sprintf(wd,"{from NONSEC[%d] to SEC - packet %d}", RNS_ID, i++);
11          AES_init_ctx_iv(&ctx, AESkey, nounce);
12          AES_CBC_encrypt_buffer(&ctx, wd, 32);
13          sendMessage((RNS_ID-1), wd);
14
15          if(fim) break;
16      }
17
18  }
```

Essa função recebe dados do servidor, e em seguida emite uma resposta. A função inicia aguardando dados provenientes do servidor, na linha 5. Em seguida, realiza a decriptação da mensagem executando à função decryptMessage(), a qual é muito semelhante a função presente no servidor. O resultado dessa operação é que o *array wd* seja convertido para texto em claro.

Com a mensagem descriptografada, o *software* do cliente verifica qual é o valor presente no primeiro *byte* da mensagem recebida. Essa posição atua como um cabeçalho e indicará para o cliente qual é o comando enviado pelo servidor. Dois comandos foram implementados: *(i)* 0 - solicita que o cliente envie uma nova mensagem; *(ii)* 1 - comando de desligamento do cliente. Os *bytes* restantes contêm a mensagem de texto enviada pelo servidor.

O processo de envio da mensagem é implementado nas linhas 10 a 13. Inicialmente o vetor *wd* é populado com uma mensagem em texto simples. Em seguida, as mesmas duas funções do AES presentes no servidor, são executadas, com a intenção de criptografar o vetor *wd*, isso é feito nas linhas 11 e 12. Finalmente a função *sendMessage* efetivamente envia a mensagem para o *secBridge* com destino ao servidor.

5. RESULTADOS

Este capítulo apresenta a validação da plataforma através da execução do *software* apresentado na Seção 4.4. Cada módulo de segurança foi testado individualmente com o objetivo de validar cada uma das etapas da aplicação.

5.1 ECDH

Antes que o protocolo de trocas de chaves *Diffie-Hellman Key Exchange* seja executado, as chaves públicas e privadas devem ser geradas utilizando curvas elípticas. A Figura 5.1 ilustra a saída da função de cálculo das chaves implementada no Cliente 1.

```
1
2  ############## Client 1 is initializing cryptographic variables...
3
4  Client 1 Private Key: HEX F1ACA979B0ABAB393FB826DDD30C143295FE469B08716BD28D8EBEDCD54A2642
5  Client 1 Public Key: 59BD8EEB937789CC83512D8D07B37F5088BA1D5D84B21757FD5A81DA156DAE23
6
```

Figura 5.1: Geração de chaves privada e pública no Cliente 1.

Após cada componente da plataforma realizar a geração das próprias chaves, é necessário que o servidor envie sua chave pública para os clientes e vice-versa. A Figura 5.2 ilustra o Servidor realizando o recebimento da chave pública do Cliente 1.

```
15 >>> Server reports it has received Public Key from Client 1
16
17 SMC: Entering Secure mode
18 Reading DATA from secure memory
19 Memory DATA read:  59BD8EEB937789CC83512D8D07B37F5088BA1D5D84B21757FD5A81DA156DAE23
20 SMC: Leaving Secure mode
21
```

Figura 5.2: Recebimento da chave pública do cliente.

Com a troca dos parâmetros públicos concluída, ambas partes agora podem realizar o protocolo *Diffe-Hellman* para obter o segredo compartilhado, de onde a chave de sessão será derivada. A Figura 5.3 mostra o Cliente 1 e o Servidor obtendo a mesma *Session Key*, com cada um utilizando sua chave privada, junto com a chave pública da outra parte. Esta figura mostra os dados de saída de forma sequencial, porém, na verdade eles são processados de forma paralela e independente. Isso é causado por uma disputa pela utilização do terminal pelas funções de *printf*.

```
// Server
Client Public Key: 59BD8EEB937789CC83512D8D07B37F5088BA1D5D84B21757FD5A81DA156DAE23
Server secret Key: 4B65E1E62A9B323E12F40039AE8C5CF25CE7BD200E4FAE0289D88BB9F3865D6C
Server is now computing the Session Key...

>>> Server and Client 1 Session Key: C02D53521C74C1D8EA23D4E2BFD6DC7271E57A710F4ABD6420B4B7D0140F0F03


// Client 1
Sever Public Key: 3B970482D2B421285741BA658F815E0A7A730358FAE4123AB0154AF0CFE45058
Client secret Key: F1ACA979B0ABAB393FB826DDD30C143295FE469B08716BD28D8EBEDCD54A2642
CLI1 is now processing the Session Key

>>>>  CLI1 Session Key: C02D53521C74C1D8EA23D4E2BFD6DC7271E57A710F4ABD6420B4B7D0140F0F03
```

Figura 5.3: Dados de saída da autenticação do Cliente 1 e o Servidor.

5.2 SHA-256

Na aplicação criada neste trabalho, o SHA-256 está incluído no processo de autenticação, trabalhando em conjunto com o protocolo ECDH. A obtenção da chave de sessão demonstrado na Figura 5.3 é realizado através de uma operação de *hash* no segredo compartilhado. A Figura 5.4 demonstra essa operação.

```
CLI1: Shared Secret: 22599247D0CC96844A32335F2AEDA539EAF753F3A4A0ED40FC425D4F352ED4B5
CLI1: Performing hash operation
>>>> CLI1 Session Key: C02D53521C74C1D8EA23D4E2BFD6DC7271E57A710F4ABD6420B4B7D0140F0F03
```

Figura 5.4: Dados de saída do algoritmo de *hash*.

É possível confirmar o funcionamento o algoritmo de *hash* implementado utilizando ferramentas *online* que realizam o *SHA-256*. A Figura 5.5 mostra que o resultado obtido neste trabalho é compatível com o resultado da ferramenta, comprovando seu funcionamento [Tools, 2017].

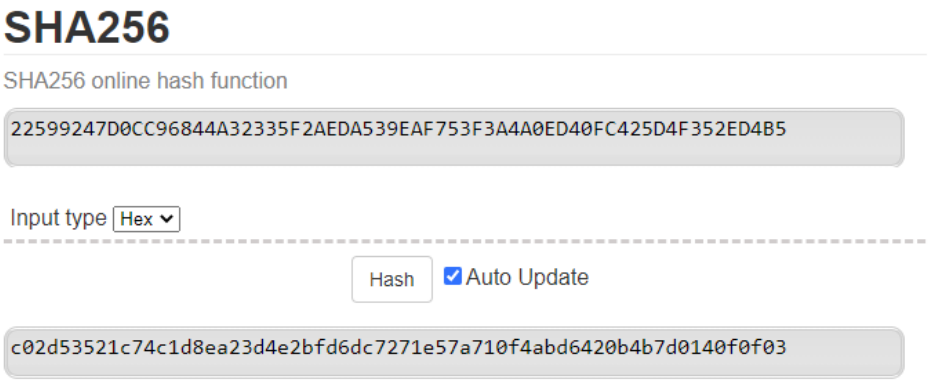


Figura 5.5: Ferramenta utilizada para validar a operação de *hash*.

5.3 AES

Para validação do AES, será mostrado nesta seção uma transmissão de uma mensagem criptografada do Cliente 1 para o Servidor. Assim é possível visualizar ambas as etapas da criptografia.

O primeiro passo é a obtenção da chave simétrica e do vetor de inicialização. Como citado anteriormente, nesse trabalho esses dois vetores são derivados da chave de sessão. Essas etapas são ilustradas na Figura 5.6

```
59  CLI1 Session Key: C02D53521C74C1D8EA23D4E2BFD6DC7271E57A710F4ABD6420B4B7D0140F0F03
60
61  CLI AES KEY: C02D53521C74C1D8EA23D4E2BFD6DC72
62
63  CLI IV: 71E57A710F4ABD6420B4B7D0140F0F03
```

Figura 5.6: Saída de dados do processo de derivação da chave e do vetor de inicialização.

O segundo passo é utilizar os vetores calculados anteriormente para inicializar o AES. Com a biblioteca inicializada, é possível encriptar a mensagem. Esse processo é mostrado pela Figura 5.7.

```
65  CLI PLAIN TEXT: {from CLIENT[1] to SEC - packet 0}
66
67  CLI CIPHERED TEXT: 788B69F294B618542D48DD3646A51717B43487BE47AE350E664AE15E0A40BE21
68
69  CLI: Requiring write permission
70  Info (RNS TX Register) nonSecToSecValidation/nonSecToSec: CLI is now transmitting!
71  CLI: Ready to send!
72  Info (TX Header Write) nonSecToSecValidation/nonSecToSec: Write to header data was 0!
73  Info (TX Header Write) nonSecToSecValidation/nonSecToSec: Write to header data was 34!
74  Info (TX Write) nonSecToSecValidation/nonSecToSec: End of transmission, generating interrupt to secure processor!
75  CLI: Sent a message!
```

Figura 5.7: Processo de encriptação e envio da mensagem.

É possível verificar na linha 65 a mensagem que o cliente deseja enviar para o servidor. Em seguida ela é criptografada utilizando o AES. A linha 67 representa a mesma mensagem, porém criptografada. Nas linhas subsequentes o cliente realiza o envio da mensagem para o *secBridge* com destino ao servidor.

A Figura 5.8 apresenta os dados de saída após o recebimento da mensagem no lado do servidor.

Inicialmente o servidor realiza duas leituras na memória segura, a primeira é no endereço onde as chaves de sessão estão armazenadas. Essa operação é feita todas as vezes em que um dado vai ser descriptografado, pois o servidor possui duas chaves de sessão, uma para cada cliente. A segunda leitura é na região de dados temporários, isso é feito para obter a mensagem recebida, uma vez que o *irq_handler* escreve nessa memória logo após recebê-la.

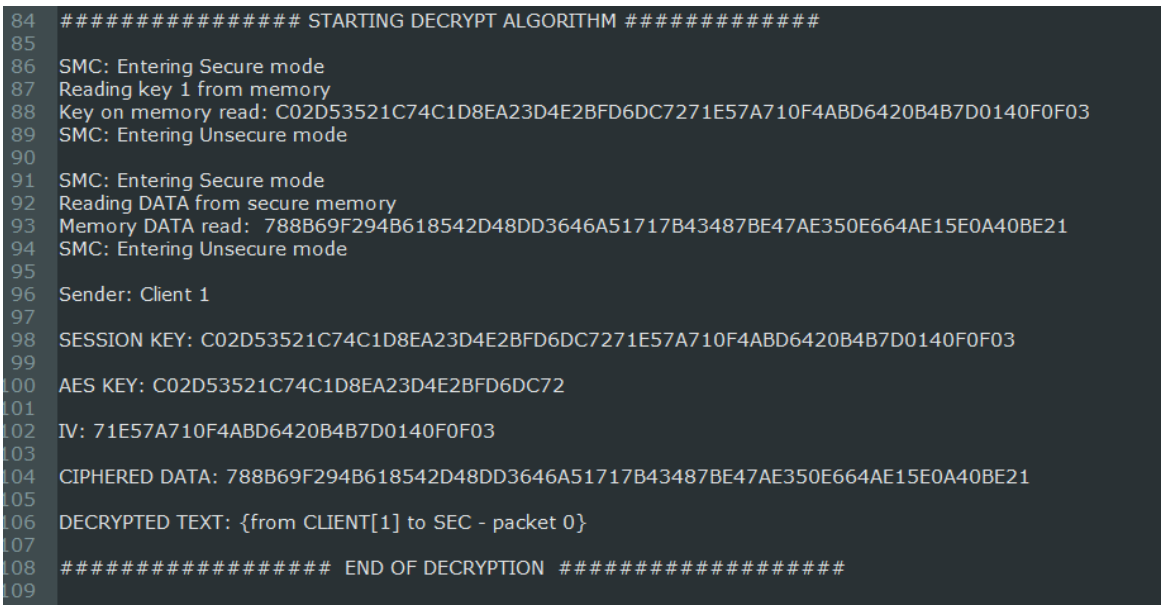


Figura 5.8: Dados de saída do servidor.

Em seguida, a chave simétrica e o vetor de inicialização são derivados da chave de sessão, como pode ser visto nas linhas 98, 100 e 102. Na linha 104 é possível ver o texto cifrado recebido, completamente ilegível. Então são feitas as operações do AES que irão descriptografar o vetor recebido. O resultado dessa operação deve ser a mensagem decifrada e legível, como mostra a linha 106. A mensagem pode ser comparada com a Figura 5.7, sendo possível ver que a mensagem é idêntica, comprovando o funcionamento correto do algoritmo.

Os dados completos de execução estão presentes na página https://github.com/rafareusch/tccreusch.git.

6. CONCLUSÕES

O presente trabalho de conclusão permitiu a utilização de conhecimentos adquiridos ao longo do curso de Engenharia de Computação na área de arquitetura de computadores e programação de periféricos. Possibilitou também agregar conhecimentos na área de segurança da informação, através de estudos sobre algoritmos e bibliotecas de segurança. Estes temas são muito relevantes para o mundo atual, onde dispositivos embarcados estão cada vez mais interconectados e presentes no nosso dia a dia.

Este trabalho apresentou uma plataforma que implementa algoritmos de criptografia e autenticação na comunicação entre um servidor e clientes. Um protocolo de autenticação foi adotado para permitir que os clientes consigam criar uma chave de sessão em conjunto com o servidor.

A plataforma foi validada realizando testes individuais nos módulos de segurança e posteriormente em testes integrados entre os processadores. O estudo, implementação e integração destes módulos exigiram um alto nível de esforço de pesquisa, consequência da ausência de implementações semelhantes para um arquitetura cliente-servidor utilizado o processador ARM, em conjunto com *Trustzone*. A validação ocorreu em um ambiente *bare metal* no processador ARM, que também exigiu uma grande e contínua pesquisa durante toda realização deste trabalho. Dentre os desafios destaca-se a complexidade no gerenciamento da troca de regiões segura para não segura, e vice-versa.

Tanto a plataforma como o *software* podem ser aprimorados em trabalhos futuros. No quesito segurança da informação, protocolos para estabelecer uma nova chave de sessão depois de um número determinado de mensagens, poderia adicionar uma maior segurança contra de ataques. No quesito de *software*, um sistema operacional, como o *freeRTOS* poderia ser implementado, abrindo um leque de possibilidades. O mesmo é válido para o chaveamento de contexto no tratamento da interrupção, onde melhoramentos podem ser realizados para permitir computações mais complexas.

REFERÊNCIAS BIBLIOGRÁFICAS

[ARM, 2010] ARM (2010). *Cortex-A8 - Technical Reference Manual*. Disponível em: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf.

[ARM, 2011] ARM (2011). *ARM Compiler toolchain*. Disponível em: http://infocenter.arm.com/help/topic/com.arm.doc.dui0489c/DUI0489C_arm_assembler_reference.pdf.

[ARM, 2016] ARM (2016). *SMC CALLING CONVENTION - System Software on ARM Platforms*. Disponível em: https://chasinglulu.github.io/downloads/ARM_DEN0028B_SMC_Calling_Convention.pdf.

[ARM, 2019] ARM (2019). *Arm Architecture Reference Manual - Armv8, for Armv8-A architecture profile*. Disponível em: https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf?_ga=2.153466424.1474128376.1574655536-220125155.1571781081.

[Bakos, 2016] Bakos, J. D. (2016). *Embedded Systems - ARM Programming and Optimization*. Morgan Kaufmann.

[Bernstein, 2006] Bernstein, D. J. (2006). Curve25519: New diffie-hellman speed records. In Yung, M., Dodis, Y., Kiayias, A., and Malkin, T., editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer.

[Bernstein et al., 2014] Bernstein, D. J., van Gastel, B., Janssen, W., Lange, T., Schwabe, P., and Smetsers, S. (2014). TweetNaCl: A Crypto Library in 100 Tweets. In Aranha, D. F. and Menezes, A., editors, *LATINCRYPT*, volume 8895 of *Lecture Notes in Computer Science*, pages 64–83. Springer.

[Burhan et al., 2018] Burhan, M., Rehman, R. A., Khan, B., and Kim, B. (2018). IoT Elements, Layered Architectures and Security Issues: A Comprehensive Survey. *Sensors*, 18(9):2796.

[Conte, 2012] Conte, B. (2012). Crypto Algorithms/sha256.c. Disponível em: https://github.com/B-Con/crypto-algorithms/blob/master/sha256.c.

[Daemen and Rijmen, 2000] Daemen, J. and Rijmen, V. (2000). Rijndael for AES. In *The Third Advanced Encryption Standard Candidate Conference, April 13-14, 2000, New York, New York, USA*, pages 343–348. National Institute of Standards and Technology,.

[Davies, 1997] Davies, D. (1997). *A brief history of cryptography*. Elsevier.

[Diffie and Hellman, 1976] Diffie, W. and Hellman, M. E. (1976). New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654.

[Fuques, 2019] Fuques, H. S. (2019). *Pesquisa e Desenvolvimento de Mecanismo de Comunicação Segura Entre Processadores Utilizando Tecnologia Trustzone*. Trabalho de Conclusão de Curso, Engenharia de Computação, PUCRS.

[IMPERAS, 2019] IMPERAS (2019). Open Virtual Plataforms. Disponível em: http://www.ovpworld.org.

[Kokke, 2019] Kokke (2019). Crypto Algorithms/aes.c. Disponível em: https://github.com/kokke/tiny-AES-c#readme.

[Ngabonziza et al., 2016] Ngabonziza, B., Martin, D., Bailey, A., Cho, H., and Martin, S. (2016). TrustZone Explained: Architectural Features and Use Cases. In *IEEE International Conference on Collaboration and Internet Computing CIC*, pages 445–451.

[NIST, 2001] NIST (2001). ADVANCED ENCRYPTION STANDARD (AES). Disponível em: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[NIST, 2012] NIST (2012). Secure Hash Standard (SHS) . Disponível em: https://csrc.nist.gov/CSRC/media/Publications/fips/180/4/archive/2012-03-06/documents/fips180-4.pdf.

[Paar and Pelzl, 2010] Paar, C. and Pelzl, J. (2010). *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2th edition.

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. M. (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126.

[Satyanarayanan, 2017] Satyanarayanan, M. (2017). Edge computing. *Computer*, 50(10):36–38.

[Tools, 2017] Tools, O. (2017). SHA256 online hash function. Disponível em: https://emn178.github.io/online-tools/sha256.html.