UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

MATHEUS KOWALCZUK FERST

IMPLEMENTAÇÃO DE COMUNICAÇÃO SEGURA COM MODBUS E TLS

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2018

MATHEUS KOWALCZUK FERST

IMPLEMENTAÇÃO DE COMUNICAÇÃO SEGURA COM MODBUS E TLS

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso de Engenharia de Computação da Coordenação de Engenharia de Computação - COENC - da Universidade Tecnológica Federal do Paraná - UTFPR, Câmpus Pato Branco, como requisito parcial para obtenção do título de Engenheiro da Computação .

Orientador: Prof. Dr. Bruno César Ribas

Coorientador: Prof. Dr. Gustavo Weber Denardin

PATO BRANCO

2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Engenharia de Computação

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

Às 14 horas do dia 04 de julho de 2018, na sala V006, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, reuniu-se a banca examinadora composta pelos professores Bruno Cesar Ribas (orientador), Gustavo Weber Denardin (coorientador), Jean Paulo Martins e Luciene de Oliveira Marin para avaliar o trabalho de conclusão de curso com o título **Implementação de comunicação segura com Modbus e TLS**, do aluno **Matheus Kowalczuk Ferst**, matrícula 01586149, do curso de Engenharia de Computação. Após a apresentação o candidato foi arguido pela banca examinadora. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

Bruno Cesar Ribas Orientador (UTFPR)	Gustavo Weber Denardin Coorientador (UTFPR)
Jean Paulo Martins (UTFPR)	Luciene de Oliveira Marin (UTFPR)
Profa. Beatriz Terezinha Borsoi Coordenador de TCC	Prof. Pablo Gauterio Cavalcanti Coordenador do Curso de Engenharia de Computação

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

Attacks always get better; they never get worse.

US National Security Agency (NSA)

RESUMO

FERST, Matheus Kowalczuk. Implementação de comunicação segura com Modbus e TLS. 2018. 136f. Monografia(Trabalho de Conclusão de Curso 2) - Curso de Engenharia de Computação , Universidade Tecnológica Federal do Paraná, Campus de Pato Branco. Pato Branco, 2018.

Sistemas de Supervisão e Aquisição de Dados (SCADA) são utilizados em diversas áreas, como em processos de manufatura na industria e no controle de ativos de infraestrutura. Algumas destas aplicações assumem caráter critico, onde a ocorrência de uma falta pode levar a danos e perdas irreparáveis. A operação destes sistemas baseia-se protocolos, como DNP3 e Modbus, que muitas vezes não possuem nenhum mecanismo de segurança originalmente. O presente trabalho propõem uma versão segura do protocolo Modbus, baseado sua camada de transporte no protocolo *Transport Layer Security* (TLS) para endereçar suas falhas de segurança.

Palavras-chave: Modbus. TLS. SSL. SCADA. Segurança.

ABSTRACT

FERST, Matheus Kowalczuk. title. 2018. 136f. Monografia(Trabalho de Conclusão de Curso 2) - Curso de Engenharia de Computação , Universidade Tecnológica Federal do Paraná, Campus de Pato Branco. Pato Branco, 2018.

Supervisory Control and Data Acquisition (SCADA) systems are used in a great variety of fields, like industrial manufacturing processes and to control infrastructure assets. Some applications have a critical character and a fault in these systems may lead to unrecoverable damages and losses. These systems rely on protocols, such as DNP3 and Modbus, that originally lack any security mechanisms. This work presents a new secure version of the Modbus protocol, basing its transport layer in the Transport Layer Security (TLS) protocol to address its security flaws.

Keywords: Modbus. TLS. SCADA. Security.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

ADU	*Application Data Unit*.
AES	*Advanced Encryption Standard*.
CBC	*Cipher-Block Chaining*.
CFB	*Cipher Feedback*.
CRC	*Cyclic Redundancy Check*.
DHCP	*Dynamic Host Configuration Protocol*.
DoS	*Denial of Service*.
DTLS	*Datagram Transport Layer Security*.
ECB	*Electronic Codebook*.
HMAC	*Hashed* MAC.
HTTPS	*Hyper Text Transfer Protocol Secure*.
ICS-CERT	*Industrial Control Systems Cyber Emergency Response Team*.
IDE	*Integrated Development Environment*.
IDS	*Intrusion Detection System*.
IETF	*Internet Engineering Task Force*.
MAC	*Message Authentication Code*.
MTU	*Master Terminal Unit*.
NTP	*Network Time Protocol*.
OFB	*Output Feedback*.
PDU	*Protocol Data Unit*.
PSK	*Pre Shared Key*.
RTOS	*Real Time Operational System*.
RTU	*Remote Terminal Unit*.
SCADA	*Supervisory Control and Data Acquisition*.
SCTP	*Stream Control Transmission Protocol*.

SIP	*Session Initiation Protocol*.
SMTP	*Simple Mail Transfer Protocol*.
SSL	*Secure Socket Layer*.
TLS	*Transport Layer Security*.
UDP	*User Datagram Protocol*.
VPN	*Virtual Private Network*.
WWW	*World Wide Web*.

SUMÁRIO

1 INTRODUÇÃO

Sistemas de Supervisão e Aquisição de Dados (SCADA) são usados em uma grande variedade de áreas, como em usinas de energia, coleta e tratamento de água e esgoto, oleodutos, gasodutos e em processos de manufatura industrial (FOVINO *et al.*, 2009a; STOUFFER *et al.*, 2011). Algumas aplicações, especialmente as relacionadas a ativos de infraestrutura, assumem um caráter crítico, e uma falta nestes sistemas pode levar a danos e perdas irreparáveis e afetar a população em geral.

A operação desses sistemas depende de redes de comunicação e de protocolos adequados aos requisitos da aplicação. Alguns dos protocolos comumente usados, como DNP3 e Modbus, não possuem originalmente nenhum mecanismos de segurança, de forma que a exposição destes sistemas a uma rede pública, seja pela implantação incorreta do sistema SCADA ou pela exploração de alguma vulnerabilidade, permitiria que agentes não autorizados controlassem o sistema.

De outra forma, a exposição intencional poderia viabilizar novas funcionalidades, como sistemas de controle distribuído, ou reduzir o custo operacional para gerir ativos geograficamente dispersos, como redes de distribuição de energia e de transporte ferroviário (STOUFFER *et al.*, 2011).

O protocolo Modbus foi criado pela Modicon, atual Schneider Electric, em 1979 e tornou-se o padrão de fato para comunicações seriais na industria (MODBUS, 2006). O protocolo foi inicialmente desenvolvido para comunicações sobre redes seriais assíncronas, como RS-232 e RS-485, e posteriormente adaptado para redes TCP/IP.

Transport Layer Security (TLS) é um protocolo da camada de trans-

porte que provê comunicações seguras pela Internet (DIERKS; RESCORLA, 2008). Este protocolo foi criado pela Netscape Communications em 1993, visando especialmente a segurança da *World Wide Web* (WWW) (OPPLIGER, 2009). Com sua evolução, entretanto, o protocolo se tornou efetivamente um *framework* para o desenvolvimento e implementação de protocolos criptográficos (RISTIC, 2014).

O presente trabalho descreve a implementação de uma versão segura do protocolo Modbus, obtida pela substituição da camada de transporte da versão TCP do protocolo, de forma a baseá-lo em TLS.

Utilizando a ferramenta de buscas Shodan[1], que varre toda a faixa de endereços IP da Internet em busca de dispositivos conectados, na data de 27 de junho de 2018 foram detectados[2] 16.210 máquinas com a porta reservada ao protocolo Modbus aberta. Isso mostra que muitas implementações negligenciam a segurança do sistema ou apresentam problemas no isolamento entre a rede de controle e a Internet.

Corroborando com esta informação, o *Industrial Control Systems Cyber Emergency Response Team* (ICS-CERT), órgão do Departamento de Segurança Interna dos Estados Unidos que monitora ataques auxilia a industria do país com relação a segurança da informação na industria, afirma que a maioria das vulnerabilidades observada durante as avaliações de segurança em instalações industriais feitas nos últimos quatro anos estão relacionadas ao isolamento incorreto entre a rede de controle e de negócios (MONITOR, 2017).

Isso mostra que nem sempre é possível confiar na separação da rede SCADA como a única segurança em uma aplicação industrial, como se fazia em sistemas baseados somente em redes seriais (CREERY; BYRES, 2005). A adição de mecanismos de segurança ao protocolo Modbus, neste caso, apresenta-se como uma camada a mais de proteção para situações onde a implementação do sistema SCADA pode não ter sido isolada corretamente.

[1] `https://www.shodan.io/`
[2] `https://www.shodan.io/report/M4rgHZpA`

Para sistemas de controle distribuído, por outro lado, proteger a aplicação pode simplificar a implementação das comunicações entre dispositivos geograficamente dispersos, como o caso de redes de distribuição de energia, redes de abastecimento de água ou de malhas ferroviárias (STOUFFER *et al.*, 2011). Apresentam-se a seguir, os principais objetivos deste trabalho.

1.1 OBJETIVO

Propor uma implementação do protocolo Modbus baseada em TLS a fim de solucionar os principais problemas de segurança do protocolo.

1.1.1 OBJETIVOS ESPECÍFICOS

- Investigar os problemas de segurança do protocolo Modbus.
- Levantar os possíveis desafios decorrentes do uso de TLS como protocolo da camada de transporte.
- Implementar a solução proposta.
- Verificar o desempenho da implementação com relação à latência, *jitter* e *overhead* do protocolo.

1.2 ESTRUTURA E ORGANIZAÇÃO DESTE TRABALHO

Este trabalho está dividido em seis capítulos. O primeiro deles, esta introdução, apresentou o contexto da segurança em sistemas SCADA, os protocolos Modbus e TLS e a proposta deste trabalho. O Capítulo 2 traz uma revisão dos conceitos básicos de criptografia, apresenta o histórico dos protocolos Modbus e TLS e detalha o funcionamento destes.

O Capítulo 3 apresenta outros trabalhos relacionados a seguraça de sistemas SCADA e do protocolo Modbus. O método utilizado neste trabalho e

o desenvolvimento da solução proposta são detalhados no Capítulo 4. Os resultados obtidos são apresentados e discutidos no Capítulos 5 e o Capítulo 6 finaliza este trabalho com um apanhado do que foi visto, da solução desenvolvida e resultados obtidos e sugere possíveis trabalhos futuros.

Adicionalmente, os Apêndices A e B descrevem detalhadamente as modificações realizadas nas bibliotecas *libmodbus* e *FreeModbus*, respectivamente.

2 REVISÃO BIBLIOGRÁFICA

Alguns conceitos precisam ser resgatados para a discussão a respeito da segurança do protocolo Modbus, das técnicas utilizadas para sua segurança e do funcionamento do protocolo TLS. Este capítulo apresenta uma breve revisão sobre criptografia e detalha os protocolos Modbus e TLS.

2.1 CRIPTOGRAFIA

A etimologia do termo criptografia remete às palavras gregas "*kryptós*" (escondido, secreto) e "*graphein*" (escrita), podendo ser compreendida como "escrita escondida" ou "escrita secreta" (OPPLIGER, 2009, p. 17). Este significado remete aos primórdios da criptografia, onde as técnicas utilizadas baseavam-se principalmente na segurança por obscuridade, onde a segurança das informações dependia do desconhecimento do código utilizado por aqueles que não faziam parte da comunicação. Além disso, todos os usuários do sistema criptográfico precisavam conhecer os detalhes de como utilizar o código. A história moderna da criptografia começa com a tecnologia de comunicação elétrica (VAUDENAY, 2006, p. 2).

Um sistema criptográfico pode ser definido como um "conjunto de algoritmos criptográficos em conjunto com um processo de gerenciamento de chaves que possibilite o uso destes algoritmos em algum contexto de uma aplicação" (SHIREY, 2000). Este sistema pode ou não utilizar algum parâmetro secreto, uma chave, e esta chave pode ou não ser compartilhada (OPPLIGER, 2009, p. 21). Podemos dividir então em três tipos de sistemas criptográficos, que serão explorados a seguir conforme seus aspectos pertinentes a este trabalho.

2.1.1 SISTEMAS CRIPTOGRÁFICOS SEM CHAVE

Sistemas criptográficos sem chave não utilizam nenhum tipo de parâmetro secreto, tendo como entrada apenas a mensagem a ser processada. São exemplos deste tipo de sistema funções de sentido único, funções de espalhamento e geradores de bits aleatórios (OPPLIGER, 2009, p. 28). Para os fins deste trabalho, é de especial relevância as funções de espalhamento.

Funções de espalhamento ou de *hashing* são funções que recebem uma entrada de tamanho variável e geram uma saída de tamanho fixo por meio de um algoritmo computacionalmente eficiente (OPPLIGER, 2009, p. 30). De maneira formal, sendo $\sum_{in}$ um alfabeto de entrada e $\sum_{out}$ um alfabeto de saída, define-se uma função de espalhamento h como:

$$h : \sum\nolimits_{in}^{*} \rightarrow \sum\nolimits_{out}^{n} \tag{1}$$

Usualmente $\sum_{in}$ e $\sum_{out}$ são o mesmo alfabeto e, por vezes, o alfabeto utilizado é o binário $\{0, 1\}$.

Na computação em geral, este tipo de função é utilizada no armazenamento de informações em uma estrutura de dados chamada de *tabela hash*. Neste uso, dado x e y, sendo x um rótulo[1] e y o dado a ser armazenado, o par (x, y) é colocado na posição $h(x)$ da tabela. Desta forma y pode ser recuperado eficientemente se x for conhecido e h puder ser computado eficientemente (OPPLIGER, 2009, p. 63)[p. 30]. Caso existam dois rótulos, x e x', tais que $h(x) = h(x')$, diz-se que ocorreu uma colisão (VAUDENAY, 2006).

Na criptografia, funções de espalhamento são utilizadas para garantir a integridade dos dados. Se é necessário garantir que um grande bloco de informações não foi modificado mas não é requisitado a confidencialidade da informação, então é possível transmitir o grande bloco por um canal não se-

[1] **Alguns autores utilizam o termo "chave" ou ainda "chave única" para se referir ao índice de uma tabela *hash*. No contexto de criptografia entretanto, o termo "chave" pode remeter a uma informação secreta e gerar algum tipo confusão, sendo preferível neste caso a utilização de termos como rótulo ou *tag***

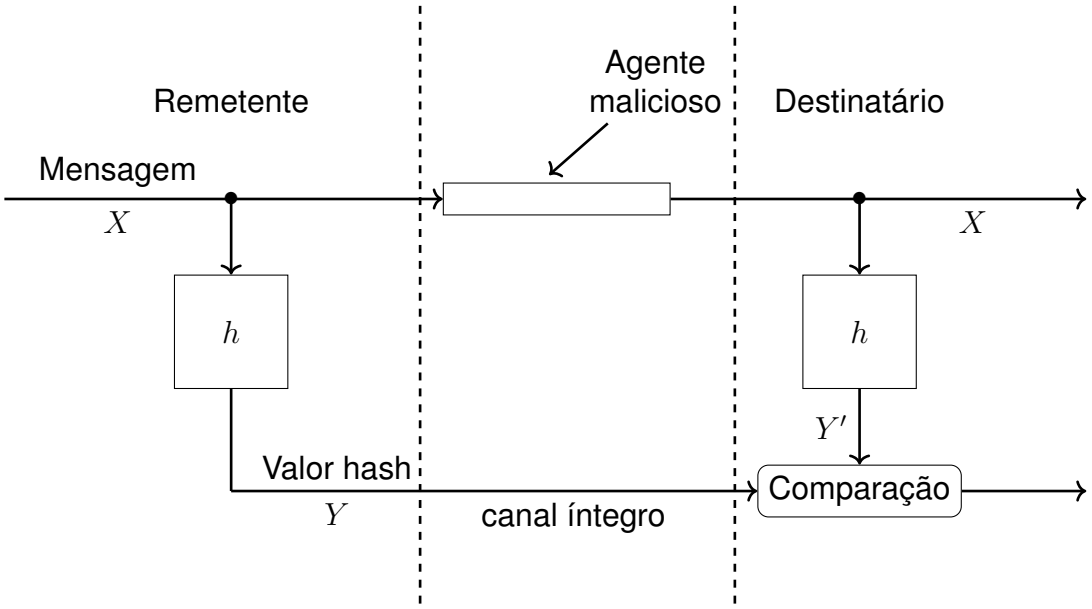

Figura 1: Utilização de funções de espalhamento para garantir a integridade dos dados

Fonte: adaptado de (VAUDENAY, 2006, p. 64)

guro e o resultado da aplicação de uma função de espalhamento por um canal seguro. O resultado da aplicação de uma função de espalhamento sobre um bloco de informações é chamado de valor *hash*, *digest* ou de impressão digital (VAUDENAY, 2006, p. 63-64).

Ao receber o bloco de informações e o valor de *hash*, o destinatário pode aplicar novamente a mesma função de espalhamento e verificar se a mensagem foi modificada ou não, como ilustrado pela Figure 1. Três características devem ser observadas na função de espalhamento:

- Resistência à pré-imagem:

  Para um $y \in \sum_{out}^{n}$ escolhido aleatoriamente, deve ser computacionalmente difícil encontrar um $x \in \sum_{in}^{*}$ tal que $h(x) = y$

- Resistência à segunda pré-imagem:

Para um dado $x \in \sum_{in}^{*}$ escolhido aleatoriamente, deve ser computacionalmente difícil encontrar um $x' \neq x$ tal que $h(x) = h(x')$.

- Resistência à colisão:

  É computacionalmente difícil encontrar quaisquer dois valores $x, x' \in \sum_{in}^{*}$ em que $x \neq x'$ e $h(x) = h(x')$

Para que a função seja considerada uma função de espalhamento criptográfica ela deve ser pelo menos resistente a pré-imagem ou à colisão. É importante também notar que apenas a integridade da mensagem é garantida, mas não a autenticidade do remetente.

2.1.2 SISTEMAS CRIPTOGRÁFICOS DE CHAVE SECRETA

Sistemas criptográficos com chave secreta utilizam parâmetros secretos que são compartilhados entre as entidades que participam da comunicação. Criptografia simétrica, códigos de autenticação de mensagens e geradores de bits pseudoaleatórios são exemplos deste tipo de sistema (OPPLIGER, 2009, p. 35). Para este trabalho são relevantes os dois primeiros.

2.1.2.1 CRIPTOGRAFIA SIMÉTRICA

Enquanto as funções de espalhamento são utilizadas para garantir a integridade de uma comunicação, um sistema criptográfico busca garantir a confidencialidade das informações. Oppliger (2009) define um sistema criptográfico simétrico como dois conjuntos de funções:

$$E = \{E_k : k \in K\} \tag{2}$$

$$D = \{D_k : k \in K\} \tag{3}$$

Onde cada uma das funções E_k e D_k são definidas como:

$$E_k : M \rightarrow C \tag{4}$$

$$D_k : C \rightarrow M \tag{5}$$

Sendo K o domínio de todas as possíveis chaves, M o domínio das mensagens em texto plano (isto é, cujo conteúdo semântico está diretamente disponível[2] (SHIREY, 2000)) e C o domínio das mensagens em texto cifrado. Por fim, para que o sistema seja simétrico, para cada $k \in K$ e $m \in M$ as funções D_k devem ser inversas das funções E_k, isto é:

$$D_k\left(E_k\left(m\right)\right) = m \tag{6}$$

Tipicamente, $M = C = \{0,1\}^*$ e $K = \{0,1\}^l$, com l fixo (por exemplo, $l = 128$). Também é comum que a ordem em que as funções seja indiferente (OPPLIGER, 2009, p. 36), isto é:

$$D_k\left(E_k\left(m\right)\right) = E_k\left(D_k\left(m\right)\right) = m \tag{7}$$

Desta forma, é possível transformar um canal de comunicação inseguro em um canal confidencial a partir de um outro canal confidencial e de um sistema criptográfico simétrico (VAUDENAY, 2006, p. 21), como mostra a Figura 2. Este canal seguro extra é utilizado para o compartilhamento da chave e geralmente é um canal cuja comunicação é muito mais cara ou não está sempre disponível.

Uma característica a ser levada em consideração a respeito das funções E_k e D_k é o tamanho da entrada que estas recebem. Uma cifra é chamada de bloco se opera somente sobre grupos de tamanho fixo de caracteres de en-

[2] **Na verdade, a RFC2828 utiliza esta definição para "texto claro", que por simplicidade é aqui utilizado como um sinônimo. Uma definição mais precisa de texto plano seria "dados de entrada para um processo criptográfico ou de saída de um processo de descriptografia", considerando que os dados de entrada de uma operação criptográfica podem ser o resultado de outra operação criptográfica**

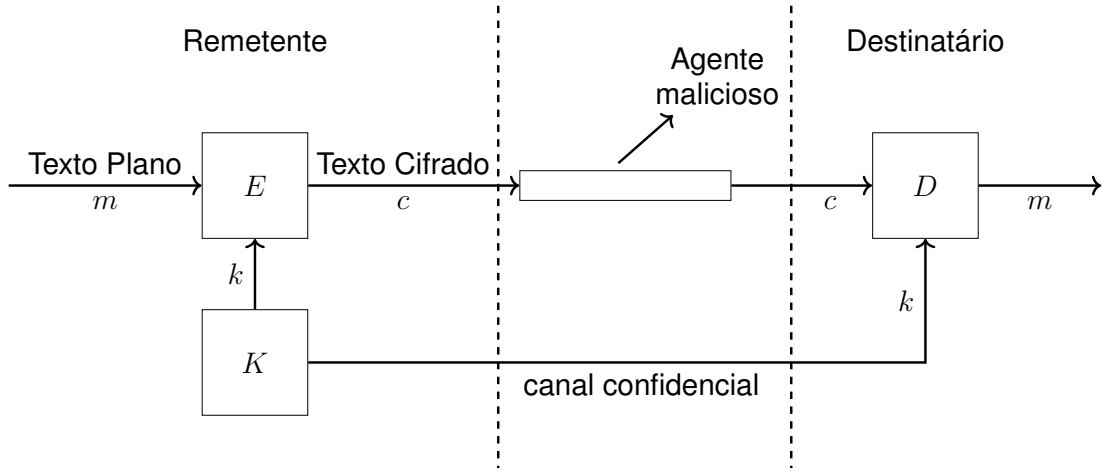

Figura 2: Utilização de um sistema criptográfico simétrico para garantir a confidencialidade dos dados.

Fonte: adaptado de (VAUDENAY, 2006, p. 22)

trada, enquanto uma cifra chamada de fluxo opera os caracteres de entrada um a um (OPPLIGER, 2009, p. 36-37). Exemplos práticos de cifras com estas características são os algoritmos *Advanced Encryption Standard* (AES) e RC4 que descrevem, respectivamente, uma cifra de bloco e uma cifra de fluxo.

Para uma cifra de bloco ser aplicada sobre um dado menor que um bloco, se faz necessário estender o dado até o tamanho do bloco, geralmente com a adição de zeros. Para a aplicação de uma cifra de bloco em uma mensagem maior que o bloco, um modo de operação deve ser adotado, como o *electronic codebook* (ECB) ou o *cipher-block chaining* (CBC). Uma cifra de bloco também pode ser transformada em uma cifra de fluxo pela aplicação de modos de operação como *cipher feedback* (CFB) ou *output feedback* (OFB) (OPPLIGER, 2009, p. 37).

2.1.2.2 CÓDIGOS DE AUTENTICAÇÃO DE MENSAGENS

Em algumas aplicações a confidencialidade de uma mensagem não é necessária ou mesmo permitida, como por exemplo em sistemas cujas men-

sagens são auditadas por um agente que não participa da comunicação e portanto não possui as chaves utilizadas. Ainda assim a autenticidade e integridade das mensagens pode ser necessária e para este fim o remetente pode utilizar uma etiquetas de autenticação.

Assim como o valor *hash* que garante a integridade de uma mensagem é a saída de uma função de espalhamento, uma etiqueta de autenticação é utilizada para garantir a autenticidade de um bloco de dados e pode ser obtida por meio de um código de autenticação de mensagens (em inglês, *Message Authentication Code*, MAC).

A Figura 3 apresenta a utilização de um MAC para garantir a autenticidade de uma mensagem transmitida por um canal inseguro. Como no caso da criptografia simétrica, um canal seguro adicional se faz necessário para a negociar a chave secreta e em seguida a mensagem e sua etiqueta de autenticação podem ser transmitidos pelo canal inseguro. Novamente, a comunicação por este canal seguro adicional pode ser muito mais cara, ou o canal pode estar disponível apenas no início da comunicação (VAUDENAY, 2006, p. 79).

Formalmente, podemos definir um MAC como dois conjuntos de funções:

$$A = \{A_k : k \in K\} \tag{8}$$

$$V = \{V_k : k \in K\} \tag{9}$$

Onde cada uma das funções A_k e V_k são definidas como:

$$A_k : M \to T \tag{10}$$

$$V_k : M \times T \to \{válido, inválido\} \tag{11}$$

Onde M é o domínio de todas as possíveis mensagens, T é o domínio de todas as possíveis etiquetas de autenticação, K é o domínio de todas as possíveis

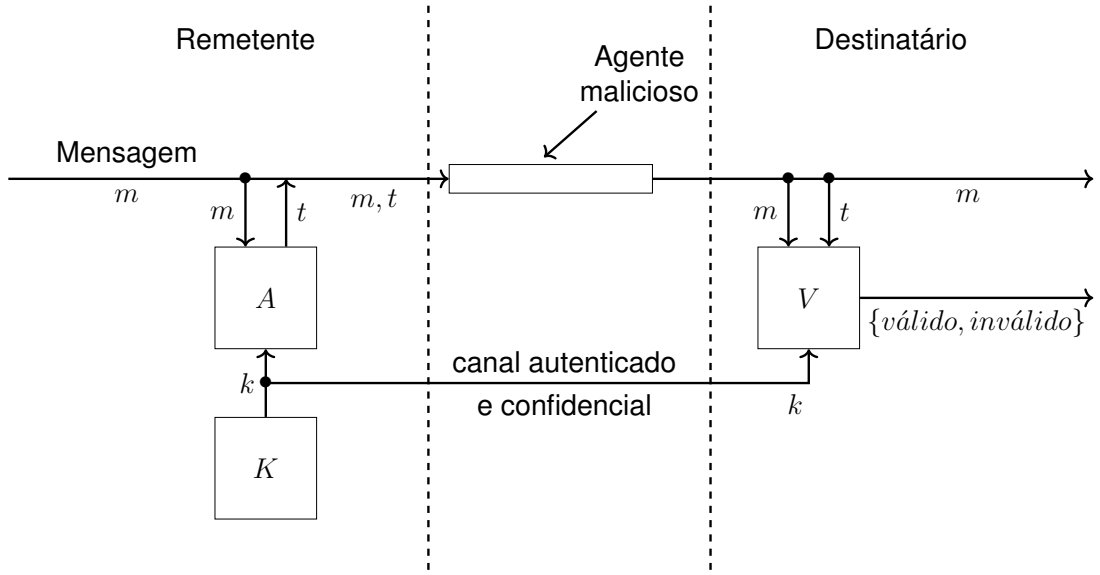

Figura 3: Emprego de MAC para garantir autenticidade de uma mensagem.

Fonte: adaptado de (VAUDENAY, 2006, p. 79)

chaves. Adicionalmente, temos que para toda mensagem $m \in M$, chave $k \in K$ e etiqueta $t \in T$:

$$V_k(m,t) = válido \iff t = A_k(m) \tag{12}$$

Tipicamente, $M = \{0,1\}^*$, $T = \{0,1\}^{l_{tag}}$ para uma etiqueta de tamanho l_{tag} e $K = \{0,1\}^{l_{key}}$ para uma chave de tamanho l_{key}. Também é comum que $l_{tag} = l_{key}$.

Existem diversos algoritmos utilizados para a construção de um MAC. Uma construção famosa é chamada de CBC-MAC, que consiste em utilizar uma cifra de blocos operada em modo CBC e utilizar apenas seu último bloco como etiqueta de autenticação. Alguns cuidados devem ser tomados caso a mensagem seja de tamanho variável, e para esse fim outros algoritmos como XCBC, RMAC e TMAC foram propostos. Outra construção baseado em CBC-MAC é o ECBC-MAC, ou EMAC, que propõem que o último bloco seja cifrado com uma segunda chave antes de ser utilizado como etiqueta (VAUDENAY, 2006, p. 80-81).

Uma construção MAC largamente aplicada em protocolos de segurança de rede é o *hashed* MAC (HMAC) (OPPLIGER, 2009, p. 43). Como seu nome sugere, HMAC é um mecanismo para autenticação de mensagens baseado em funções de espalhamento criptográficas iterativas. A força desta construção MAC depende das propriedades da função de espalhamento escolhida (KRAWCZYK *et al.*, 1997).

Os principais objetivos desta construção são a aplicação das funções de espalhamento sem modificá-las, preservar o desempenho da função de espalhamento, lidar de maneira simples com as chaves, permitir uma fácil análise da força desta construção e permitir facilmente a substituição da função de espalhamento utilizada (KRAWCZYK *et al.*, 1997).

A construção é definida da seguinte forma:

$$HMAC_k\left(m\right) = h\left(k \oplus opad \| h\left(k \oplus ipad \| m\right)\right) \tag{13}$$

Sendo k a chave secreta escolhida aleatoriamente ou por meio de um algoritmo pseudoaleatorio criptográfico alimentado por uma semente aleatória, m a mensagem, h a função de espalhamento iterativa, $\oplus$ a operação lógica "ou exclusivo" (XOR) aplicada bit a bit, $\|$ a concatenação de cadeias de caracteres, $opad$ e $ipad$ duas constantes, respectivamente 0x5C e 0x36, repetidas B vezes, sendo B o tamanho do bloco utilizado para iteração da função de espalhamento (KRAWCZYK *et al.*, 1997). Os valores de B para as funções de espalhamento MD5, SHA-1 e da família SHA-2 podem ser vistos no Quadro 1.

A construção também define que se a chave k for maior do que B bytes, então ela deve ser primeiramente substituída por $h\left(k\right)$. Isto é necessário para as operações XOR envolvidas e, como decorrência, o uso de chaves maiores não aumenta significativamente a segurança da construção. O uso de uma chave maior é recomendado caso a aleatoriedade da escolha da chave seja considerada fraca. Recomenda-se também que a chave não seja menor que o tamanho L da saída da função de espalhamento (KRAWCZYK *et al.*, 1997).

Alguns exemplos de valores de L estão no Quadro 1.

Quadro 1: Tamanho do bloco (B) e tamanho da saída (L) para as funções de espalhamento MD5, SHA-1 e da família SHA-2

Função de espalhamento	B	L
MD5	64	16
SHA-1	64	20
SHA-224	64	28
SHA-256	64	32
SHA-384	128	48
SHA-512	128	64
SHA-512/224	128	28
SHA-512/256	128	32

Fonte: (National Institute of Standards and Technology, 2002, p. 3)

A saída do HMAC pode ser truncada, a fim de reduzir a quantidade de informações que um atacante teria sobre o valor *hash*, ao custo de existirem menos bits a serem preditos por um atacante. Para tanto a construção deve utilizar os t mais a esquerda do resultado da Equação 13, com t não menor que metade de L para a função de espalhamento utilizada.

Por fim, mesmo que uma função de espalhamento seja considerada insegura para garantir a integridade de um bloco de informações, como ocorre com as funções SHA-1 (BETAFRED, 2017; THE..., 2017; A..., 2017) e MD5 (GOLEMON, 2017; BETAFRED, 2013), um HMAC que as utilize não é necessariamente inseguro (RISTIC, 2014, p. 51). Isso se deve ao fato de que estes algoritmos são considerados fracos em relação a resistência a colisões, uma característica que não é utilizada como premissa para a prova da segurança da construção HMAC (BELLARE, 2006).

2.1.3 SISTEMAS CRIPTOGRÁFICOS DE CHAVE PÚBLICA

Em um sistema criptográfico de chave pública, cada entidade participante da comunicação possui seu próprio conjunto de parâmetros secretos, denominados de chave privada, que não é compartilhado com nenhuma das demais entidades e um conjunto de parâmetros não-secretos, denominados de chave pública, que devem ser compartilhados abertamente. São exemplos deste tipo de sistema os protocolos para estabelecimento de chaves, sistemas de assinatura digital e criptografia assimétrica (OPPLIGER, 2009, p. 45).

A base deste tipo de sistema é a aplicação de uma função de sentido único que possua uma função armadilha (também chamada de função arapuca) que permita sua inversão. A função de sentido único baseia-se na chave pública, de forma que qualquer um possa utilizá-la, enquanto a função armadilha utiliza a chave privada, de forma que apenas a entidade dona do par de chaves possa inverter a função de sentido único (VAUDENAY, 2006, p. 299).

Um requisito necessário, porém insuficiente, para que este tipo de sistema seja seguro é que deve ser computacionalmente inviável a obtenção da chave privada a partir da chave pública, a fim de que esta possa ser compartilhada abertamente (OPPLIGER, 2009, p. 46).

Também se faz válido citar a existência de poucas funções de sentido único conhecidas que possam ser utilizadas neste tipo de sistema (VAUDENAY, 2006, p. 231), e que geralmente um sistema criptográfico de chave pública é computacionalmente menos eficiente do que um sistema criptográfico de chave secreta (OPPLIGER, 2009, p. 46). Usualmente, este tipo de sistema é utilizado para autenticação e gerenciamento de chaves, resultando em um sistema criptográfico híbrido, que combina o uso de chaves secretas e chaves públicas.

2.1.3.1 CRIPTOGRAFIA ASSIMÉTRICA

Assim como a criptografia simétrica, os algoritmos de criptografia assimétrica visam garantir a confidencialidade das informações. A principal diferença, entretanto, é o emprego de um sistema criptográfico de chave pública (OPPLIGER, 2009, p. 46), que requisita a existência de um canal extra autenticado, no lugar de um canal confidencial. Vaudenay (2006) define um sistema de criptografia assimétrica constituído por três elementos:

- Um gerador pseudoaleatório de chaves G: um algoritmo probabilístico que gera um par de chaves (k, k^{-1}), sendo k a chave pública e k^{-1} a chave privada.
- Um algoritmo de encriptação E: um algoritmo, possivelmente probabilístico, que toma como entrada uma chave pública k e um texto plano m e tem como saída um texto cifrado c.
- Um algoritmo de desencriptação D: um algoritmo, necessariamente determinístico, que toma como entrada uma chave privada k^{-1} e um texto cifrado c e tem como saída um texto plano m.

Ainda existindo os seguintes requisitos:

- Para qualquer par de chaves (k, k^{-1}) gerados por G, qualquer texto plano m e qualquer possível saída c de $E_k(x)$, necessariamente $D_{k^{-1}}(c)$ deve ser igual a m, ou seja, $D_{k^{-1}}(E_k(m)) = m$. Adicionalmente, se as chaves k e k^{-1} não forem correspondentes, a saída de D não deve possuir qualquer significado.
- Dado k e c, sendo $E_k(m) = c$, deve ser computacionalmente inviável recuperar m.

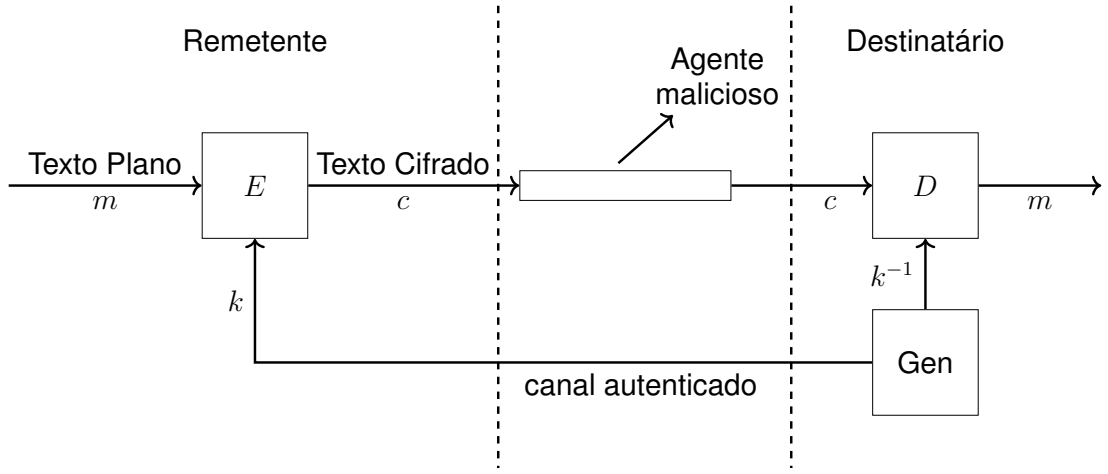

Figura 4: Utilização de um sistema criptográfico assimétrico para garantir a confidencialidade dos dados

Fonte: adaptado de (VAUDENAY, 2006, p. 229)

Como mencionado anteriormente, a partir da aplicação de um sistema criptográfico assimétrico e de um canal autenticado é possível transformar um canal inseguro em um canal confidencial. Este canal extra é necessário para a transmissão da chave pública, e sua característica de autenticidade é requerida para impedir ataques do tipo *Man-In-the-Middle* (VAUDENAY, 2006, p. 233). A Figura 4 ilustra este tipo de aplicação.

2.1.3.2 SISTEMAS DE ASSINATURA DIGITAL

Assim como a alternativa de chave pública da criptografia simétrica é a criptografia assimétrica, a alternativa ao uso de MACs é a assinatura digital. (VAUDENAY, 2006, p. 254). Um sistema de assinatura digital tem o objetivo de garantir a autenticidade, e consequentemente a integridade, de uma mensagem. Shirey (2000) define assinatura digital como “Dados anexados a, ou uma transformação criptográfica de, uma unidade de dados que permite a um destinatário da unidade de dados provar a fonte e a integridade da unidade de dados e proteger contra falsificação”.

Quando a assinatura digital é feita pela anexação de dados ela é dita uma “assinatura digital com apêndice” e quando é feita por uma transformação criptográfica, trata-se de uma “assinatura digital com recuperação da mensagem” (OPPLIGER, 2009, p. 50). Para os fins deste trabalho, o foco será mantido na primeira.

Vaudenay (2006) define um esquema de assinatura digital com três elementos:

- Um gerador pseudoaleatório de chaves G, que gera as chaves (k, k^{-1}).
- Um algoritmo de assinatura S, possivelmente probabilístico, que gera uma assinatura s para cada mensagem m utilizando uma chave secreta k^{-1}.
- Um algoritmo de verificação V, necessariamente determinístico, que dado uma mensagem m, uma assinatura s e uma chave pública k, verifica se a assinatura é válida.

A definição do sistema criptográfico RSA propõem a utilização do algoritmo para um esquema de assinatura digital. Entretanto, a relação entre sistemas criptográficos assimétricos e sistemas de assinatura digital é bastante genérica, e não específica a este algoritmo (VAUDENAY, 2006, p. 255).

Considerando a existência de um sistema de criptografia assimétrica, podemos utilizar o algoritmo de desencriptação D como o algoritmo de assinatura e o algoritmo de encriptação E como o algoritmo de verificação, desde que E seja determinístico (VAUDENAY, 2006, p. 255).

Na prática, o tamanho da mensagem m a ser assinada pode ser muito longo, tornando mais apropriado a aplicação da assinatura sobre a saída de uma função de espalhamento h (OPPLIGER, 2009, p. 50). Esta função de espalhamento, entretanto, deve ser resistente à colisões, para reduzir as chances de duas mensagens diferentes possuírem a mesma assinatura e evitar alguns tipos de ataques (VAUDENAY, 2006, p. 256).

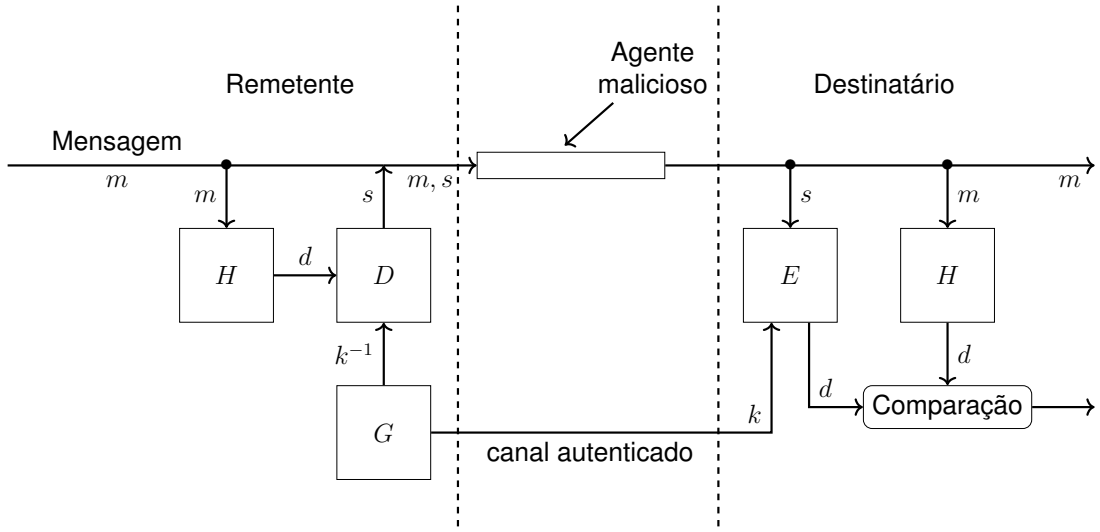

Figura 5: Utilização de criptografia assimétrica e função de espalhamento para um sistema de assinatura digital a fim de garantir autenticidade de uma mensagem.

Fonte: adaptado de (VAUDENAY, 2006, p. 255)

2.2 O PROTOCOLO MODBUS

O protocolo Modbus foi criado pela Modicon, atual Schneider Eletric, em 1979 e define uma estrutura de mensagens para comunicações mestre-escravo entre dispositivos inteligentes (Modbus FAQ, 2017b). É um protocolo aberto, sua especificação é distribuída gratuitamente[3] e não há taxas de licenciamento. A Modicon foi comprada pela Schneider Electric, que em abril de 2004 transferiu o protocolo para a Modbus Organization, uma organização independente, sem fins lucrativos, que gerencia a evolução do protocolo atualmente (Modbus FAQ, 2017a).

O protocolo foi originalmente projetado para redes seriais assíncronas, como RS-232 e RS-485, com dois modos de transmissão, RTU e ASCII, sendo obrigatória apenas a implementação do primeiro (Modbus Org, 2006). Um quadro Modbus para redes seriais consiste em um único PDU (do inglês *Pro-*

[3] `http://www.modbus.org/specs.php`

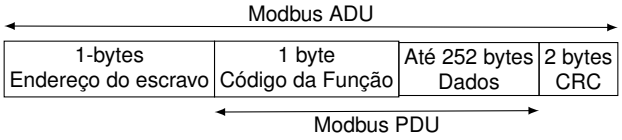

Figura 6: Quadro Modbus para transmissão por redes seriais no mode RTU

Fonte: Adaptado de (Modbus Org, 2006)

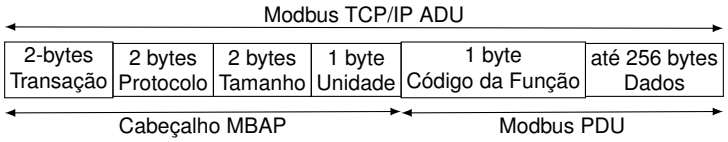

Figura 7: Quadro Modbus TCP

Fonte: Adaptado de (Schneider Automation, 2006)

tocol Data Unit, unidade de dados de protocolo) Modbus contendo um ADU (do inglês *Application Data Unit*, unidade de dados de aplicação) Modbus. Sua construção para o modo RTU é ilustrada na Figura 6.

Visando as vantagens do padrão Ethernet, como escalabilidade, desempenho de 10/100 Mbps e 1 Gbps e a facilidade para a integração com outros sistemas, uma versão para redes TCP/IP foi desenvolvida em 1999 (Modbus FAQ, 2017b).

A Figura 7 mostra um quadro Modbus/TCP (isto é, Modbus sobre TCP/IP). O endereço do escravo é omitido, pois a rede IP já endereça seus nós. O campo “Unidade” foi adicionado para possibilitar a comunicação por dispositivos como *proxies*, *gateways* e roteadores que utilizam um único endereço IP. O campo de verificação cíclica de redundância (do inglês, *Cyclic Redundancy Check*, CRC) também é omitido pois a verificação por erros já é realizada pelo protocolo TCP (Schneider Automation, 2006).

O campo “Transação” é utilizado para emparelhar as transações, todo quadro de resposta Modbus/TCP possuí o mesmo número identificador de transação da requisição que o gerou. O campo “Protocolo” é utilizado para multiplexar diferentes protocolos entre sistemas e é sempre zero para Modbus/TCP

(Schneider Automation, 2006). O campo de tamanho é utilizado para delimitar o tamanho do quadro, pois TCP é um protocolo de fluxo de dados e portanto não preserva os limites das mensagens enviadas. A porta TCP 502 é reservada para a operação do protocolo Modbus/TCP (TOUCH *et al.*, 2018).

Para qualquer um dos modos de transmissão, o Protocolo de Aplicação Modbus especifica a camada de aplicação de dispositivos Modbus. Esta camada é definida por um protocolo do tipo requisição-resposta onde um dispositivo escravo oferece serviços identificados por códigos de funções, na faixa de 1 a 127. Alguns códigos de funções são apresentados no Quadro 3.

O modelo de dados do protocolo é definido por quatro tabelas principais com diferentes propriedades, que são apresentadas no Quadro 2. Para cada tabela o protocolo permite até 65.535 elementos, que podem representar o estado lógico de chaves e válvulas para entradas discretas e bobinas, o valor analógico lido por um sensor para registradores de entradas ou algum parâmetro do sistema de controle para os registradores comuns.

Um dispositivo atuando como mestre do protocolo é comumente chamado MTU (do inglês *Master Terminal Unit*, Unidade Terminal Mestre), enquanto um dispositivo atuando como escravo é referenciado como um RTU (do inglês *Remote Terminal Unit*, Unidade Terminal Remota).

Quadro 2: Modbus Data Modbus primary tables

Tabela	Tamanho	Tipo de Acesso
Entradas Discretas	Um bit	Somente Leitura
Bobinas	Um bit	Leitura e Escrita
Registradores de Entrada	16-bit	Somente Leitura
Registradores	16-bit	Leitura e Escrita

Quadro 3: Códigos de Funções Modbus

0x01	Ler Bobina	0x07	Ler Código de Erro
0x02	Ler Entrada Discreta	0x0F	Escrever em Múltiplas Bobinas
0x03	Ler Registrador	0x10	Escrever em Múltiplos Registradores
0x04	Ler Registrador de Entrada	0x11	Reportar Endereço dos Escravos
0x05	Escrever em Bobina	0x17	Ler/Escrever Múltiplos Registradores

Fonte: adaptado de (MODBUS, 2006)

2.3 O PROTOCOLO TRANSPORT LAYER SECURITY

Os protocolos *Secure Socket Layer* (SSL) e *Transport Layer Security* (TLS) são protocolos da camada de transporte que buscam gerar uma comunicação segura no formato cliente/servidor para aplicações baseadas em TCP/IP (OPPLIGER, 2009, p. 75-78).

O protocolo SSL foi desenvolvido inicialmente dentro da Netscape Communications, entre 1993 e 1994, tendo sua versão inicial, o SSL 1.0, circulando apenas internamente devido a diversos problemas de segurança (OPPLIGER, 2009, p. 68-69). Estes problemas foram solucionados na segunda versão, o SSL 2.0, que foi liberada publicamente no navegador Netscape e em outros produtos da empresa. Em 1995 o protocolo foi submetido como um *Internet-Draft* e como uma patente para a empresa, com a intenção de liberá-la à comunidade gratuitamente (OPPLIGER, 2009, p. 69).

A terceira versão do protocolo contou com a consultoria de diversos profissionais de segurança de relevância da época e foi lançada em 1996. A fim de facilitar a adoção por concorrentes da empresa, o protocolo foi entregue à *Internet Engineering Task Force* (IETF), que realizou algumas melhorias e liberou o protocolo com o nome de *Transport Layer Security* em 1999 (OPPLIGER, 2009, p. 69-70).

Atualmente diversos protocolos de aplicação têm sua segurança ba-

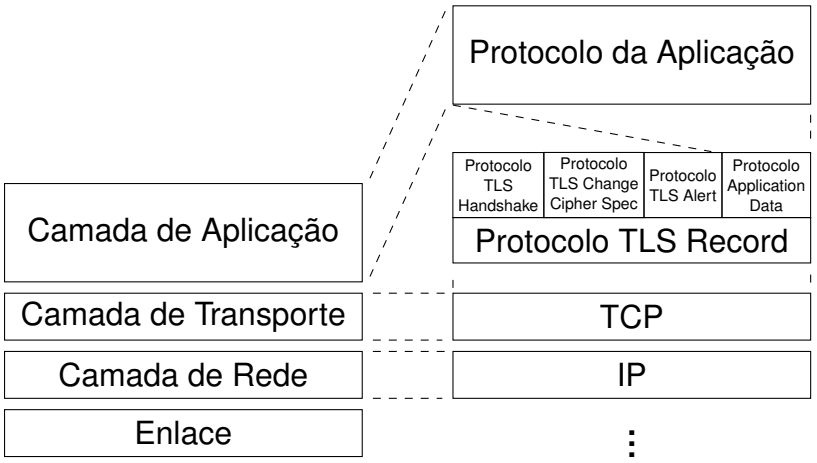

Figura 8: Subprotocolos e camadas TLS

seada em SSL/TLS como HTTPS (RESCORLA, 2000), (SMTP) (HOFFMAN, 1999), (SIP) (AUDET, 2009; ROSENBERG *et al.*, 2002) e algumas implementações de VPN (KOTULIAK *et al.*, 2011). Existe ainda diversas bibliotecas que implementam o protocolo como a OpenSSL (OpenSSL Management Committee, 2017), BoringSSL (Google, 2017), LibreSSL (OpenBSD Foundation, 2017), GnuTLS (Nikos Mavrogiannopoulos, 2017), mbedTLS (ARM Limited., 2017) e a wolfSSL (wolfSSL Inc., 2017).

O protocolo pode ser colocado na sexta camada do modelo OSI (RISTIC, 2014) ou entre a camada de aplicação e de transporte (OPPLIGER, 2009, p.67,76-77) do modelo Internet e é subdividido em duas camadas e cinco "subprotocolos", como ilustrado na Figura 8.

Uma associação entre dois pares comunicantes do protocolo é representada por uma Sessão TLS (OPPLIGER, 2016), que define um conjunto de parâmetros criptográficos que pode ser compartilhado por múltiplas conexões (DIERKS; RESCORLA, 2008):

- O Identificador de Sessão, um valor arbitrário gerado pelo servidor para identificar as sessões;
- O Certificado do outro par da conexão, se necessário;

- O Método de Compressão, isto é, o algoritmo usado para comprimir os dados antes da encriptação;
- A Especificação da Suíte Criptográfica, compreendida por uma função pseudoaleatória, um algoritmo para encriptação de dados, um algoritmo MAC e seus atributos, como o tamanho do MAC.
- O *Master Secret*, uma chave secreta de 48 bytes compartilhada entre os pares comunicantes;
- E se a sessão pode ser resumida ou não.

2.3.1 PROTOCOLO TLS RECORD

O protocolo TLS Record está baseado sobre TCP (OPPLIGER, 2016). A estrutura de um quadro do protocolo TLS Record pode ser vista na Figura 9. Este protocolo é utilizado para fragmentar e transmitir os dados das camadas superiores, opcionalmente comprimindo, autenticado e encriptando o quadro antes da transmissão (OPPLIGER, 2016; RISTIC, 2014).

Por outro lado, o protocolo é responsável por remontar mensagens recebidas da camada de transporte e entregar para as camadas superior, desencriptando, verificando a autenticidade e descomprimindo se necessário (RISTIC, 2014; DIERKS; RESCORLA, 2008). Cada quadro do protocolo pode transportar múltiplas mensagens de uma camada superior, desde que todas essas mensagens pertençam ao mestre subprotocolo TLS.

Uma visão geral da operação do protocolo é apresentada na Figura 10. É importante ressaltar que os passos de compressão, autenticação e en-

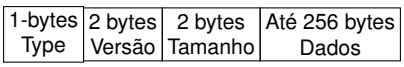

Figura 9: Quadro TLS Record

Fonte: Adaptado de (Schneider Automation, 2006)

criptação são opcionais. O passo de compressão em geral não é utilizado por questões de segurança (RISTIC, 2014, p. 207-219; OPPLIGER, 2016, p. 158-162).

As informações de quais algoritmos são utilizados para autenticar e encriptar o quadro são representadas pela suíte criptográfica em uso. Atualmente existem 339 suítes oficialmente suportadas (NIR *et al.*, 2018).

O nome de uma suíte indica o algoritmo de troca de chaves, o método de autenticação dos pares da conexão, o método criptográfico simétrico usado e seus parâmetros, a construção MAC e, a partir da versão 1.2 do protocolo, opcionalmente uma função pseudoaleatória (RISTIC, 2014, p. 50). Tomando `TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA` como exemplo a suíte criptográfica:

- `DH` refere-se a troca de chaves utilizando o algoritmo de Diffie-Hellman;
- `RSA` indica que uma assinatura RSA irá garantir a autenticidade das partes;
- `3DES_EDE_CBC` diz que os dados da comunicação serão encriptados com a cifra 3DES em modo CBC. O termo `EDE` é um parâmetro da cifra 3DES;
- E `SHA` informa que a construção HMAC será baseada na função de espalhamento SHA-1.

Diferentes suítes criptográficas podem ter diferentes objetivos de segurança. Algumas suítes, como a `TLS_ECDH_anon_WITH_AES_128_CBC_SHA`, abrem mão da autenticação dos pares em favor da anonimidade, trazendo maior privacidade aos pares comunicantes.

Neste exemplo o algoritmo de Diffie-Hellman sobre curvas elípticas (`ECDH`) é utilizado para troca de chaves, nenhuma autenticação é feita, AES com uma chave de 128-bits em modo CBC é usado para encriptar as mensagem e novamente a construção HMAC é baseada em SHA-1.

Outras suítes trazem apenas a autenticação das partes e autenticidade das mensagem, como o caso da `TLS_ECDHE_RSA_WITH_NULL_SHA`. Aqui a troca de chaves é feita com a versão efêmera do algoritmo de Diffie-Hellman sobre curvas elípticas, uma assinatura RSA é utilizada para autenticação, os dados da comunicação são transmitidos em texto plano e uma construção MAC baseada SHA-1 garante a integridade e autenticidade das mensagens.

Este tipo de cifra pode ser necessária em ambientes como os descritos em 2.1.2.2 ou em casos onde o custo computacional adicionado pela encriptação das mensagens tornaria o uso do protocolo proibitivo.

Uma suíte especial é a `TLS_NULL_WITH_NULL_NULL`, que indica a não autenticação das partes, o uso de uma função identidade para encriptação e a inexistência da construção MAC. Esta suíte claramente não traz segurança alguma a comunicação e é utilizada, na verdade, no início de uma conexão TLS, onde as entidades envolvidas ainda não trocaram informações suficientes para estabelecer a conexão segura.

Com esta suíte é realizado o processo de negociação dos parâmetros da conexão, chamado de *handshake*, que é explicado a seguir.

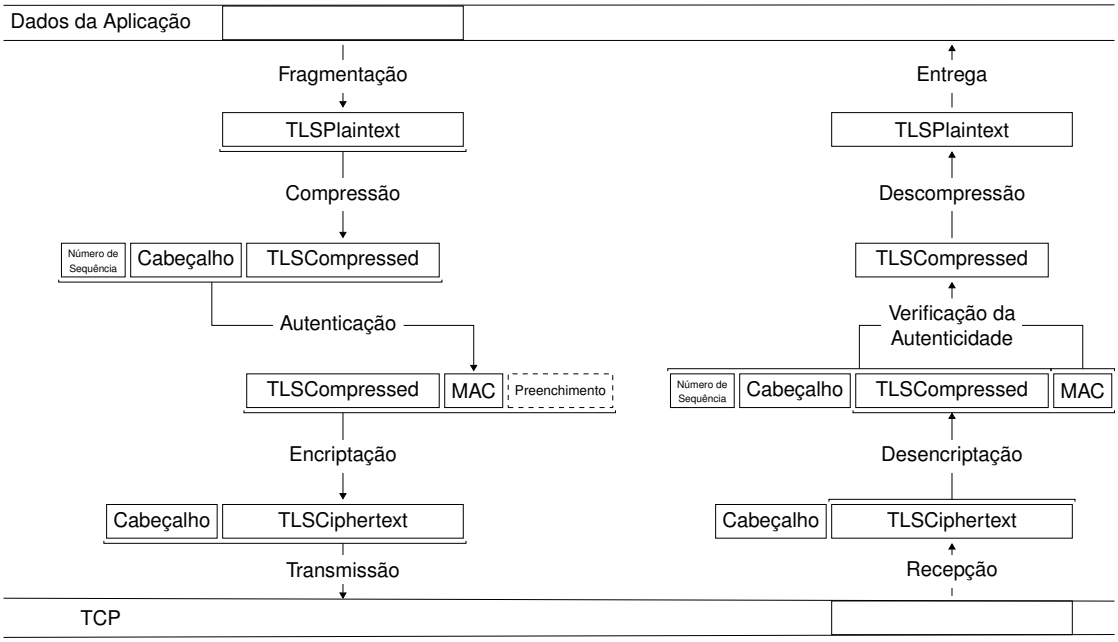

Figura 10: Visão geral da operação do protocolo TLS Record

2.3.2 PROTOCOLO TLS HANDSHAKE

O protocolo TLS Handshake é baseado no protocolo TLS Record e é o encarregado pelo processo de *handshake*, por meio do qual os pares comunicantes combinam os parâmetros da sessão TLS (DIERKS; RESCORLA, 2008). O processo é realizado pela troca de seis a treze mensagens, dependendo da configuração do cliente e do servidor (RISTIC, 2014), como ilustrado na Figura 11. Mensagens apresentadas em itálico são opcionais e mensagens entre colchetes não pertencem ao protocolo de *handshake*, mas sim ao protocolo TLS Change Cipher Spec.

A primeira mensagem do processo é do tipo `ClientHello`. Esta mensagem é mandada pelo cliente para o servidor para comunicar suas capacidades e preferências. Seu conteúdo compreende:

- A melhor versão do protocolo suportada pelo cliente;
- Um valor aleatório de 32 bytes, que é a contribuição do cliente em dados

aleatórios para o *handshake*;

- O identificador de uma sessão anterior, se o cliente deseja resumir uma sessão anteriormente negociada;
- A lista de suítes criptográficas suportadas, ordenadas por preferência;
- A lista de métodos de compressão suportadas, também ordenados por preferência;
- E a lista de extensões do protocolo que são suportadas.

A segunda mensagem de *handshake* é uma resposta do servidor para o cliente, do tipo `ServerHello`. Seu conteúdo é muito similar ao de uma `ClientHello`, exceto por possuir apenas a suíte criptográfica e o método de compressão escolhidos, ao invés de uma lista (OPPLIGER, 2016).

Uma mensagem do tipo `Certificate` é usada para transportar cadeias de certificados X.509 do servidor ou cliente para o outro par da conexão. Estes certificados são codificados no padrão ASN.1 DER, como o certificado principal sendo o primeiro, seguido por todos os certificados intermediários até, mas sem incluir, um certificado raiz.

Nem todas as suítes criptográficas requem certificados, seja por serem suítes anônimas (que não possuem autenticação) ou porque o método de autenticação não depende de certificados. Se a suíte negociada necessitar, o servidor dará continuidade ao processo de *handshake* enviando uma mensagem do tipo `Certificate`.

Se o método de troca de chaves selecionado pela suíte criptográfica necessitar de mais informações do que as providas por certificados, o servidor enviará em seguida uma mensagem do tipo `ServerKeyExchange`, cujo conteúdo varia conforme o método de troca de chaves em uso.

Quando o servidor requer a autenticação do cliente, uma mensagem do tipo `CertificateRequest` é enviada. Esta mensagem contém uma lista de

tipos de certificados aceitos, tipos de algoritmos de assinatura suportados e autoridades certificadoras, identificadas por seus nomes distintos.

O servidor então sinaliza que enviou todas as mensagens de *handshake* que pretendia por meio de uma mensagem do tipo `ServerHelloDone`. Se uma mensagem do tipo `CertificateRequest` foi enviada, o servidor irá responder com uma mensagem do tipo `Certificate` com o requerido certificado. Se o método de troca de chaves necessitar de mais informações do cliente, uma mensagem do tipo `ClientKeyExchange` é enviada.

Se uma mensagem do tipo `Certificate` foi enviada, o cliente dará prosseguimento ao processo de *handshake* com uma mensagem do tipo `CertificateVerify`, que prova a posse da chave privada do certificado enviado por meio da assinatura de todas as mensagens de *handshake* enviadas até este ponto. Finalmente, o cliente envia uma mensagem `ChangeCipherSpec`, sinalizando que possui informações suficientes para gerar os parâmetros da conexão e chaves criptográficas necessárias e que agora está apto a utilizar o canal de comunicação de forma segura.

A mensagem `ChangeCipherSpec` não faz parte do protocolo TLS Handshake e sim a única mensagem do protocolo TLS Change Cipher Spec. Por ser parte de um subprotocolo diferente, esta mensagem necessariamente será transportada por um quadro TLS Record próprio e todos os quadros enviados a partir deste utilizarão os novos parâmetros negociados.

O cliente então envia sua última mensagem de *handshake*, do tipo `Finished`. Esta é a primeira mensagem encriptada e sinaliza o fim do processo de *handshake*. Seu conteúdo é o valor *hash* de todas as mensagens do protocolo TLS Handshake trocadas até este ponto, combinado com o *master secret* negociado por meio de uma função pseudoaleatória que pode variar de acordo com a versão do protocolo. Para a versão TLS 1.2, esta função é definida como (14) mostra. Trata-se de uma função iterativa que pode produzir uma quantidade arbitrária de dados e é definida em termos de P_{hash} e $A(i)$, apresentados

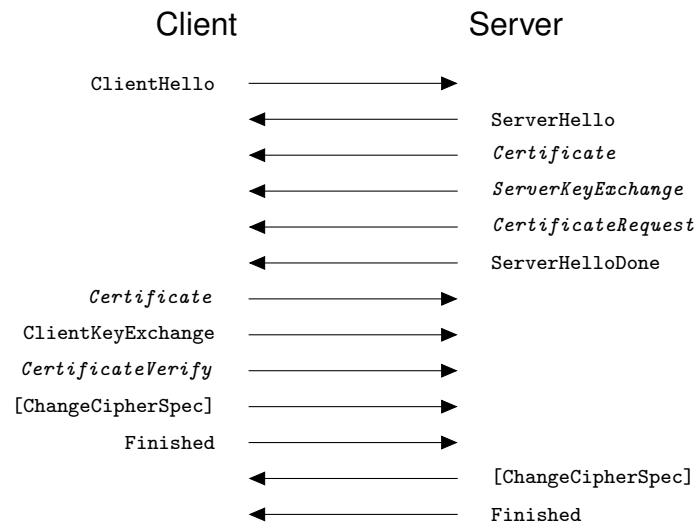

Figura 11: Processo de *handshake* do protocolo TLS

respectivamente em (15) e (16).

$$PRF\left(secret, label, seed\right) = P_{hash}\left(secret, label + seed\right) \tag{14}$$

$$\begin{aligned} P_{hash}\left(secret, seed\right) = {} & HMAC_{hash}\left(secret, A(1) + seed\right) + \\ & HMAC_{hash}\left(secret, A(2) + seed\right) + \\ & \ldots \end{aligned} \tag{15}$$

$$A\left(i\right) = \begin{cases} seed, & \text{if } i = 0 \\ HMAC_{hash}\left(secret, A\left(i-1\right)\right), & \text{if } i < 0 \end{cases} \tag{16}$$

Para gerar o conteúdo da mensagem `Finished`, PRF é invocada com o *master secrete* como $secret$, os bytes ASCII que representam a cadeia 'cliente finished' para a mensagem do cliente e 'server finished' par a mensagem do servidor como $label$ e o valor *hash* de todas as mensagem trocadas como $seed$.

O servidor então responde com uma mensagem `ChangeCipherSpec` e em seguida sua própria mensagem `Finished`, considerando a ultima mensa-

gem enviada pelo cliente e portanto com um valor *hash* diferente como entrada para a PRF. Sendo PRF uma função determinística, com os dados necessários para seu cálculo conhecido por ambos os lados e considerando que a mensagem é encriptada e autenticada com os parâmetros negociados para a sessão, seu conteúdo pode ser utilizado para verificar o sucesso da troca de chaves e do processo de autenticação.

Após este ponto, cliente e servidor podem trocar dados da aplicação por meio de um canal seguro. A qualquer momento o cliente pode enviar uma nova mensagem do tipo `ClienteHello` para indicar que deseja renegociar os parâmetros da sessão. Semelhantemente, o servidor pode requisitar a renegociação por meio do envio de uma mensagem do tipo `HelloRequest`.

2.3.3 PROTOCOLO TLS ALERT

O protocolo TLS Alert é usado para notificar o outro par da conexão sobre situações anormais e o fechamento da conexão. As mensagem deste protocolo são compostas por dois campos de um byte cada. O primeiro indica a severidade do alerta, com o valor 1 para avisos e 2 para alertas fatais.

Um par que envia ou recebe um alerta fatal deve imediatamente terminar a conexão atual e invalidar a sessão TLS. Outras conexões em andamento podem continuar abertas, mas novas conexões não poderão ser estabelecidas com a sessão invalidada. Quando um alerta de aviso é enviado, fica a cargo do receptor decidir tratar o evento como um erro fatal ou tomar outras ações.

O segundo byte carrega um código de descrição do alerta, como 20 para `bad_record_mac` quando um quadro TLS Record recebido tem um valor MAC inválido, ou 40 para `handshake_failure`, quando for impossível negociar um conjunto de parâmetros de segurança aceitável para ambas as partes.

Um importante alerta é o `close_notify`, representado pelo código de descrição zero. Este alerta é usado para notificar o outro par que não se

deseja enviar ou receber mais nenhuma mensagem e que a conexão será fechada. Este processo de fechamento da conexão é necessário para garantir ataques de truncamento, onde um agente malicioso encerra a comunicação de maneira abrupta. A falta deste alerta permite que ambas as partes da conexão detectem o problema.

2.4 CONSIDERAÇÕES

Este capítulo apresentou uma revisão dos conceitos básicos de criptografia, como os três tipos básicos de sistemas criptográficos (sem chave, de chave secreta e de chave pública) e suas aplicações para criptografia simétrica e assimétrica, algoritmos MAC e assinatura digital. Também foi apresentado um pequeno histórico e detalhado o funcionamento dos protocolos Modbus e TLS.

3 TRABALHOS RELACIONADOS

Os problemas de segurança relacionados a sistemas SCADA foram endereçados por inúmeros autores. O trabalho de Stouffer *et al.* (2011) apresenta um guia de segurança para sistemas de controle industrial em geral. Creery e Byres (2005) apresenta um método para a avaliação de riscos em sistemas de automação industrial, identificando possíveis problemas de segurança e as contramedidas cabíveis.

Especificamente ao protocolo Modbus, Huitsing *et al.* (2008) fez uma análise do protocolo a fim de caracterizar os possíveis ataques a sistemas que o utilizam. O trabalho identifica 33 taxonomias de ataques, sendo cinco destas específicas ao modo de transmissão serial e 13 específicas a Modbus TCP. Todos os ataques descritos dependem da existência de um analisador ou de um injetor de pacotes para a sua realização.

Fovino *et al.* (2009b) observaram o impacto dos *malware Code Red*, *Nimda*, *Slammer* e *Scalper* em sistemas SCADA que utilizam o protocolo Modbus e desenvolveram dois *malware* específicos para atacar dispositivos Modbus TCP. O primeiro deles realiza ataques DoS (do inglês *Denial of Service*, negação de serviço), injetando pacotes válidos do protocolo na rede a fim de consumir a largura de banda disponível sem ser detectado por um possível IDS (do inglês *Intrusion Detection System*, sistema de detecção de intrusão) que monitore a rede.

O segundo *malware* caracterizava-se como um *worm* que explorava a rede de novas máquinas infectadas em busca de escravos Modbus e, ao detectá-los, aplicava diversas heurísticas para prejudicar o funcionamento do sistema SCADA, como forçar todas as bobinas para um estado lógico, escrever

valores aleatórios em registradores ou inverter o estado lógico de bobinas.

Fovino *et al.* (2009a), Hayes e El-Khatib (2013) e Shahzad *et al.* (2015) propõem modificações ao protocolo Modbus para solucionar seus problemas de segurança. A solução dos dois primeiros é discutida nas seções a seguir:

3.1 A SOLUÇÃO DE FOVINO ET AL.

Fovino *et al.* (2009a) propôs uma solução que visa inserir um pequeno número de mecanismos de segurança ao protocolo. Nessa implementação, a confidencialidade dos dados é negligenciada em favor de uma solução mais barata computacionalmente, evitando impactos no desempenho de sistemas que trabalham em tempo real.

A solução apresentada adiciona uma estampa de tempo ao início do cabeçalho e concatena o valor *hash* da mensagem encriptado por um algoritmo de criptografia assimétrica ao fim do quadro. Como visto na seção 2.1.3.2, esta é uma das construções de um sistema de assinatura digital com apêndice e sua operação está ilustrada na Figura 5.

Um sistema de assinatura digital endereça os problemas de integridade e autenticidade e a estampa de tempo impede ataques de repetição de pacote, onde um atacante observa passivamente os pacote que trafegam pela rede, sabendo ou não o seu significado, e os repete em algum momento futuro. Ao processar uma mensagem, o destinatário considera uma janela de tempo em que a mensagem é válida.

Utiliza-se como função de espalhamento uma das funções da família SHA2 e o algoritmo criptográfico assimétrico empregado é o RSA. Uma representação de um quadro Modbus com esta solução pode ser vista na Figura 12. A estampa de tempo utilizada é proveniente de um servidor NTP (do inglês *Network Time Protocol*, protocolo de horário de rede) que deve estar na mesma

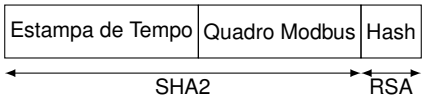

Figura 12: Proposta de implementação segura por Fovino et al.

rede do sistema SCADA e seu formato é de um inteiro de 64-bits (MILLS *et al.*, 2010).

Os resultados experimentais obtidos mostram um aumento negligenciável na latência do protocolo pela adição da segurança. O aumento da quantidade de dados a serem transmitidos (*overhead*) é de 32 bytes por quadro, um valor considerável para funções como as de escrita de uma única bobina (0x05) ou registrador (0x06), cujos quadros de requisição possuem originalmente apenas 11 bytes, mas torna-se negligenciável para funções como a escrita múltiplas bobinas (0x0F) ou múltiplos registradores (0x10), que possuem até 260 bytes.

3.2 MODBUSSEC DE HAYES E EL-KHATIB

Hayes e El-Khatib (2013) apresentam uma solução intitulada de ModbusSec baseada em HMAC e *Stream Control Transmission Protocol* (SCTP), protocolo da camada de transporte que substitui o protocolo TCP e que apresenta funcionalidades que buscam impedir ataques de negação de serviço. Esta solução também não endereça a confidencialidade dos dados por considerar um atributo de pouco relevância para controle industrial.

O protocolo TCP possui suas próprias vulnerabilidades que podem afetar os protocolos nele baseado. De fato, várias taxonomias de ataques identificadas por Huitsing *et al.* (2008), como *TCP FIN Flood* e *TCP RST Flood*, baseiam-se especificamente em características do protocolo TCP, e não em aspectos do protocolo Modbus. Por outro lado, SCTP foi desenvolvido considerando tais problemas (STEWART, 2007), e apresenta mecanismos para mitigá-los, como o *four-way-handshake* que endereça ataques do tipo *TCP SYN Flood*.

Como detalhado em 2.1.2.2, a utilização de HMAC garante os atributos de integridade e autenticidade dos pacotes de maneira tão computacionalmente eficiente quanto a função de espalhamento escolhida. O trabalho, entretanto, não trata da distribuição da chave secreta necessária para o algoritmo, assumindo que esta foi compartilhada previamente entre os dispositivos ou enviada por outro protocolo. Uma representação do quadro Modbus dentro do quadro SCTP é apresentada na Figura 13.

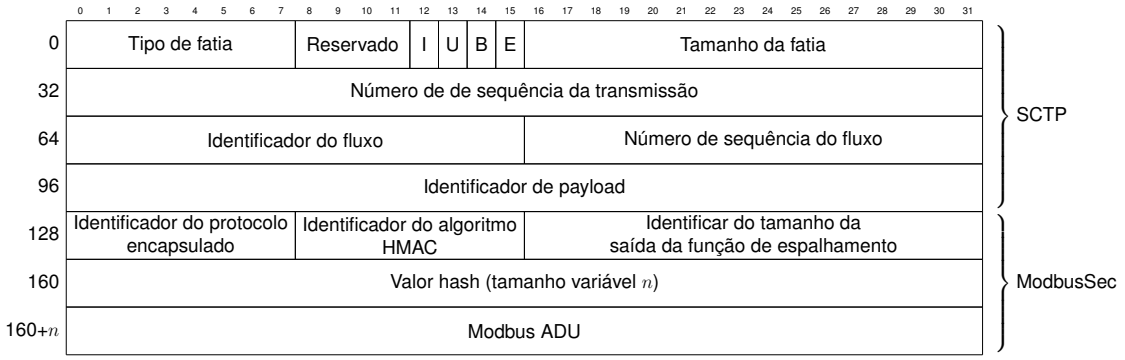

Figura 13: Proposta de implementação segura por Hayes e El-Khatib.

O cabeçalho SCTP é oito bytes maior do que o cabeçalho TCP e a solução proposta adiciona os campos de identificação do protocolo, do algoritmo HMAC e o tamanho da etiqueta de autenticação, somando quatro bytes ao tamanho do pacote a ser transmitido em cada transação. O tamanho da etiqueta depende da função de espalhamento escolhida como base para o algoritmo HMAC, podendo ser de 16 bytes para HMAC-MD5 ou 20 para HMAC-SHA1, por exemplo.

Somando estes aumentos, o *overhead* do protocolo ModbusSec em relação a Modbus TCP é de 28 bytes para HMAC-MD5 e 32 bytes para HMAC-SHA1, seguindo os mesmos exemplos de construções HMAC. O trabalho não apresenta informações a respeito da latência do protocolo proposto.

3.3 CONSIDERAÇÕES

Este capítulo apresentou alguns trabalhos de outros autores que tratam a respeito da segurança de sistemas SCADA e do protocolo Modbus. Em especial, foram detalhados os trabalhos de FOVINO *et al.* e HAYES; EL-KHATIB que, assim como este trabalho, propõem as soluções dos problemas de segurança do protocolo Modbus por meio da modificação do protocolo.

4 DESENVOLVIMENTO

Neste capítulos serão detalhados o método utilizado neste trabalho e seu desenvolvimento.

4.1 MÉTODO

Inicialmente realizou-se um levantamento do estado da arte a fim de elencar os principais problemas de segurança do protocolo Modbus e as soluções já existentes. Em seguida, foi proposta a solução apresentada neste trabalho, que foi implementada em duas plataformas.

A primeira é uma plataforma Linux, de propósito geral, constituída de uma Raspberry Pi 3b executando uma distribuição gerada com a ferramenta *Buildroot*[1]. A implementação do protocolo TLS foi fornecida pela biblioteca *mbedTLS*[2] e a implementação do protocolo Modbus foi feita com a biblioteca *libmodbus*[3], que foi modificada para prover conexões Modbus sobre TLS.

A segunda plataforma é de propósito específico, um sistema embarcado baseado no kit de desenvolvimento STM32F746G-DISC0 da STMicroeletronics com uma *firmware* utilizando o Sistema Operational de Tempo Real (do inglês, *Real Time Operational System*, RTOS) *FreeRTOS* (Real Time Engineers Ltd, 2017). A *lwIP* foi utilizada como pilha TCP/IP, a *mbedTLS* foi utilizada para prover as funcionalidades do protocolo TLS e a *FreeModbus* foi utilizada como implementação do protocolo Modbus.

[1] `https://buildroot.org/`
[2] `https://tls.mbed.org/`
[3] `http://libmodbus.org/`

Por fim, testes foram realizados para validar a implementação e verificar seu desempenho com relação a latência, *jitter* e *overhead*, comparado a versão original do protocolo.

4.2 ANÁLISE DO ESTADO DA ARTE

Como apresentado no Capítulo 3, ao menos três outros trabalhos propuseram modificações ao protocolo Modbus para resolver seus problemas de segurança. O mais antigo destes, de FOVINO *et al.*, utiliza um esquema de assinatura digital com apêndice em cada quadro transmitido. Isso significa que a cada transmissão Modbus será necessário a invocação de um método de criptografia assimétrica que, como descrito em 2.1.3, costuma ser computacionalmente menos eficiente do que métodos criptográficos simétricos.

Observa-se também que a proteção contra ataques de repetição proposta por FOVINO *et al.* depende da sincronização dos relógios dos dispositivos da rede SCADA, que segundo o autor foi realizada com o uso do protocolo NTP. Entretanto, o próprio protocolo NTP possui suas vulnerabilidades (MALHOTRA *et al.*, 2016) que podem ser utilizadas por um atacante para burlar a proteção proposta ou para causar uma negação de serviço, fazendo com que os relógios dos escravos estejam a frente do relógio do mestre, por exemplo.

A solução de HAYES; EL-KHATIB pode apresentar problemas práticos para ser implementada devido a utilização do protocolo SCTP. Pilhas TCP/IP largamente utilizadas em sistemas embarcados, como a *uIP* e *lwIP* ainda não o suportam (MANSLEY, 2008) e alguns sistemas operacionais de propósito geral como *DragonFly BSD* deixaram de suportá-lo (SASCHA, 2015).

O trabalho de HUITSING *et al.* identifica as taxonomias de ataques ao protocolo Modbus considerando a existência de um analisador de pacotes e/ou injetor de pacotes. Neste contexto, proteger o transporte das mensagens com o protocolo TLS parece ser uma solução eficiente para endereçar a todas as taxonomias descritas e, portanto, esta foi a abordagem escolhida para este

trabalho.

4.3 IMPLEMENTAÇÃO

O desenvolvimento teve inicio com a realização de um *fork* do repositório oficial da *libmodbus*, onde foram feitas as modificações necessárias para que a biblioteca pudesse realizar comunicações Modbus sobre TLS. Em seguida, foi construído um cliente e um servidor com a biblioteca para que fosse possível validar as modificações feitas e testes foram realizados localmente.

Utilizou-se então a adaptação para Linux da *FreeModbus* como base para a construção de uma camada de portabilidade que utiliza a *mbedTLS* para realizar as comunicações sobre TLS. Optou-se por um desenvolvimento inicial em um sistema de propósito genérico devido a existência de um amplo conjunto de ferramentas para depuração que auxiliam o processo de desenvolvimento, considerando ainda que a API da *mbedTLS* é a mesma em sistemas embarcados e em sistemas de propósito genérico, como Linux ou Microsoft Windows.

A *firmware* para o sistema embarcado então foi gerada com o auxílio da ferramenta *STM32CubeMX* (STMicroelectronics, 2018), que fornece os *drivers* e as adaptações (*ports*) de diversas bibliotecas para kits da STMicroeletronics. A ferramenta também gera o código de inicialização para os periféricos dos microcontroladores da empresa e exporta na forma de projetos para *Integrated Development Environments* (IDE), como Eclipse (Eclipse Foundation, 2017), ou como um conjunto de *Makefiles* para a ferramenta *Make* (FELDMAN, 2016).

Para o desenvolvimento deste projeto, optou-se pela utilização de *Makefiles*. Todavia, o projeto gerado pela ferramenta apresentou diversos problemas no processo de ligação dos arquivos objeto e montagem da imagem da *firmware*, de forma que se fez necessário a reescrita dos *Makefiles*. O projeto gerado contava com a inicialização para o *hardware* gerador de números aleatórios presente no kit, para o periférico de *ethernet* e os *ports* do *FreeRTOS*, *lwIP* e *mbedTLS*.

A *toolchain* utilizada foi a fornecida pela Arm para desenvolvimento sem sistema operacional para núcleos Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M7, Cortex-M23, Cortex-M33, Cortex-R4, Cortex-R5, Cortex-R7 e Cortex-R8 (Arm Limited, 2017). O código da *FreeModbus* e sua adaptação para utilizar a *mbedTLS* foram integrados ao projeto gerado e testes foram realizados entre a plataforma Linux e a plataforma embarcada.

Nas seções a seguir serão descritas as modificações necessárias à *libmodbus* e *FreeModbus* para o desenvolvimento deste trabalho.

4.3.1 LIBMODBUS

libmodbus é um biblioteca livre que implementa o envio e recebimento de dados seguindo as especificações do protocolo Modbus, tanto para operação como mestre quanto para operação como escravo. A biblioteca é escrita na linguagem C e suporta comunicações RTU e TCP. Comunicações RTU são implementadas por leituras e escritas em dispositivos de caracteres de sistemas *Unix* e derivados. Comunicações TCP utilizam a interface de *sockets* definida pelo padrão POSIX.

Seguindo as especificações do protocolo, nenhum tipo de segurança para as comunicações é implementado. O Código 4.1 apresenta um exemplo de utilização da biblioteca para comunicações TCP como mestre.

```
1  modbus_t *mb;
2  uint16_t tab_reg[32];
3
4  mb = modbus_new_tcp(MODBUS_SLAVE_ADDRESS, 502);
5  modbus_connect(mb);
6
7  modbus_read_registers(mb, 0, 5, tab_reg);
8
9  modbus_close(mb);
10 modbus_free(mb);
```

Código 4.1: Exemplo de utilização da biblioteca *libmodbus*

Neste exemplo, um cliente cria um novo contexto Modbus e se conecta a um escravo com endereço definido pela macro `MODBUS_SLAVE_ADDRESS` (linhas 4 e 5), lê cinco registradores iniciando pelo registrador zero (linha 7), encerra a conexão e libera os recursos alocados para o contexto (linhas 9 e 10). A função `modbus_new_tcp` cria um novo contexto Modbus para a utilização do protocolo sobre TCP/IP. A definição da estrutura que armazena este contexto, `modbus_t`, pode ser observada no Código 4.2,

```
1  struct _modbus {
2    /* Slave address */
3    int slave;
4    /* Socket or file descriptor */
5    int s;
6    int debug;
7    int error_recovery;
8    struct timeval response_timeout;
9    struct timeval byte_timeout;
10   const modbus_backend_t *backend;
11   void *backend_data;
12 };
13 ...
14 typedef struct _modbus modbus_t;
```

Código 4.2: Estrutura `mobus_t` da biblioteca *libmodbus*

Ao invocar os métodos que realizam algum tipo de comunicação, como `modbus_read_registers()`, a biblioteca utiliza os métodos apontados pelo campo `backend` para manipular os dados armazenados na estrutura apontada pelo capo `backend_data`. A definição da estrutura `modbus_backend_t` e os métodos por ela armazenados podem ser observados no Código 4.3.

```
1  typedef struct _modbus_backend {
2    unsigned int backend_type;
```

```
3   unsigned int header_length;
4   unsigned int checksum_length;
5   unsigned int max_adu_length;
6   int (*set_slave) (modbus_t *ctx, int slave);
7   int (*build_request_basis) (modbus_t *ctx, int function, int
     ↪ addr,
8                               int nb, uint8_t *req);
9   int (*build_response_basis) (sft_t *sft, uint8_t *rsp);
10  int (*prepare_response_tid) (const uint8_t *req, int *req_length
     ↪ );
11  int (*send_msg_pre) (uint8_t *req, int req_length);
12  ssize_t (*send) (modbus_t *ctx, const uint8_t *req, int
     ↪ req_length);
13  int (*receive) (modbus_t *ctx, uint8_t *req);
14  ssize_t (*recv) (modbus_t *ctx, uint8_t *rsp, int rsp_length);
15  int (*check_integrity) (modbus_t *ctx, uint8_t *msg,
16                          const int msg_length);
17  int (*pre_check_confirmation) (modbus_t *ctx, const uint8_t *req
     ↪ ,
18                                 const uint8_t *rsp, int
     ↪ rsp_length);
19  int (*connect) (modbus_t *ctx);
20  void (*close) (modbus_t *ctx);
21  int (*flush) (modbus_t *ctx);
22  int (*select) (modbus_t *ctx, fd_set *rset, struct timeval *tv,
23                 int msg_length);
24  void (*free) (modbus_t *ctx);
25 } modbus_backend_t;
```

Código 4.3: Estrutura `modbus_backend_t` da biblioteca *libmodbus*

Os métodos apontados por esta estrutura utilizarão as informações contidas na estrutura apontada pelo campo `backend_data` da estrutura `modbus_t` para realizar a comunicação. Este ponteiro não possui tipo definido pois o tipo de estrutura a ser apontada depende do tipo de comunicação do contexto Modbus. Para comunicações TCP uma estrutura do tipo `modbus_tcp_t` é utilizada e

para comunicações RTU utiliza-se uma estrutura do tipo `modbus_rtu_t`. Suas definições podem ser observadas nos Códigos 4.4 e 4.5, respectivamente.

```
1 typedef struct _modbus_tcp {
2   uint16_t t_id;
3   /* TCP port */
4   int port;
5   /* IP address */
6   char ip[16];
7 } modbus_tcp_t;
```

Código 4.4: Estrutura `modbus_tcp_t` da biblioteca *libmodbus*

```
1  typedef struct _modbus_rtu {
2    /* Device: "/dev/ttyS0", "/dev/ttyUSB0" or "/dev/tty.USA19*" on
     ↪ Mac OS X. */
3    char *device;
4    /* Bauds: 9600, 19200, 57600, 115200, etc */
5    int baud;
6    /* Data bit */
7    uint8_t data_bit;
8    /* Stop bit */
9    uint8_t stop_bit;
10   /* Parity: 'N', 'O', 'E' */
11   char parity;
12 #if defined(_WIN32)
13   struct win32_ser w_ser;
14   DCB old_dcb;
15 #else
16   /* Save old termios settings */
17   struct termios old_tios;
18 #endif
19 #if HAVE_DECL_TIOCSRS485
20   int serial_mode;
21 #endif
22 #if HAVE_DECL_TIOCM_RTS
23   int rts;
24   int rts_delay;
```

```
25    int onebyte_time;
26    void (*set_rts) (modbus_t *ctx, int on);
27  #endif
28    /* To handle many slaves on the same link */
29    int confirmation_to_ignore;
30  } modbus_rtu_t;
```

Código 4.5: Estrutura `modbus_rtu_t` da biblioteca *libmodbus*

Para a implementação de uma comunicação segura nesta biblioteca, um novo *backend* foi desenvolvido, sendo necessário implementar os métodos da estrutura `modbus_backend_t` e criar uma nova estrutura semelhante a `modbus_tcp_t` para armazenar o contexto da conexão TLS.

Esta adição à biblioteca não segue o padrão do protocolo definido por Modbus (2006) e portanto a modificação do código foi incluída como uma opção em tempo de compilação. A modificação feita ao sistema de compilação da biblioteca está detalhada no Apêndice A.1.

No arquivo `modbus-tls-private.h` a estrutura `modbus_tls_t` é declarada, como pode ser visto na Listagem 4.6. Esta estrutura cumpre a mesma função da estrutura `modbus_tcp_t` em comunicações Modbus TCP, armazenando as informações da sessão TLS e do contexto Modbus. Os primeiros campos da estrutura, entre as linha 6 e 10, são declarados na mesma ordem e tipos da estrutura `modbus_tcp_t` para que seja possível realizar a conversão do tipo `modbus_tls_t` para `modbus_tcp_t` a fim de que alguns dos métodos do *backend* TCP possam ser reutilizados, evitando a duplicidade de códigos de mesma finalidade.

```
1  #ifndef MODBUS_TLS_PRIVATE_H
2  #define MODBUS_TLS_PRIVATE_H
3
4  typedef struct _modbus_tls {
5      /* Transaction ID */
6      uint16_t t_id;
7      /* TCP port */
```

```
8     int port;
9     /* IP address */
10    char ip[16];
11 #if defined(USE_OPENSSL)
12   SSL_CTX *ctx;
13   SSL *ssl;
14 #elif defined(USE_MBEDTLS)
15   mbedtls_entropy_context entropy;
16   mbedtls_ctr_drbg_context drbg;
17   mbedtls_x509_crt cert;
18   mbedtls_pk_context pk;
19   mbedtls_ssl_config cfg;
20   mbedtls_ssl_context ctx;
21 #endif
22 } modbus_tls_t;
23
24 #endif
```

Código 4.6: Conteúdo do arquivo `modbus-tls-private.h`

O restante da `modbus_tls_t` armazena as estruturas necessárias para uma sessão e uma conexão TLS, de acordo com a escolha da biblioteca que implementa este protocolo.

Assim como os demais arquivos que possuem "*private*" em seus nomes, este cabeçalho é utilizado apenas em tempo de compilação, não sendo instalado no sistema. Desta forma a estrutura `modbus_tls_t`, assim como as estruturas `modbus_tcp_t` e `modbus_rtu_t`, é feita opaca ao usuário da *libmodbus*.

No arquivo `modbus-tls.h` os métodos para criar e manipular um contexto Modbus TLS são declarados de maneira análoga aos métodos utilizados para contextos Modbus TCP, como pode ser visto na Listagem 4.7. A principal diferença a ser notada está no método que cria o contexto, `modbus_new_tls`, que recebe o caminho para os certificados e chaves necessários para o protocolo SSL/TLS.

```
1  #ifndef MODBUS_TLS_H
2  #define MODBUS_TLS_H
3
4  #include "modbus.h"
5
6  MODBUS_BEGIN_DECLS
7
8  MODBUS_API modbus_t* modbus_new_tls(const char *ip_address, int
       ↪ port, const char *cert, const char *key, const char *ca);
9  MODBUS_API int modbus_tls_listen(modbus_t *ctx, int nb_connection)
       ↪ ;
10 MODBUS_API int modbus_tls_accept(modbus_t *ctx, int *s);
11
12 MODBUS_END_DECLS
13
14 #endif /* MODBUS_TLS_H */
```

Código 4.7: Conteúdo do arquivo `modbus-tls.h`

O método `modbus_new_tls` pode ser observado no Código 4.8. Este método aloca os recursos necessário em um contexto Modbus para um mestre ou escravo do protocolo. Entre as linhas 15 a 23 o método tenta ignorar o sinal `SIGPIPE`, como feito no método `modbus_new_tcp`. Na linha 25 é alocado memória para o contexto Modbus e seus campos são inicalizados entre as linhas 32 a 34.

```
1  #if defined(USE_TLS)
2  modbus_t* modbus_new_tls(const char *ip, int port, const char *
       ↪ cert, const char *key, const char *ca)
3  {
4    int ret;
5      modbus_t *ctx;
6  #if defined(USE_TLS)
7      modbus_tls_t *ctx_tls;
8  #if defined(USE_MBEDTLS)
9    char err_buf[200];
```

```
10 #endif
11 #endif
12     size_t dest_size;
13     size_t ret_size;
14 
15 #if defined(OS_BSD)
16     struct sigaction sa;
17 
18     sa.sa_handler = SIG_IGN;
19     if (sigaction(SIGPIPE, &sa, NULL) < 0) {
20         fprintf(stderr, "Could not install SIGPIPE handler.\n");
21         return NULL;
22     }
23 #endif
24 
25     ctx = (modbus_t *)malloc(sizeof(modbus_t));
26     if (ctx == NULL) {
27         return NULL;
28     }
29     _modbus_init_common(ctx);
30 
31     /* Could be changed after to reach a remote serial Modbus
    ↪ device */
32     ctx->slave = MODBUS_TCP_SLAVE;
33 
34     ctx->backend = &_modbus_tls_backend;
35 
36     ctx->backend_data = (modbus_tls_t *)malloc(sizeof(modbus_tls_t
    ↪ ));
37     if (ctx->backend_data == NULL) {
38         modbus_free(ctx);
39         errno = ENOMEM;
40         return NULL;
41     }
42     ctx_tls = (modbus_tls_t *)ctx->backend_data;
43 
```

```
44      if (ip != NULL) {
45          dest_size = sizeof(char) * 16;
46          ret_size = strlcpy(ctx_tls->ip, ip, dest_size);
47          if (ret_size == 0) {
48              fprintf(stderr, "The␣IP␣string␣is␣empty\n");
49              modbus_free(ctx);
50              errno = EINVAL;
51              return NULL;
52          }
53
54          if (ret_size >= dest_size) {
55              fprintf(stderr, "The␣IP␣string␣has␣been␣truncated\n");
56              modbus_free(ctx);
57              errno = EINVAL;
58              return NULL;
59          }
60      } else {
61          ctx_tls->ip[0] = '0';
62      }
63      ctx_tls->port = port;
64      ctx_tls->t_id = 0;
65
66  #ifdef USE_OPENSSL
        .
        .
        .
128 #elif defined(USE_MBEDTLS)
129   mbedtls_entropy_init(&ctx_tls->entropy);
130   mbedtls_ctr_drbg_init(&ctx_tls->drbg);
131
132   mbedtls_x509_crt_init(&ctx_tls->cert);
133
134   mbedtls_pk_init(&ctx_tls->pk);
135
136   mbedtls_ssl_config_init(&ctx_tls->cfg);
137
138   ret = mbedtls_ctr_drbg_seed(&ctx_tls->drbg, mbedtls_entropy_func
```

```
    ↪ , &ctx_tls->entropy, NULL, 0);
139 if(ret == MBEDTLS_ERR_CTR_DRBG_ENTROPY_SOURCE_FAILED) {
      .
      .
      .
128 }
129
130 mbedtls_ssl_init(&ctx_tls->ctx);
131
132 ret = mbedtls_x509_crt_parse_file(&ctx_tls->cert, cert);
133 if(ret != 0) {
      .
      .
      .
128 }
129
130
131 ret = mbedtls_pk_parse_keyfile(&ctx_tls->pk, key, NULL);
132 if(ret != 0) {
      .
      .
      .
128 }
129
130 ret = mbedtls_x509_crt_parse_file(&ctx_tls->cert, ca);
131 if(ret != 0) {
      .
      .
      .
128 }
129
130 mbedtls_ssl_conf_ca_chain(&ctx_tls->cfg, ctx_tls->cert.next,
    ↪ NULL);
131
132 ret = mbedtls_ssl_conf_own_cert(&ctx_tls->cfg, &ctx_tls->cert, &
    ↪ ctx_tls->pk);
133 if(ret != 0) {
```

```
    .
    .
    .
  }
#endif

    return ctx;
}
#endif
```

Código 4.8: Implementação do método `modbus_new_tls` no arquivo `modbus-tcp.c`

Na linha 36 aloca-se memória para os dados do *backend*, que correspondem a uma estrutura do tipo `modbus_tls_t` no caso de um contexto Modbus TLS. Em seguida, entre as linhas 44 a 63, é feito a cópia do endereço de IP fornecido como argumento ao método, caso este não seja nulo.

Entre as linha 129 a 136 os campos da estrutura `modbus_tls_t` são inicializados. Na linha 138 o gerador de números pseudoaleatórios é inicializado e na linha 130 ocorre a inicialização da sessão TLS. Nas linhas 132, 131 e 130 são carregados respectivamente o certificado local, a chave deste certificado e o certificado do CA. Na linha 130 é montado a cadeia de certificados confiáveis e na linha 132 a cadeia é adicionada à estrutura de configuração da sessão TLS.

O método `modbus_tls_listen` está listado no Código 4.9. Este método é fornecido apenas por completude da API, pois o método `modbus_tcp_listen` já é suficiente para escutar uma nova conexão TLS.

```
1 int modbus_tls_listen(modbus_t *ctx, int nb_connection)
2 {
3   return modbus_tcp_listen(ctx, nb_connection);
4 }
```

Código 4.9: Implementação do método `modbus_tls_listen` no arquivo `modbus-tcp.c`

O método `modbus_tls_accept` está listado no Código 4.10, inciando por uma chamada ao método `modbus_tcp_accept`, na linha 12, para estabelecer a conexão TCP com o cliente. Em seguida, na linha 43, realiza-se a configuração da sessão TLS. Na linha 45 são aplicadas as configurações padrão para um servidor TLS, na linha 53 é feito obrigatória a autenticação do cliente e na linha 56 configura-se o *socket* do cliente aceito como descritor para entrada e saída para a conexão TLS.

```
1  #if defined(USE_TLS)
2  int modbus_tls_accept(modbus_t *ctx, int *s)
3  {
4    int ret;
5    modbus_tls_t *ctx_tls = (modbus_tls_t *)ctx->backend_data;
6    struct timeval tv = ctx->response_timeout;
7    fd_set fds;
8  #if defined(USE_OPENSSL)
9    X509 *cert;
10 #endif
11
12   ret = modbus_tcp_accept(ctx, s);
13
14   if(ret < 0) {
15     return ret;
16   }
17
18 #if defined(USE_OPENSSL)
       .
       .
       .
42 #elif defined(USE_MBEDTLS)
43   mbedtls_ssl_setup(&ctx_tls->ctx, &ctx_tls->cfg);
44
45   ret = mbedtls_ssl_config_defaults(&ctx_tls->cfg,
     ↪ MBEDTLS_SSL_IS_SERVER, MBEDTLS_SSL_TRANSPORT_STREAM,
     ↪ MBEDTLS_SSL_PRESET_DEFAULT);
46
```

```
47  if(ret != 0) {
48    mbedtls_ssl_session_reset(&ctx_tls->ctx);
49    mbedtls_net_free(&ctx->s);
50    return -1;
51  }
52
53  mbedtls_ssl_conf_authmode(&ctx_tls->cfg,
     ↪ MBEDTLS_SSL_VERIFY_REQUIRED);
54  mbedtls_ssl_conf_rng(&ctx_tls->cfg, mbedtls_ctr_drbg_random, &
     ↪ ctx_tls->drbg);
55
56  mbedtls_ssl_set_bio(&ctx_tls->ctx, &ctx->s, mbedtls_net_send,
     ↪ mbedtls_net_recv, NULL);
57
58  do {
59    ret = mbedtls_ssl_handshake(&ctx_tls->ctx);
60
61    if(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
     ↪ MBEDTLS_ERR_SSL_WANT_WRITE) {
62      FD_ZERO(&fds);
63      FD_SET(ctx->s, &fds);
64
65      if(select(ctx->s+1, &fds, &fds, NULL, &tv) <= 0) {
66        mbedtls_ssl_session_reset(&ctx_tls->ctx);;
67        mbedtls_net_free(&ctx->s);
68        return -1;
69      }
70    }
71  } while(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
     ↪ MBEDTLS_ERR_SSL_WANT_WRITE);
72
73  if(ret != 0) {
74    mbedtls_ssl_session_reset(&ctx_tls->ctx);
75    mbedtls_net_free(&ctx->s);
76    return -1;
77  }
```

```
78 #endif
79
80   return ctx->s;
81 }
82 #endif
```

Código 4.10: Implementação do método `modbus_tls_accept` **no arquivo** `modbus-tcp.c`

Por fim, entre as linhas 58 a 71 realiza-se o *handshake* com o cliente. Caso o método `mbedtls_ssl_handshake` indique que a camada TLS precisa escrever ou ler mais dados, uma chamada ao método `select` é feita, na linha 65, para realizar a espera com *timeout*.

Na Listagem 4.11 pode ser observado a declaração do tipo `modbus_backend_t`. Nas Listagens 4.12 e 4.13 apresentam-se respectivamente as declarações dos *backends* TCP e TLS. Dados como o tipo de *backend* ou tamanhos de cabeçalho e do *checksum* são os mesmos em ambos os *backends*. Os métodos reimplementados para a comunicação segura foram `send`, `recv`, `connect`, `close`, `flush` e `select`. A descrição desta implementação está detalhada no Apêndice A.2.

```
1  typedef struct _modbus_backend {
2      unsigned int backend_type;
3      unsigned int header_length;
4      unsigned int checksum_length;
5      unsigned int max_adu_length;
6      int (*set_slave) (modbus_t *ctx, int slave);
7      int (*build_request_basis) (modbus_t *ctx, int function, int
       ↪ addr,
8                                  int nb, uint8_t *req);
9      int (*build_response_basis) (sft_t *sft, uint8_t *rsp);
10     int (*prepare_response_tid) (const uint8_t *req, int *
       ↪ req_length);
11     int (*send_msg_pre) (uint8_t *req, int req_length);
12     ssize_t (*send) (modbus_t *ctx, const uint8_t *req, int
```

```
    ↪ req_length);
13     int (*receive) (modbus_t *ctx, uint8_t *req);
14     ssize_t (*recv) (modbus_t *ctx, uint8_t *rsp, int rsp_length);
15     int (*check_integrity) (modbus_t *ctx, uint8_t *msg,
16                             const int msg_length);
17     int (*pre_check_confirmation) (modbus_t *ctx, const uint8_t *
    ↪ req,
18                                    const uint8_t *rsp, int
    ↪ rsp_length);
19     int (*connect) (modbus_t *ctx);
20     void (*close) (modbus_t *ctx);
21     int (*flush) (modbus_t *ctx);
22     int (*select) (modbus_t *ctx, fd_set *rset, struct timeval *tv
    ↪ , int msg_length);
23     void (*free) (modbus_t *ctx);
24 } modbus_backend_t;
```

Código 4.11: Declaração do tipo `modbus_backend_t` no arquivo `modbus-private.h`

```
1
2  const modbus_backend_t _modbus_tcp_backend = {
3      _MODBUS_BACKEND_TYPE_TCP,
4      _MODBUS_TCP_HEADER_LENGTH,
5      _MODBUS_TCP_CHECKSUM_LENGTH,
6      MODBUS_TCP_MAX_ADU_LENGTH,
7      _modbus_set_slave,
8      _modbus_tcp_build_request_basis,
9      _modbus_tcp_build_response_basis,
10     _modbus_tcp_prepare_response_tid,
11     _modbus_tcp_send_msg_pre,
12     _modbus_tcp_send,
13     _modbus_tcp_receive,
14     _modbus_tcp_recv,
15     _modbus_tcp_check_integrity,
16     _modbus_tcp_pre_check_confirmation,
17     _modbus_tcp_connect,
18     _modbus_tcp_close,
```

```
19      _modbus_tcp_flush,
20      _modbus_tcp_select,
21      _modbus_tcp_free
```

Código 4.12: Declaração do *backend* TCP no arquivo `modbus-tcp.c`

```
1
2  #if defined(USE_TLS)
3  const modbus_backend_t _modbus_tls_backend = {
4      _MODBUS_BACKEND_TYPE_TCP,
5      _MODBUS_TCP_HEADER_LENGTH,
6      _MODBUS_TCP_CHECKSUM_LENGTH,
7      MODBUS_TCP_MAX_ADU_LENGTH,
8      _modbus_set_slave,
9      _modbus_tcp_build_request_basis,
10     _modbus_tcp_build_response_basis,
11     _modbus_tcp_prepare_response_tid,
12     _modbus_tcp_send_msg_pre,
13     _modbus_tls_send,
14     _modbus_tcp_receive,
15     _modbus_tls_recv,
16     _modbus_tcp_check_integrity,
17     _modbus_tcp_pre_check_confirmation,
18     _modbus_tls_connect,
19     _modbus_tls_close,
20     _modbus_tls_flush,
21     _modbus_tls_select,
22     _modbus_tls_free
23 };
```

Código 4.13: Declaração do *backend* TLS no arquivo `modbus-tcp.c`

4.3.2 FREEMODBUS

FreeModbus é uma implementação livre de um escravo Modbus para sistemas embarcados. A implementação é dividida em duas camadas: uma

camada de aplicação Modbus, para a implementação das funções de escravo do protocolo, e a camada de rede, que é responsável por transmitir e receber os dados da comunicação.

A camada de rede define diversos *callbacks* a serem implementados por uma camada de portabilidade, que gera a abstração do *hardware* onde a FreeModbus está sendo implementada. Exemplos de *callbacks* desta camada são `xMBTCPPortSendResponse`, para enviar dados via TCP, `prvbMBPortSerialRead` para ler dados de uma porta serial, ou `xMBPortTimersInit` para inicializar o temporizador necessário para detecção de *timeouts*.

Da mesma forma, a camada de aplicação define *callbacks* onde o usuário da FreeModbus deve implementar as funções do protocolo, como por exemplo o *callback* `eMBRegInputCB`, chamado para realizar a leitura de registradores de entrada (função 0x04), ou `eMBRegHoldingCB`, chamado para realizar tanto a escrita quanto a leitura de registradores comuns (funções 0x06 e 0x10 para escrita e 0x03 para leitura).

Os Código 4.14, 4.15 e 4.16 apresentam um exemplo de utilização da FreeModbus e duas implementações de *callbacks*.

```
1  static USHORT   usRegInputStart = REG_INPUT_START;
2  static USHORT   usRegInputBuf[REG_INPUT_NREGS];
3  static USHORT   usRegHoldingStart = REG_HOLDING_START;
4  static USHORT   usRegHoldingBuf[REG_HOLDING_NREGS];
5
6  void ModbusTask(void const * argument)
7  {
8    eMBErrorCode xStatus;
9
10   while(1) {
11     if(eMBTCPInit(MB_TCP_PORT_USE_DEFAULT) != MB_ENOERR) {
12       eMBClose();
13       continue;
14     }
```

```
15
16      eMBEnable();
17
18      do {
19        xStatus = eMBPoll();
20      } while(xStatus == MB_ENOERR);
21
22      eMBDisable();
23      eMBClose();
24    }
25  }
```

Código 4.14: Tarefa exemplificando a utilização das funionalidades da FreeModbus

No Código 4.14 exemplifica-se uma tarefa de um servidor Modbus. Entre as linhas 1 e 4 são declarados os mapas de registradores de entrada e de registradores. As constantes `REG_INPUT_START` e `REG_HOLDING_START` indicam o início de cada mapa e as constantes `REG_INPUT_NREGS` e `REG_HOLDING_NREGS` indicam o tamanho.

Na linha 10 inicia-se um laço principal da tarefa. Na linha 11 a pilha do procolo Modbus/TCP é inicializada e em seguida habilitada, na linha 16. Entre as linhas 18 e 20 realiza-se a espera ativa (*polling*) de novas requisições Modbus. A chamada ao método `eMBPoll` verifica se novos eventos ocorreram, como uma nova conexão ou a chegada de mais dados, e invoca os métodos internos da FreeModbus para o correto tratamento do evento identificado. Se nenhum dos métodos invocados apresentar erros, `eMBPoll` retorna a constante `MB_ENOERR` e o laço de *polling* se repete. Caso alguma falha seja apresentada, a tarefa sai do laço de *polling*, desativa e destrói a pilha nas linhas 22 e 23 e retorna ao início do laço principal, onde tenta inicializar a pilha novamente.

O Código 4.15 apresenta a implementação de um *callback* da camada de aplicação para operações de leitura e escrita em registradores. Na linha 6 é verificado se os registradores a serem operados estão dentro do mapa

de registradores e na linha 8 utiliza-se um `switch-case` para verificar o tipo de operação a ser realizada. De acordo com a operação informada pelo argumento `eMode` do *callback* o loop de leitura da linha 11 ou de escrita da linha 22 será executado.

```
1  eMBErrorCode eMBRegHoldingCB(UCHAR * pucRegBuffer, USHORT
       ↪ usAddress, USHORT usNRegs, eMBRegisterMode eMode)
2  {
3      eMBErrorCode    eStatus = MB_ENOERR;
4      int             iRegIndex;
5
6      if((usAddress >= REG_HOLDING_START) && (usAddress + usNRegs <=
       ↪  REG_HOLDING_START + REG_HOLDING_NREGS)) {
7          iRegIndex = (int)(usAddress - usRegHoldingStart);
8          switch ( eMode ) {
9              /* Pass current register values to the protocol stack.
       ↪  */
10         case MB_REG_READ:
11             while(usNRegs > 0) {
12                 *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[
       ↪ iRegIndex] >> 8 );
13                 *pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[
       ↪ iRegIndex] & 0xFF );
14                 iRegIndex++;
15                 usNRegs--;
16             }
17             break;
18
19             /* Update current register values with new values from
       ↪  the
20              * protocol stack. */
21         case MB_REG_WRITE:
22             while(usNRegs > 0) {
23                 usRegHoldingBuf[iRegIndex] = *pucRegBuffer++ << 8;
24                 usRegHoldingBuf[iRegIndex] |= *pucRegBuffer++;
25                 iRegIndex++;
```

```
26                  usNRegs--;
27              }
28          }
29      } else {
30          eStatus = MB_ENOREG;
31      }
32      return eStatus;
33  }
```

Código 4.15: Exemplo de *callback* da FreeModbus para operações de leitura e escrita em registradores

No código 4.16 apresenta-se a implementação de um *callback* para uma função do protocolo Modbus que o escravo não deseja dar suporte. Neste exemplo, o servidor não tem mapas de bobinas, e portanto qualquer função sobre bobinas deve falhar. A implementação da FreeModbus irá chamar o *callback* `eMBRegCoilsCB` que um mestre requisitar este tipo de operação, sendo necessário esta implementação mínima do método para apenas retornar um erro.

```
1  eMBErrorCode eMBRegCoilsCB(UCHAR * pucRegBuffer, USHORT usAddress,
       ↪ USHORT usNCoils, eMBRegisterMode eMode)
2  {
3      return MB_ENOREG;
4  }
```

Código 4.16: Exemplo de *callback* da FreeModbus para uma função não implementada de leitura e escrita em bobinas

Para realizar comunicações seguras com a FreeModbus foi necessário implementar uma camada de portabilidade que utilize os métodos fornecidos pela biblioteca mbedTLS para realizar as comunicações com o protocolo TLS. Como base para esta implementação utilizou-se a adaptação para Linux, que utiliza a interfaces de *sockets* POSIX e é composta por um cabeçalho, `port.h`, e dos fontes `portevent.c`, `portother.c` e `porttcp.c`.

As funções modificadas desta camada de abstração foram `xMBTCPPortInit`, `vMBTCPPortClose`, `xMBPortTCPPool`, `xMBTCPPortSendResponse`, `prvvMBPortReleaseClient` e `prvbMBPortAcceptClient`, todos pertencentes ao arquivo `porttcp.c`. A descrição do funcionamento destes métodos é apresentada no Apêndice B.

4.4 SUÍTE DE TESTES

A suíte de testes da *libmodbus* foi tomada como base para validar a solução desenvolvida e verificar seu desempenho. Um dos programas desta suíte, chamado `bandwidth-client`, foi modificado para suportar a versão segura do protocolo, gerar informações a respeito do tempo necessário para realizar a conexão e latência das transações Modbus.

Os testes realizados por este programa são baseado no guia de referência PI-MBUS-300 (MODICON,), que descreve o procolo Modbus, a operação de diversos códigos de funções e as taxas de transferências esperadas. Três funções do protocolo são utilizadas, a leitura de bobinas (código 0x01), a leitura de registradores (código 0x03) e a escrita e leitura de registradores (código 0x17), que são invocados 100.000 vezes cada para operar sobre o número máximo de elementos permitidos pelo protocolo para cada uma destas funções.

São tomados os tempos de cada transação Modbus, utilizando o método POSIX `clock_gettime` antes da invocação do método da API da *libmodbus* e após o seu retorno. O relógio utilizado como referência para esta medida foi o `CLOCK_MONOTONIC`, que representa o tempo monotônico a partir de um ponto não especificado (CLOCK_GETRES(2)..., 2017) e serve, portanto, apenas para o calculo de diferenças temporais. Este relógio foi escolhido por não ser afetado por saltos temporais, como a redefinição de data e hora do sistema.

Os dados obtidos são usados por `bandwidth-client` para calcular o *goodput* da conexão, isto é, a taxa de transferência de dados úteis a aplicação

(FLOYD, 2008), desconsiderando neste caso os dados transmitidos pelos protocolos Modbus, TLS, TCP e IP. As modificações realizadas fizeram com que a aplicação também calcule o tempo necessário para realizar a conexão e os valores mínimos, médios e máximos de latência e o desvio padrão desta.

Como contraparte deste cliente, um servidor de testes chamado `bandwidth-server-one` é fornecido pela suíte. Sua implementação é bastante trivial, tratando-se de um servidor Modbus que apenas atende as requisições, sem gerar mais estatísticas ou outras informações sobre o desempenho da conexão. Seu funcionamento é bastante semelhante ao exemplo de tarefa da FreeModbus, apresentado na Seção 4.3.2 e listado no Código 4.14, sendo apenas necessário garantir que a faixa de registradores e bobinas utilizadas nos testes esteja disponível.

Esta tarefa foi implementada na plataforma embarcada o teste foi executado com 43 das 160 suítes criptográficas implementadas pela `mbedTLS`. As suítes não testadas não foram utilizadas por não serem suportadas pela adaptação da biblioteca na plataforma embarcada utilizada ou pelos certificados gerados, como o caso da suíte `TLS-RSA-PSK-WITH-AES-256-GCM-SHA384`, que utiliza uma *Pre Shared Key* (do inglês, chave pré-compartilhada, PSK) ao invés de certificados para autenticar os pares de uma conexão.

O ambiente de rede montado para a realização dos testes está exemplificado na Figura 14. Um switch de oito portas baseado no *chipset* RTL8309SC da Realtek foi utilizado para interligar a Raspberry e o kit STM32F749. Uma terceira máquina foi conectada a rede para prover outros serviços, como *Dynamic Host Configuration Protocol* (DHCP), para distribuir o endereçamento IP.

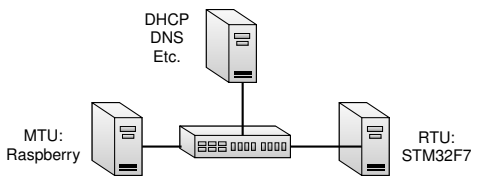

Figura 14: Ambiente de rede montado para a realização dos testes

5 RESULTADOS

Os resultados da execução dos testes serão apresentados neste capítulo. Como visto em 2.3.1, nem todas as suítes criptográficas realizam a encriptação dos dados transmitidos. Desta forma, as suítes testadas foram divididas em dois grupos: 39 suítes que proveem a confidencialidade dos dados transmitidos e quatro que proveem apenas a autenticidade e integridade das mensagens.

Para as o primeiro grupo, a solução inviabiliza a utilização de analisadores e injetores de pacotes, necessários para todas as taxonomias descritas por Huitsing *et al.* (2008). Desta forma, a solução endereça todos os ataques baseados nas 15 taxonomias comuns aos modos de transmissão serial e TCP.

Para as 13 taxonomias específicas a transmissões TCP, entretanto, a solução impede apenas os ataques que exploram características da especificação da camada de aplicação Modbus, como *TCP Pool Exhaustion*, permanecendo suscetível às vulnerabilidades do protocolo TCP, como *TCP SYN Flood* e *TCP RST Flood*. Isso se deve ao fato do protocolo TLS ser baseado no protocolo TCP e, portanto, não possuir escopo para solucionar estes problemas. Tais vulnerabilidades podem ser endereçadas com outras soluções comumente aplicadas em sistemas baseados no protocolo TCP, como a filtragem de pacotes e o uso de *SYN cookies* (EDDY, 2007).

Caso a solução seja implementada com um suíte que ofereça apenas autenticidade, somente a existência de injetores de pacotes será mitigada, tornando a solução suscetível aos ataques passivos de reconhecimento da rede SCADA. Como descrito em 2.3, a utilização deste tipo de suíte pode ser vantajosa por oferecer um menor custo computacional para a solução.

Isso se torna evidente na comparação da latência das soluções, que são apresentadas nas Figuras 15 a 20. Os gráficos mostram a latência para a execução das funções de leitura de bobinas (código 0x01), leitura de registradores (código 0x03) e escrita e leitura de registradores (código 0x17).

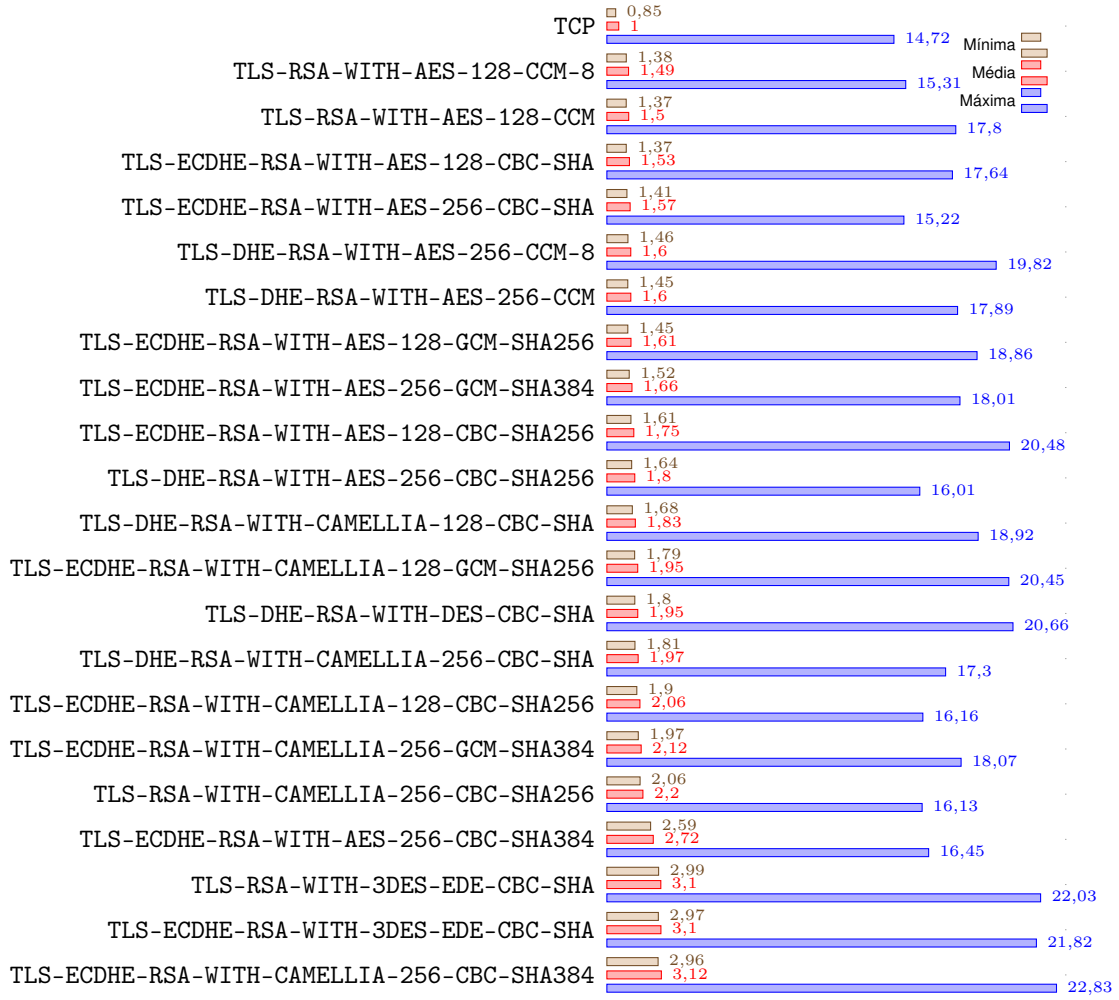

Figura 15: Latência para suítes criptográficas com confidencialidade em operações de leitura de bobinas (ms)

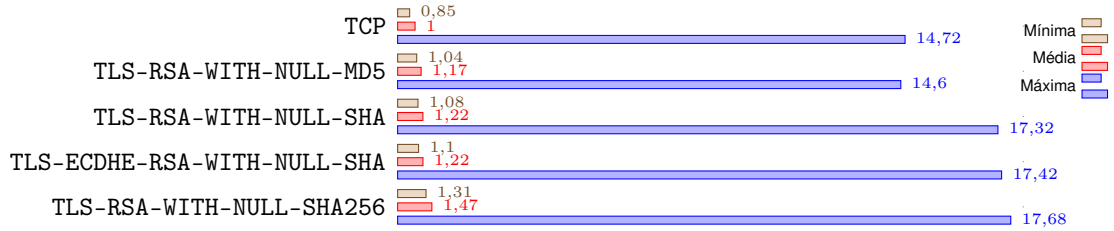

Figura 16: Latência para suítes criptográficas sem confidencialidade em operações de leitura de bobinas (ms)

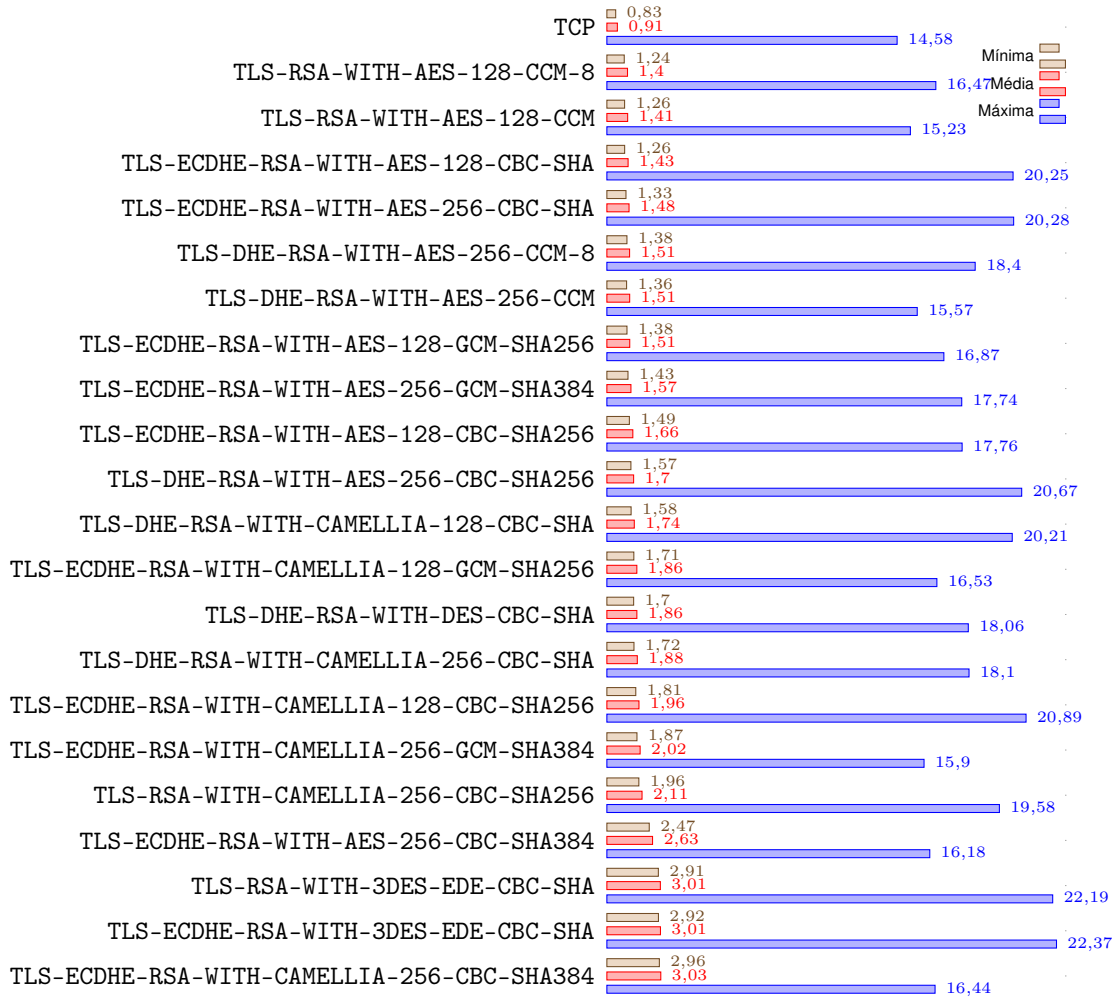

Figura 17: Latência para suítes criptográficas com confidencialidade em operações de leitura de registradores (ms)

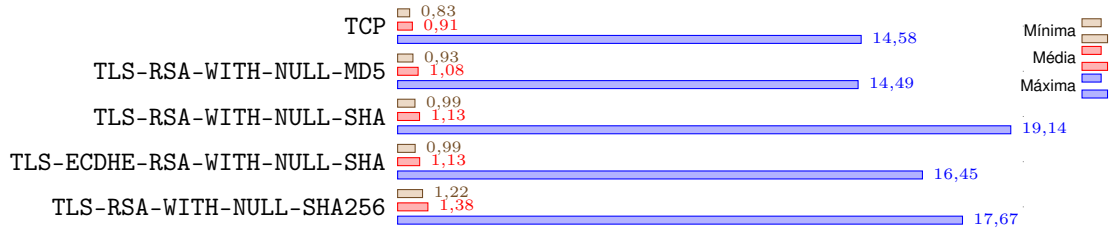

Figura 18: Latência para suítes criptográficas sem confidencialidade em operações de leitura de registradores (ms)

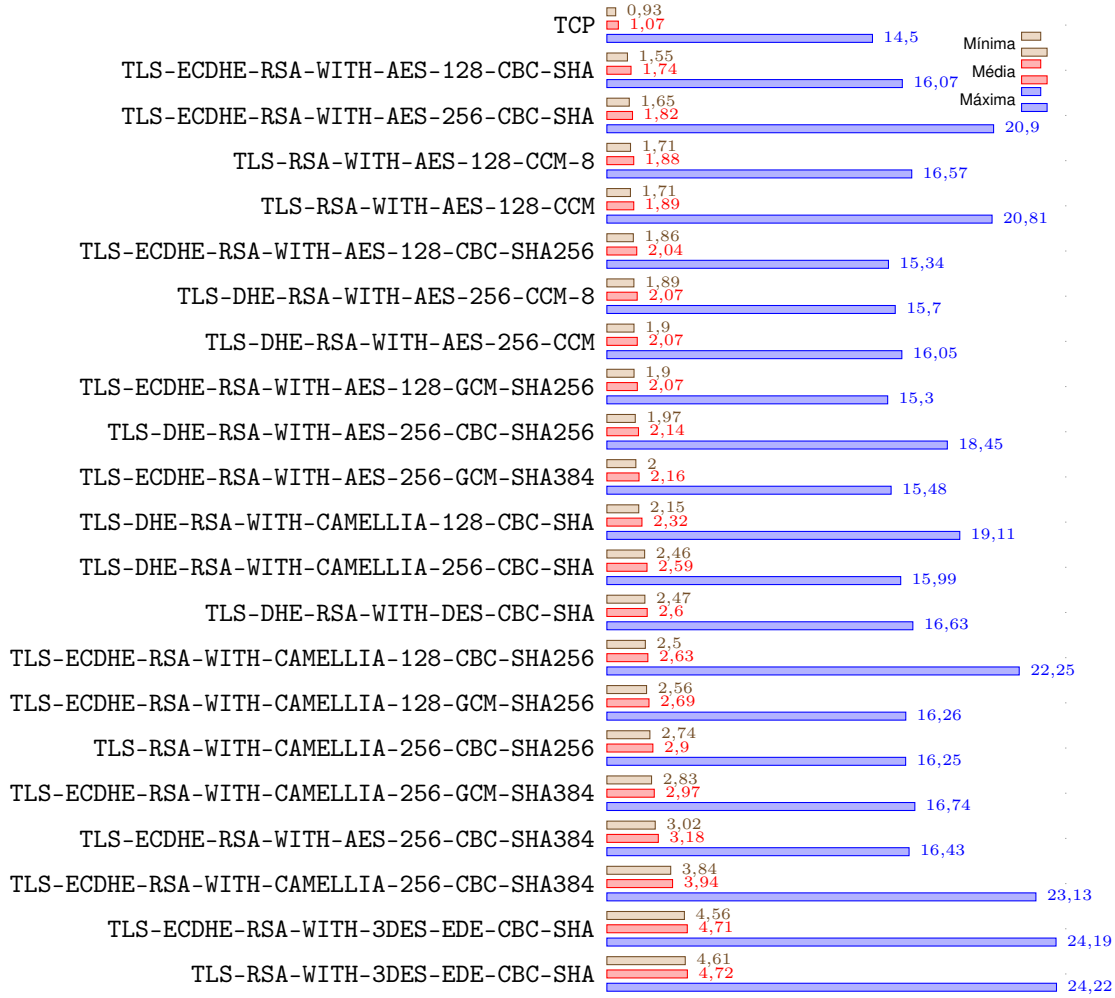

Figura 19: Latência para suítes criptográficas com confidencialidade em operações de escrita e leitura de registradores (ms)

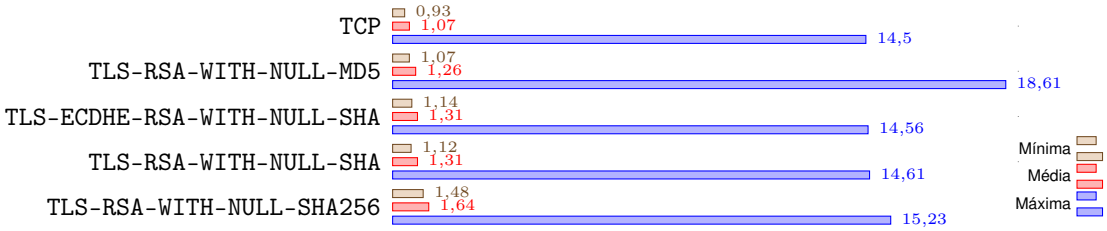

Figura 20: Latência para suítes criptográficas sem confidencialidade em operações de escrita e leitura de registradores (ms)

O algoritmo de troca de chaves só é empregado no início da conexão e, portanto, sua escolha não impacta no tempo das transações Modbus. Desta forma, as suítes que utilizam diferentes algoritmos de troca de chaves para um mesmo método de encriptação foram omitidas.

Sistema que opera em tempo real devem atender as tarefas em execução em um determinado limite de tempo, comumente referido como *deadline*. Este geralmente é o caso em ICS, de forma que uma elevada latência pode inviabilizar a utilização do protocolo proposto em algumas aplicações. Estas restrições temporais, no entanto, são extremamente ligadas a aplicação, de forma que um estudo de caso seria necessário para determinar a viabilidade da solução em diferentes áreas.

Nota-se que para todas as funções testadas, a suíte com autenticação mais lenta ainda é, na média, de 2 a 10 μs mais rápida do que a suíte com criptografia de menor latência. Da mesma forma, a suíte confidencial com melhor segurança segundo a *mbedTLS*, a `TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384`, apresenta uma latência 66% maior do que a versão não segura do protocolo enquanto a suíte com autenticação que utiliza a construção MAC baseada na função de espalhamento mais segura entre as testadas, a `TLS-RSA-WITH-NULL-SHA256`, apresenta uma latência 47% maior que versão original do protocolo.

Essa diferença se deve não apenas ao aumento do custo computacional para realizar a encriptação mas também pelo aumento do tamanho do

pacote a ser transmitido pela rede, também chamado de *overhead*, que é apresentado nas Figuras 21 a 24. Como dito em 2.1.2.1, cifras operadas em bloco, como o caso de AES-256-GCM, operam sobre um bloco de tamanho fixo de dados, sendo necessário em alguns casos adicionar um preenchimento ao pacote para que os dado possuam um tamanho múltiplo do tamanho do bloco sobre o qual a cifra trabalha.

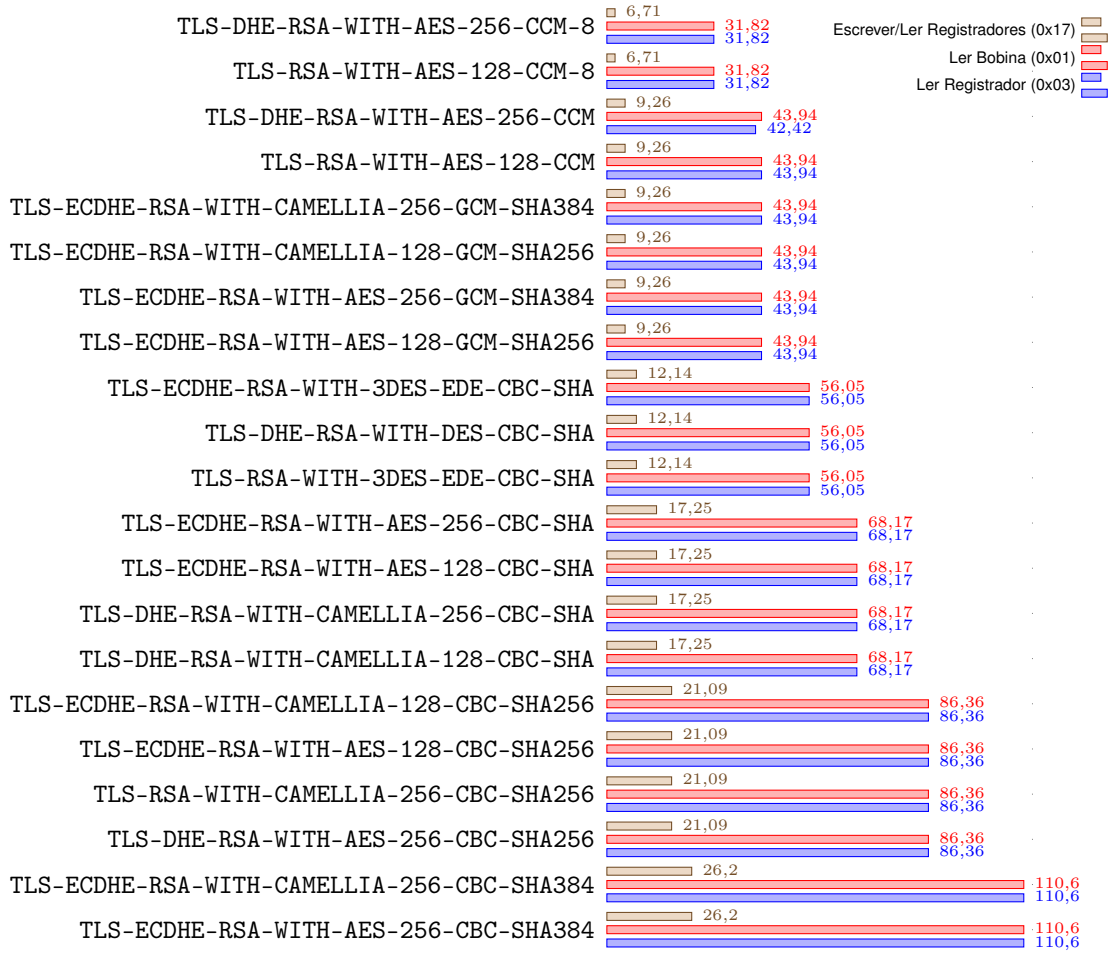

Figura 21: *Overhead* para requisições em suítes criptográficas com confidencialidade (%)

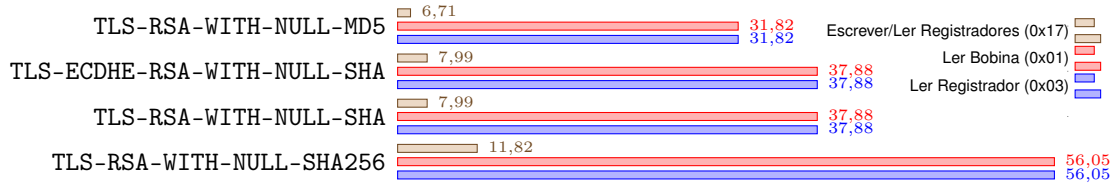

Figura 22: *Overhead* para requisições em suítes criptográficas sem confidencialidade (%)

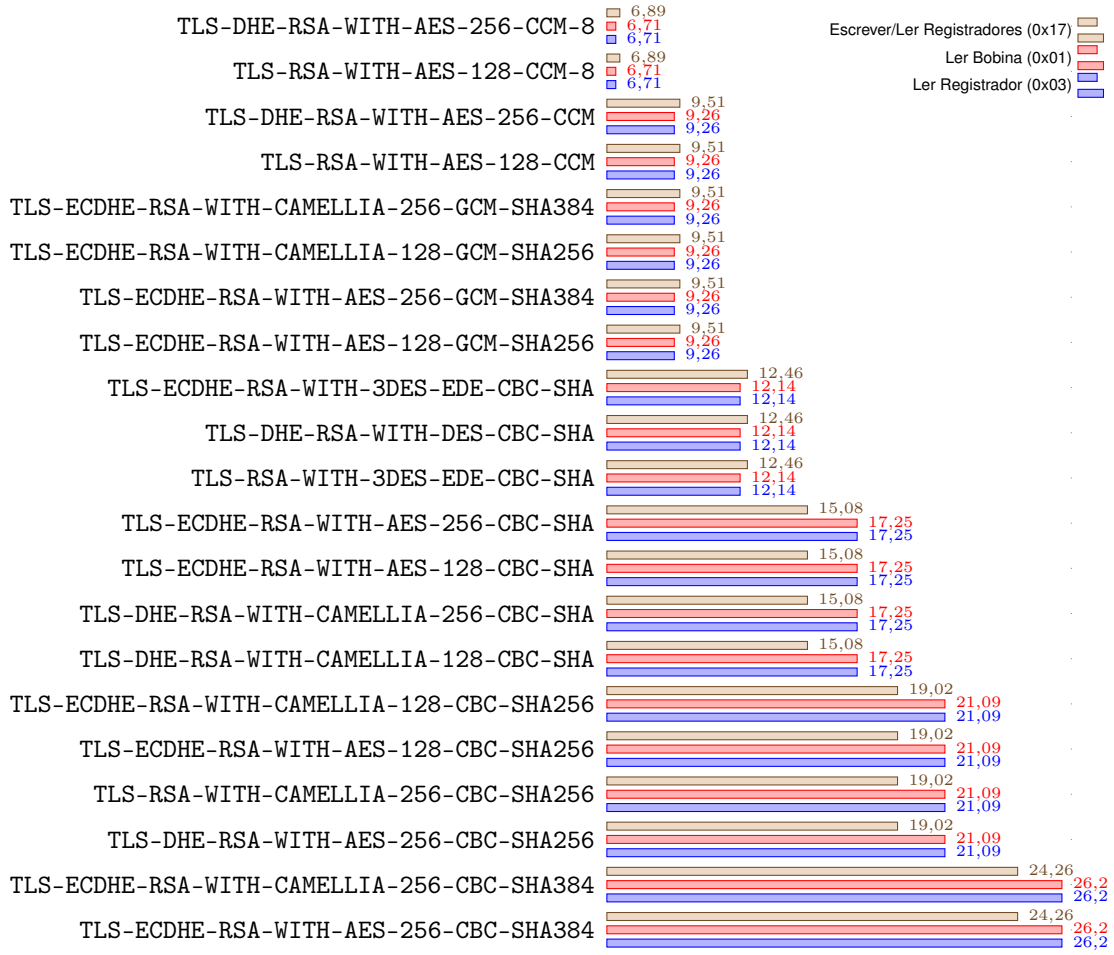

Figura 23: *Overhead* para respostas em suítes criptográficas com confidencialidade (%)

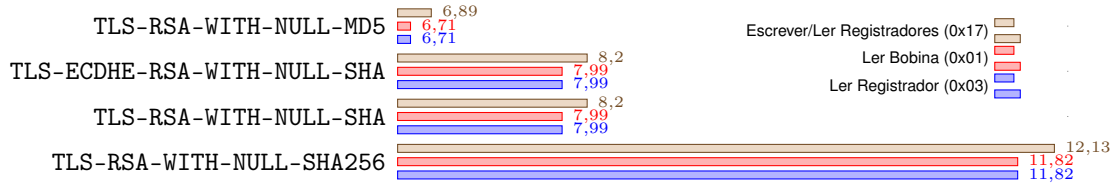

Figura 24: *Overhead* para respostas em suítes criptográficas sem confidencialidade (%)

É possível ainda observar nestas figuras que as suítes com que são operadas em modo CCM apresentam um *overhead* consideravelmente menor. Este comportamento é o esperado devido ao fato de suas etiquetas de autenticação serem menores (MCGREW; BAILEY, 2012), caracterizando-as como as melhores opções para ambientes onde a largura de banda disponível é a maior restrição.

Para uma rede de pacotes, *jitter* é definido como a variabilidade na latência com que os pacotes são entregues (KUROSE; ROSS, 2009, p. 602). As Figura 25 e 26 apresentam, respectivamente, o *jitter* obtido para as suítes com e sem confidencialidade.

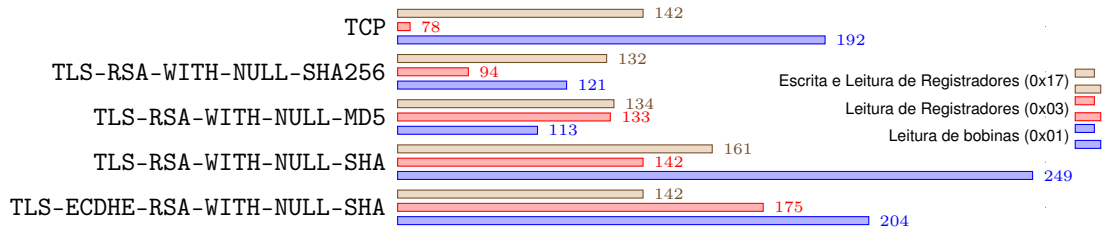

Figura 25: *Jitter* para suítes criptográficas sem confidencialidade (ns)

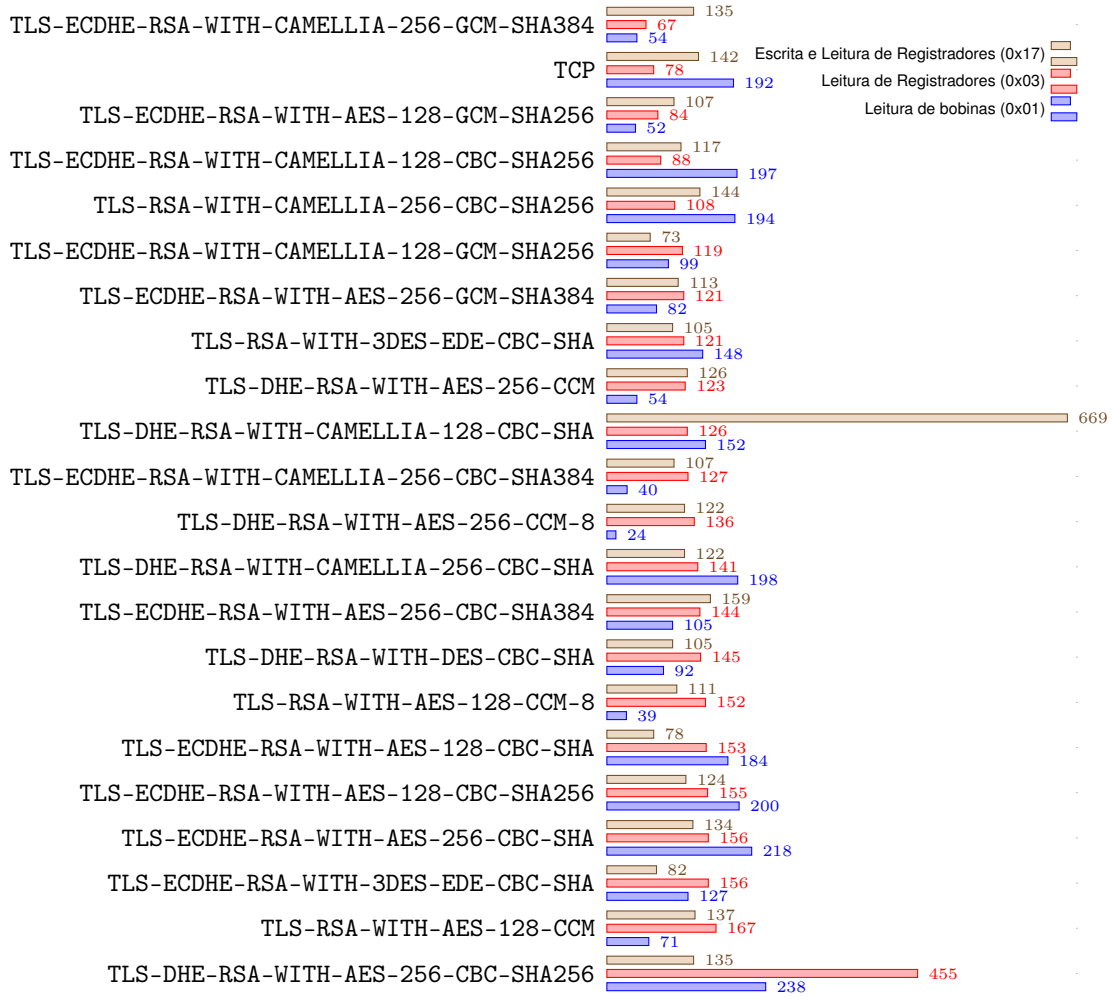

Figura 26: *Jitter* para suítes criptográficas com confidencialidade (ns)

Observa-se que os valores medidos são bastante pequenos, na casa de nanosegundos e que a versão não segura do protocolo não apresentou o menor *jitter*. Isso indica que provavelmente estas variações estão mais relacionadas a instabilidades na rede que interconectou os dispositivos do que a variações no tempo necessário para encriptar e autenticar os pacotes.

Os gráficos das Figuras 27 e 28 apresentam o tempo necessário para estabelecer a conexão utilizando, respectivamente, suítes com e sem confidencialidade. A cifra simétrica em uso pela suíte não é de grande relevância

neste teste pois apenas as duas últimas mensagens trocadas para estabelecer a conexão (do tipo `FINISHED` do subprotocolo TLS Handshake) serão encriptadas.

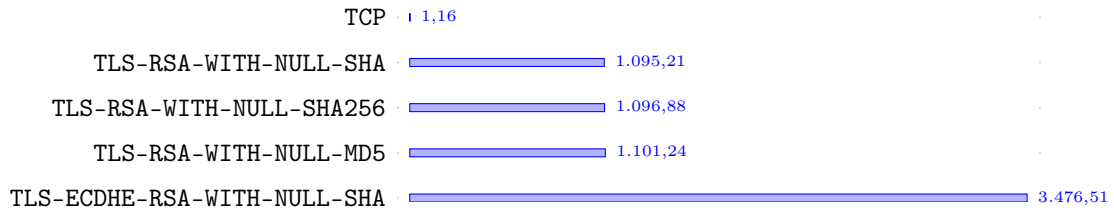

Figura 27: Tempo de conexão para suítes criptográficas sem confidencialidade (ms)

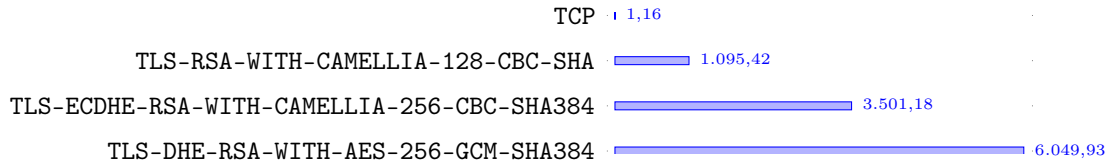

Figura 28: Tempo de conexão para suítes criptográficas com confidencialidade (ms)

Desta forma, para melhorar a legibilidade das informações apresentadas, suítes com o mesmo algoritmo de troca de chaves e diferentes métodos simétricos de encriptação foram apresentadas uma única vez na Figura 28. Os resultados completos para todas as suítes estão apresentados no Apêndice C, onde é possível observar que o tempo de conexão destas suítes é, de fato, bastante semelhante.

Nota-se que os tempos apresentados em ambos os gráficos são bastante elevados em comparação a versão não segura do protocolo, com valores próximos a um segundo para suítes RSA e chegando a seis segundos para a suíte `TLS-DHE-RSA-WITH-AES-256-GCM-SHA384`. Em geral, o tempo de conexão não é de grande relevância pois em operações normais a conexão entre mestre e escravo deve ficar aberta constantemente (Schneider Automation, 2006). Entretanto, sempre existe a possibilidade de instabilidades na rede causarem desconexões. Neste cenário, um sistema SCADA perderia o controle sobre a planta

pelo período necessário para o reestabelecimento da conexão, possivelmente tornando os algoritmos efêmeros de Diffie-Hellman (DHE e ECDHE) inapropriados para sistemas críticos de resposta rápida.

As Figuras 29 e 30 apresentam o *goodput* obtido com o uso das funções de leitura de bobinas, leitura de registradores e escrita e leitura de registradores. Nesse resultado é possível observar que mesmo com um *overhead* semelhante, as suítes sem confidencialidade obtiveram uma taxa de transferência superior em relação as suítes com cifras operadas em modo CCM.

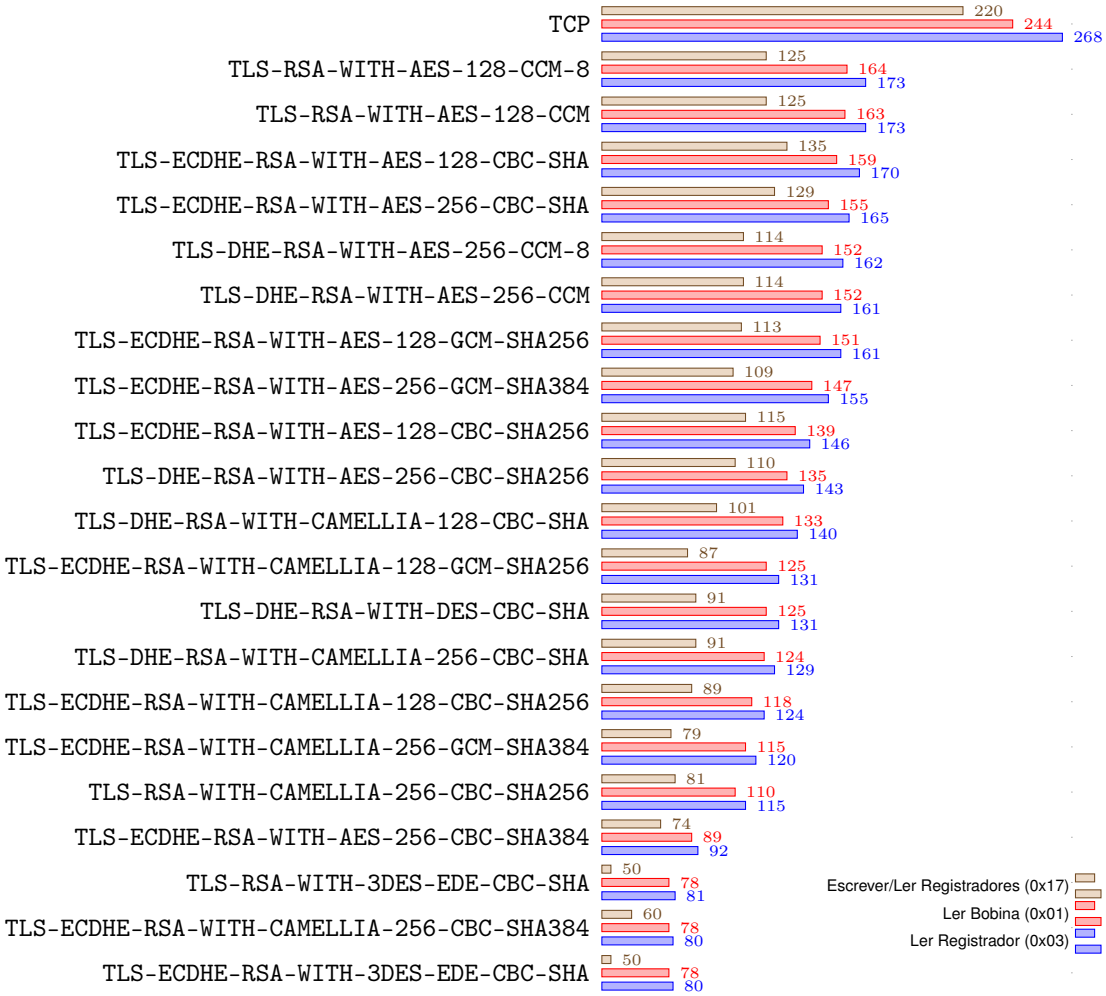

Figura 29: *Goodput* para suítes criptográficas com confidencialidade (KiB/s)

Figura 30: *Goodput* para suítes criptográficas sem confidencialidade (KiB/s)

Comparativamente, a suíte `TLS-RSA-WITH-AES-128-CCM-8` apresentou taxas de transferência útil 32.79%, 35.45% e 43.18% menores do que a versão sem segurança do protocolo para as operações de leitura de bobinas, leitura de registradores e escrita e leitura de registradores, respectivamente, enquanto a suíte `TLS-RSA-WITH-NULL-MD5` apresentou para as mesmas operações valores 14.75%, 16.04% e 14.55% menores.

Para uma suíte de maior custo computacional, como o caso da `TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384`, a redução do *goodput* para as mesmas operações foram de 39.75%, 42.16% e 50.45%. Para a suíte sem confidencialidade de mair custo computacional testada, por sua vez, estes valores foram observados como 31.96%, 34.33% e 34.55%.

6 CONCLUSÃO

O presente trabalho abordou os problemas de segurança do protocolo Modbus aplicado a sistemas de controle industrial e redes SCADA. Tendo como base trabalhos anteriores que investigaram a segurança do protocolo, como os de Huitsing *et al.* (2008) e Fovino *et al.* (2009b), e analisando as demais soluções existentes, como as de Fovino *et al.* (2009a) e Hayes e El-Khatib (2013), uma solução baseada em TLS foi proposta e implementada em duas plataformas.

A primeira implementação foi realizada para um sistema Linux, em uma distribuição gerada com a ferramenta *Buildroot* e executada em uma Raspberry Pi 3b, baseada nas biblioteca *libmodbus* e *mbedTLS*. A segunda implementação foi feita em uma plataforma embarcada, baseada no kit de desenvolvimento STM32F469G-DISC0 da STMicroeletronics, com uma *firmware* baseada em *FreeRTOS* utilizando as bibliotecas *FreeModbus* e *mbedtls*.

Testes foram realizados com 43 suítes criptográficas para avaliar o desempenho da solução. Observou-se um *overhead* constante para as transações, uma pequena variabilidade na latência das operações e uma queda na taxa de transferência de dados úteis de ao menos 14.55% para suítes que proveem confidencialidade, 32.79% para suítes com confidencialidade baseadas em cifras operadas em modo CCM e 39.75% para suítes com confidencialidade de maior custo computacional.

Foi possível perceber ainda um aumento considerável no tempo necessário para estabelecer conexões com o novo protocolo, o que não deve ser um grande problema em cenário normais de operação mas pode inviabilizar o uso de algumas suítes criptográficas em ambientes críticos. A fim de corrigir

este problema e possivelmente aprimorar esta solução, apresentam-se a seguir algumas sugestões de trabalhos futuro.

6.1 TRABALHOS FUTUROS

No decorrer deste trabalho foram notados alguns tópicos que não puderam ser aqui explorados. Como possíveis trabalhos futuros sugere-se:

- Verificar a viabilidade da utilização de *tokens* para resumir sessões TLS previamente estabelecidas:

  Como apresentado na seção 2.3.2, o protocolo TLS Handshake possibilita que um cliente abrevie a negociação de uma conexão por meio de um mecanismo para resumir sessões anteriores. Originalmente, as informações da sessão deveriam ser guardadas pelo servidor TLS. No contexto da solução proposta, entretanto, o servidor TLS é em geral um sistema embarcado, com restrições de memória maiores do que o cliente da conexão.

  Para este tipo de cenário existe uma extensão do protocolo TLS possibilita que o servidor envie um *token* com as informações da sessão para que o cliente o armazene. Esta otimização pode acelerar o processo de reconexão em casos de falha e viabilizar o uso de Modbus TLS em alguns sistemas.

- Analisar a aplicabilidade de suítes criptográficas que utilizam *Pre-Shared Keys*

  Usualmente são utilizados certificados ou credenciais Kerberos. O uso de *Pre-Shared Keys* no lugar de certificados é especificado pela RFC4279 (ERONEN; TSCHOFENIG, 2005) e apresenta algumas vantagens como não depender de métodos assimétricos para negociar os parâmetros da conexão. Entretanto, a segurança para a distribuição e gerência das chaves pode ser novo problema.

- Avaliar a possibilidade de basear a solução em DTLS:

  A especificação do protocolo Modbus TCP requer sua implementação sobre o TCP e não cita a possibilidade da utilização de *User Datagram Protocol* (UDP) para transportar os quadros do protocolo Modbus. Esta modificação reduziria o *overhead* do protocolo sem grandes alterações em suas funcionalidades, pois a camada de aplicação do protocolo Modbus já realiza a confirmação da entrega das requisições e a fragmentação dos quadros só é necessária em redes com MTU menor que o tamanho máximo do quadro Modbus TCP, de 256 bytes.

  De fato, existe uma especificação não oficial de um protocolo Modbus UDP[1], a porta UDP 502 está reservada para o protocolo Modbus (TOUCH *et al.*, 2018), existem algumas implementações do protocolo em PHP[2], Java[3], C#[4] e .Net[5], mas o padrão nunca foi estabelecido pela Modbus Organization.

  Datagram Transport Layer Security (DTLS) é um protocolo que provê privacidade e autenticidade para protocolos baseados em UDP, trazendo mesmas garantias de segurança do protocolo TLS para comunicações baseadas em datagramas (RESCORLA; MODADUGU, 2012). Sua utilização no lugar TLS para esta solução, criando um protocolo Modbus DTLS, geraria um *overhead* menor e possivelmente um *goodput* maior.

  Adicionalmente, alguns problemas de segurança decorrentes do protocolo TCP, como os ataques do tipo *TCP SYN Flood*, poderiam ser resolvidos por meio desta modificação.

[1] `http://jamod.sourceforge.net/kbase/modbus_udp.html`
[2] `https://github.com/adduc/phpmodbus/`
[3] `https://sourceforge.net/projects/modbus4j/files/modbus4j/`
[4] `https://code.google.com/archive/p/free-dotnet-modbus/`
[5] `https://github.com/highfield/cetdevelop`

REFERÊNCIAS

A further update on SHA-1 certificates in Chrome. 2017. [Online; accessed 14. Jun. 2018]. Disponível em: <https://www.chromium.org/Home/chromium-security/education/tls/sha-1>.

Arm Limited. **GNU Arm Embedded Toolchain**. dec 2017. Disponível em: <https://developer.arm.com/open-source/gnu-toolchain/gnu-rm>.

ARM Limited. **mbedTLS**. aug 2017. Disponível em: <https://tls.mbed.org/>.

AUDET, F. **The Use of the SIPS URI Scheme in the Session Initiation Protocol (SIP)**. [S.l.], October 2009.

BELLARE, Mihir. New proofs for nmac and hmac: Security without collision-resistance. In: SPRINGER. **Annual International Cryptology Conference**. [S.l.], 2006. p. 602–619.

BETAFRED. **Microsoft Security Advisory 2862973**. Aug 2013. [Online; accessed 14. Jun. 2018]. Disponível em: <https://docs.microsoft.com/en-us/security-updates/SecurityAdvisories/2014/2862973>.

BETAFRED. **Microsoft Security Advisory 4010323**. May 2017. [Online; accessed 14. Jun. 2018]. Disponível em: <https://docs.microsoft.com/en-us/security-updates/securityadvisories/2017/4010323>.

CLOCK_GETRES(2) Linux Programmer's Manual. 4.14. ed. [S.l.], Sep 2017.

CREERY, A; BYRES, EJ. Industrial cybersecurity for power system and scada networks. In: IEEE. **Petroleum and Chemical Industry Conference, 2005. Industry Applications Society 52nd Annual**. [S.l.], 2005. p. 303–309.

DIERKS, T.; RESCORLA, E. **The Transport Layer Security (TLS) Protocol Version 1.2**. [S.l.], August 2008. `http://www.rfc-editor.org/rfc/rfc5246.txt`. Disponível em: <http://www.rfc-editor.org/rfc/rfc5246.txt>.

Eclipse Foundation. **Eclipse IDE**. dec 2017. Disponível em: <https://www.eclipse.org>.

EDDY, W. **TCP SYN Flooding Attacks and Common Mitigations**. [S.l.], August 2007.

ERONEN, P.; TSCHOFENIG, H. **Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)**. [S.l.], December 2005. `http://www.rfc-editor.org/rfc/rfc4279.txt`. Disponível em: <http://www.rfc-editor.org/rfc/rfc4279.txt>.

FELDMAN, Stuart. **GNU Make**. jun 2016. Disponível em: <https://www.gnu.org-/software/make/>.

FLOYD, S. **Metrics for the Evaluation of Congestion Control Mechanisms**. [S.l.], March 2008. `http://www.rfc-editor.org/rfc/rfc5166.txt`. Disponível em: <http://www.rfc-editor.org/rfc/rfc5166.txt>.

FOVINO, Igor Nai; CARCANO, Andrea; MASERA, Marcelo; TROMBETAA, Alberto. Design and implementation of a secure modbus protocol. **Critical Infrastructure Protection**, Springer, v. 3, p. 83–96, 2009.

FOVINO, Igor Nai; CARCANO, Andrea; MASERA, Marcelo; TROMBETTA, Alberto. An experimental investigation of malware attacks on scada systems. **International Journal of Critical Infrastructure Protection**, Elsevier, v. 2, n. 4, p. 139–145, 2009.

GOLEMON, Sara. **PHP: rfc:release-md5-deprecation**. May 2017. [Online; accessed 14. Jun. 2018]. Disponível em: <https://wiki.php.net/rfc/release-md5-deprecation>.

Google. **BoringSSL**. nov 2017. Disponível em: <https://boringssl.googlesource-.com/boringssl>.

HAYES, Garrett; EL-KHATIB, Khalil. Securing modbus transactions using hash-based message authentication codes and stream transmission control protocol. In: IEEE. **Communications and Information Technology (ICCIT), 2013 Third International Conference on**. [S.l.], 2013. p. 179–184.

HOFFMAN, P. **SMTP Service Extension for Secure SMTP over TLS**. [S.l.], January 1999.

HUITSING, Peter; CHANDIA, Rodrigo; PAPA, Mauricio; SHENOI, Sujeet. Attack taxonomies for the modbus protocols. **International Journal of Critical Infrastructure Protection**, v. 1, p. 37 – 44, 2008. ISSN 1874-5482. Disponível em: <http://www.sciencedirect.com/science/article/pii/S187454820800005X>.

KOTULIAK, I.; RYBáR, P.; TRúCHLY, P. Performance comparison of ipsec and tls based vpn technologies. In: **2011 9th International Conference on Emerging eLearning Technologies and Applications (ICETA)**. [S.l.: s.n.], 2011. p. 217–221.

KRAWCZYK, Hugo; BELLARE, Mihir; CANETTI, Ran. **HMAC: Keyed-Hashing for Message Authentication**. [S.l.], February 1997. `http://www.rfc-editor.org/rfc/rfc2104.txt`. Disponível em: <http://www.rfc-editor.org/rfc/rfc2104-.txt>.

KUROSE, James F.; ROSS, Keith W. **Computer Networking: A Top-Down Approach (5th Edition)**. [S.l.]: Pearson, 2009. ISBN 0136079679.

MALHOTRA, Aanchal; COHEN, Isaac E; BRAKKE, Erik; GOLDBERG, Sharon. Attacking the network time protocol. In: **NDSS**. [S.l.: s.n.], 2016.

MANSLEY, Kieran. **Re: [lwip-devel] SCTP support in lwip, any plan?** Oct. 2008. Mensagem enviada à lista de email de desenvolvedores da lwIP. Disponível em: <https://lists.nongnu.org/archive/html/lwip-devel/2008-10/msg00054-.html>.

MCGREW, D.; BAILEY, D. **AES-CCM Cipher Suites for Transport Layer Security (TLS)**. [S.l.], July 2012.

MILLS, D.; MARTIN, J.; BURBANK, J.; KASCH, W. **Network Time Protocol Version 4: Protocol and Algorithms Specification**. [S.l.], June 2010. `http://www.rfc-editor.org/rfc/rfc5905.txt`. Disponível em: <http://www.rfc-editor-.org/rfc/rfc5905.txt>.

Modbus FAQ. About the modbus organization. **FAQ. Modbus Organization inc.**, 2017.

Modbus FAQ. About the protocol. **FAQ. Modbus Organization inc.**, 2017.

MODBUS, IDA. Modbus application protocol specification v1.1b. **North Grafton, Massachusetts (www.modbus.org/specs.php)**, 2006.

Modbus Org. **MODBUS over serial line specification & implementation guide V1.02**. 2006.

MODICON, I. **Industrial automation systems, 1996 Modicon modbus protocol**. [S.l.].

MONITOR, ICS-CERT. **November/December**. [S.l.]: NCCIC, 2017.

National Institute of Standards and Technology. **FIPS PUB 180-4: Security requirements for cryptographic modules**. Gaithersburg, MD, USA: National Institute for Standards and Technology, 2002. http://dx.doi.org/10.6028/NIST.FIPS.140-2. Disponível em: <http://dx.doi.org/10.6028/NIST.FIPS.140-2>.

Nikos Mavrogiannopoulos. **GnuTLS**. oct 2017. Disponível em: <https://www.gnutls.org/>.

NIR, Yoav; SALZ, Rich; SULLIVAN, Nick. Tls cipher suite registry. **The Internet Assigned Numbers Authority (IANA)**, May 2018. Disponível em: <https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml>.

OpenBSD Foundation. **LibreSSL**. nov 2017. Disponível em: <http://www.libressl.org/>.

OpenSSL Management Committee. **OpenSSL**. nov 2017. Disponível em: <https://www.openssl.org>.

OPPLIGER, Rolf. **SSL and TLS: Theory and Practice**. [S.l.]: Artech House, 2009.

OPPLIGER, Rolf. **SSL and TLS: Theory and Practice**. [S.l.]: Artech House, 2016.

Real Time Engineers Ltd. **FreeRTOS**. dec 2017. Disponível em: <http://www.freertos.org>.

RESCORLA, E. **HTTP Over TLS**. [S.l.], May 2000. Disponível em: <http://www.rfc-editor.org/rfc/rfc2818.txt>.

RESCORLA, E.; MODADUGU, N. **Datagram Transport Layer Security Version 1.2**. [S.l.], January 2012. http://www.rfc-editor.org/rfc/rfc6347.txt. Disponível em: <http://www.rfc-editor.org/rfc/rfc6347.txt>.

RISTIC, Ivan. **Bulletproof SSL and TLS: Understanding and Deploying SSL/-TLS and PKI to Secure Servers and Web Applications**. [S.l.]: Feisty Duck, 2014.

ROSENBERG, J.; SCHULZRINNE, H.; CAMARILLO, G.; JOHNSTON, A.; PETERSON, J.; SPARKS, R.; HANDLEY, M.; SCHOOLER, E. **SIP: Session Initiation Protocol**. [S.l.], June 2002. Disponível em: <http://www.rfc-editor.org/rfc/rfc3261.txt>.

SASCHA, Wildner. **git: kernel: Remove our ancient SCTP support.** Jan. 2015. Disponível em: <http://lists.dragonflybsd.org/pipermail/commits/2015-January-/417496.html>.

Schneider Automation. Modbus messaging on tcp/ip implementation guide v1. 0b. **MODBUS Organization, last accessed June**, p. 46, 2006.

SHAHZAD, Aamir; LEE, Malrey; LEE, Young-Keun; KIM, Suntae; XIONG, Naixue; CHOI, Jae-Young; CHO, Younghwa. Real time modbus transmissions and cryptography security designs and enhancements of protocol sensitive information. **Symmetry**, Multidisciplinary Digital Publishing Institute, v. 7, n. 3, p. 1176–1210, 2015.

SHIREY, R. **Internet Security Glossary**. [S.l.], May 2000.

STEWART, R. **Stream Control Transmission Protocol**. [S.l.], September 2007. `http://www.rfc-editor.org/rfc/rfc4960.txt`. Disponível em: <http://www-.rfc-editor.org/rfc/rfc4960.txt>.

STMicroelectronics. **STM32CubeMX**. may 2018. Disponível em: <http://www-.st.com/en/development-tools/stm32cubemx.html>.

STOUFFER, Keith; FALCO, Joe; SCARFONE, Karen. Guide to industrial control systems (ics) security. **NIST special publication**, v. 800, n. 82, p. 16–16, 2011.

THE end of SHA-1 on the Public Web. Feb 2017. [Online; accessed 14. Jun. 2018]. Disponível em: <https://blog.mozilla.org/security/2017/02/23/the-end-of-sha-1-on-the-public-web>.

TOUCH, Joe; LEAR, E; MANKIN, A; KOJO, M; ONO, K; STIEMERLING, M; EGGERT, L; MELNIKOV, A; EDDY, W; ZIMMERMANN, A; TRAMMELL, B; IYENGAR, J; TUEXEN, M; KOHLER, E; NISHIDA, Y. Service name and transport protocol port number registry. **The Internet Assigned Numbers Authority (IANA)**, 2018.

VAUDENAY, Serge. **A classical introduction to cryptography: Applications for communications security**. [S.l.]: Springer Science & Business Media, 2006.

wolfSSL Inc. **wolfSSL**. oct 2017. Disponível em: <https://www.wolfssl.com>.

APÊNDICE A - ALTERAÇÕES DA LIBMODBUS

Neste apêndice serão descritas as modificações feitas ao sistema de compilação e o novo *backend* criado para a biblioteca *libmodbus*.

A.1 SISTEMA DE COMPILAÇÃO

A adição de comunicação segura ao protocolo Modbus não segue o padrão do protocolo descrito por Modbus (2006) e foi, portanto, adicionado como uma opção em tempo de compilação. O projeto da *libmodbus* utiliza o *GNU Build System Autotools* para compilação. A modificação feita no arquivo `configure.ac` para adicionar a nova opção pode ser observada na Listagem A.1.

```
1  diff --git a/configure.ac b/configure.ac
2  index 4fa41eb..0df6bcb 100644
3  --- a/configure.ac
4  +++ b/configure.ac
5  @@ -162,6 +162,42 @@ AC_CONFIG_FILES([
6           libmodbus.pc
7   ])
8
9  +AC_ARG_ENABLE([tls],
10 +    AS_HELP_STRING([--enable-tls=[TLS_IMPLEMENTATION]],
11 +    [Build with SSL/TLS support (default: no)]),
12 + [
13 +   case "${enable_tls}" in
14 +     yes|openssl)
15 +       enable_tls=openssl
16 +       ;;
17 +     mbedtls)
18 +       enable_tls=mbedtls
19 +       ;;
20 +     no)
21 +       enable_tls=no
```

```
22 +        ;;
23 +      *)
24 +      AC_MSG_ERROR([Invalid value ${enable_tls} for --enable-tls])
25 +        ;;
26 +      esac
27 + ]
28 +     ,[enable_tls=no])
29 +
30 +AM_CONDITIONAL(BUILD_TLS, [test $enable_tls != no])
31 +
32 +AS_IF([test $enable_tls == openssl],
33 +     [AC_CHECK_LIB([ssl], [SSL_new])]
34 +     [AC_CHECK_LIB([crypto], [CRYPTO_new_ex_data])]
35 +     [AC_DEFINE([USE_TLS], [1], [Define if SSL/TLS feature are
    ↪ enabled])]
36 +     [AC_DEFINE([USE_OPENSSL], [1], [Define if using openssl for
    ↪ SSL/TLS support])],)
37 +
38 +AS_IF([test $enable_tls == mbedtls],
39 +     [AC_CHECK_LIB([mbedtls], [mbedtls_net_bind])]
40 +     [AC_CHECK_LIB([mbedx509], [mbedtls_x509_get_name])]
41 +     [AC_CHECK_LIB([mbedcrypto], [mbedtls_pk_parse_key])]
42 +     [AC_DEFINE([USE_TLS], [1], [Define if SSL/TLS feature are
    ↪ enabled])]
43 +     [AC_DEFINE([USE_MBEDTLS], [1], [Define if using mbedTLS for
    ↪ SSL/TLS support])],)
44 +
45  AC_OUTPUT
46  AC_MSG_RESULT([
47          $PACKAGE $VERSION
48 @@ -176,6 +212,7 @@ AC_MSG_RESULT([
49          cflags:                 ${CFLAGS}
50          ldflags:                ${LDFLAGS}
51
52 +        tls:                    ${enable_tls}
53          documentation:          ${ac_libmodbus_build_doc}
```

```
54          tests:                      ${enable_tests}
55  ])
56
57 --
```

Código A.1: Patch aplicado ao aquivo configure.ac para adição da opção de compilção das funcionalidades relacionadas à comunicação segura

Na linha 9 os argumentos `--enable-tls` e `--disable-tls` são definidos para o *script* de configuração. Por padrão, o suporte a comunicações seguras está desabilitado. A opção possibilita a escolha da biblioteca que irá fornecer a implementação do protocolo TLS, utilizado a sintaxe `--enable-tls=TLS_IMPLEMENTATION`. O argumento informado a opção é verificado entre as linhas 13 a 26, sendo possível optar entre as bibliotecas OpenSSL e mbedTLS, com a primeira utilizada como padrão caso nenhuma implementação seja informada, conforme a linha 14.

A linha 30 define um condicional para ser utilizado nos arquivos `Makefile.am` chamado `BUILD_TLS` que indica se a compilação incluirá ou não a comunicação segura.

De acordo com a implementação escolhida para o protocolo TLS verifica-se pelas existencia das bibliotecas necessárias no sistema entre as linhas 32 a 36 para a compilação com OpenSSL, e entre as linha 38 a 43 para a compilação com mbedTLS. Se as dependências necessárias para a implementação escolhida forem satisfeitas, a macro `USE_TLS` será definida no arquivo `config.h`, conforme a linha 35 ou 42, e as macros `USE_OPENSSL` ou `USE_MBEDTLS` serão definidas de acordo com a implementação escolhida, conforme as linhas 36 e 43

```
1 diff --git a/src/Makefile.am b/src/Makefile.am
2 index 551fe43..db3222e 100644
3 --- a/src/Makefile.am
4 +++ b/src/Makefile.am
5 @@ -22,6 +22,10 @@ libmodbus_la_SOURCES = \
```

```
6           modbus-tcp-private.h \
7           modbus-version.h
8
9  +if BUILD_TLS
10 +libmodbus_la_SOURCES += modbus-tls.h modbus-tls-private.h
11 +endif
12 +
13  libmodbus_la_LDFLAGS = -no-undefined \
14           -version-info $(LIBMODBUS_LT_VERSION_INFO)
15
16 @@ -37,6 +41,10 @@ endif
17  libmodbusincludedir = $(includedir)/modbus
18  libmodbusinclude_HEADERS = modbus.h modbus-version.h modbus-rtu.h
      ↪  modbus-tcp.h
19
20 +if BUILD_TLS
21 +libmodbusinclude_HEADERS += modbus-tls.h
22 +endif
23 +
24  DISTCLEANFILES = modbus-version.h
25  EXTRA_DIST += modbus-version.h.in
26  CLEANFILES = *~
27 --
```

Código A.2: Patch aplicado ao aquivo src/Makefile.am para adição da opção de compilção das funcionalidades relacionadas à comunicação segura

Na listagem A.2 apresenta-se um *patch* com as modificações realizadas no arquivo `src/Makefile.am`. Nas linhas 9 a 11 os cabeçalhos necessários para a compilação com as funcionalidades de segurança são adicionados à compilação, observando o condicional `BUILD_TLS`, e nas linhas 20 a 22 os cabeçalhos a serem instalados são incluidos.

A.2 *BACKEND* PARA MODBUS TLS

Um novo *backend* foi desenvolvido para que a biblioteca *libmodbus* possa se comunicar de forma segura. Este *backend*, nomeado como `_modbus_tls_backend`, reutiliza alguns dos métodos do *backend* TCP para evitar a duplicidade de códigos com a mesma função. Desta forma, apenas os métodos `send`, `recv`, `connect`, `close`, `flush` e `select` foram reimplementados.

Os métodos dos *backends* TCP e RTU são declarados respectivamente nos arquivos `modbus-tcp.c` e `modbus-rtu.c`, sempre com o atributo `static`, de forma que tais métodos só podem ser diretamente invocados no mesmo arquivo em que foram declarados. Desta forma, para que fosse possível reutilização de métodos do *backend* TCP foi necessário adicionar os métodos do *backend* TLS no arquivo `modbus-tcp.c` ao invés de criar um novo arquivo `modbus-tls.c`.

```
1  #if defined(USE_TLS)
2  static ssize_t _modbus_tls_send(modbus_t *ctx, const uint8_t *req,
       ↪ int req_length)
3  {
4    int ret;
5    modbus_tls_t *ctx_tls = (modbus_tls_t *)ctx->backend_data;
6  #if defined(USE_OPENSSL)
7    fd_set fds;
       .
       .
       .
42 #elif defined(USE_MBEDTLS)
43   do {
44     ret = mbedtls_ssl_write(&ctx_tls->ctx, req, req_length);
45   } while(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
       ↪ MBEDTLS_ERR_SSL_WANT_WRITE);
46 #endif
47
48   return ret;
49 }
```

```
50 #endif
```

Código A.3: Implementação do método `send` para comunicações TLS no arquivo `modbus-tcp.c`

```
1  #if defined(USE_TLS)
2  static ssize_t _modbus_tls_send(modbus_t *ctx, const uint8_t *req,
       ↪ int req_length)
3  {
4    int ret;
5    modbus_tls_t *ctx_tls = (modbus_tls_t *)ctx->backend_data;
6  #if defined(USE_OPENSSL)
7    fd_set fds;
       .
       .
       .
34 #elif defined(USE_MBEDTLS)
35   do {
36     ret = mbedtls_ssl_write(&ctx_tls->ctx, req, req_length);
37   } while(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
       ↪ MBEDTLS_ERR_SSL_WANT_WRITE);
38 #endif
39
40   return ret;
41 }
42 #endif
```

Código A.4: Implementação do método `recv` para comunicações TLS no arquivo `modbus-tcp.c`

A implementação dos métodos `send` e `recv` utilizados respectivamente para escrita e leitura podem ser observados nas Listagens A.3 e A.4. Estes métodos chamam o método de leitura ou escrita da biblioteca que fornece a implementação do protocolo TLS, observando o valor de retorno para erros que indiquem que o protocolo TLS necessita trocar mais informações com o outro par da comunicação, como os erros `MBEDTLS_ERR_SSL_WANT_READ`

e `MBEDTLS_ERR_SSL_WANT_WRITE` da biblioteca mbedTLS. Ambos os erros podem acontecer tanto em operação leitura quanto em operações escrita, caso a implementação do protocolo decida que se faz necessário renegociar as chaves utilizadas na comunicação, por exemplo.

```
1  #if defined(USE_TLS)
2  static int _modbus_tls_connect(modbus_t *ctx)
3  {
4    int ret;
5    modbus_tls_t *ctx_tls = (modbus_tls_t *)ctx->backend_data;
6    struct timeval tv = ctx->response_timeout;
7    fd_set fds;
8  #if defined(USE_OPENSSL)
9    X509 *cert;
10 #endif
11
12   ret = _modbus_tcp_connect(ctx);
13   if(ret < 0) {
14     return ret;
15   }
16
17 #if defined(USE_OPENSSL)
       .
       .
       .
84 #elif defined(USE_MBEDTLS)
85   ret = mbedtls_ssl_config_defaults(&ctx_tls->cfg,
       ↪ MBEDTLS_SSL_IS_CLIENT, MBEDTLS_SSL_TRANSPORT_STREAM,
       ↪ MBEDTLS_SSL_PRESET_DEFAULT);
86
87   if(ret != 0) {
88     mbedtls_ssl_session_reset(&ctx_tls->ctx);
89     mbedtls_net_free(&ctx->s);
90     return -1;
91   }
92
```

```
93    mbedtls_ssl_conf_authmode(&ctx_tls->cfg,
       ↪ MBEDTLS_SSL_VERIFY_REQUIRED);
94    mbedtls_ssl_conf_rng(&ctx_tls->cfg, mbedtls_ctr_drbg_random, &
       ↪ ctx_tls->drbg);
95    mbedtls_ssl_setup(&ctx_tls->ctx, &ctx_tls->cfg);
96
97    mbedtls_ssl_set_bio(&ctx_tls->ctx, &ctx->s, mbedtls_net_send,
       ↪ mbedtls_net_recv, NULL);
98
99    do {
100     ret = mbedtls_ssl_handshake(&ctx_tls->ctx);
101
102     if(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
       ↪ MBEDTLS_ERR_SSL_WANT_WRITE) {
103       FD_ZERO(&fds);
104       FD_SET(ctx->s, &fds);
105
106       if(select(ctx->s+1, &fds, &fds, NULL, &tv) <= 0) {
107         mbedtls_ssl_session_reset(&ctx_tls->ctx);
108         mbedtls_net_free(&ctx->s);
109         return -1;
110       }
111     }
112   } while(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
       ↪ MBEDTLS_ERR_SSL_WANT_WRITE);
113
114   if(ret != 0) {
115     mbedtls_ssl_session_reset(&ctx_tls->ctx);
116     mbedtls_net_free(&ctx->s);
117     return -1;
118   }
119 #endif
```

Código A.5: Implementação do método `connect` para comunicações TLS no arquivo `modbus-tcp.c`

A implementação do métdodo `connect` pode ser observada na Lista-

gem A.5. Primeiramente, na linha 12 a conexão TCP é realizada com método `_modbus_tcp_connect` do *backend* TCP. Na linha 85 a estrutura de configuração da sessão TLS é inicializada com as opções padrão para um cliente TLS, sendo verificado por erros em seguida entre as linhas 85 a 85.

Na linha 93 é definido que a autenticação do servidor é obrigatória, de forma que o *handshake* falhará caso o certificado apresentado pelo servidor não seja confiável. A linha 94 configura o método a ser utilizado para geração de números pseudoaleatórios e a linha 95 aplica as configurações à estrutura da sessão TLS.

O *handshake* é realizado na linha 100 e nas linhas 102 a 111 é verificado se a camada TLS tem mais dados a serem transmitidos ou recebidos, sendo utilizado uma chamada ao método `select` para aguardar até que seja possível escrever ou ler no *socket* sem bloqueios. Por fim, verifica-se por erros de *handshake* entre as linhas 114 a 118.

```
1  #if defined(USE_TLS)
2  static void _modbus_tls_close(modbus_t *ctx)
3  {
4    modbus_tls_t *ctx_tls = (modbus_tls_t *)ctx->backend_data;
5
6    if(ctx->s != -1) {
7  #if defined(USE_OPENSSL)
8      SSL_shutdown(ctx_tls->ssl);
9  #elif defined(USE_MBEDTLS)
10     mbedtls_ssl_close_notify(&ctx_tls->ctx);
11 #endif
12     shutdown(ctx->s, SHUT_RDWR);
13     close(ctx->s);
14     ctx->s = -1;
15 #if defined(USE_OPENSSL)
16     SSL_free(ctx_tls->ssl);
17 #endif
18   }
19 }
```

```
20 #endif
```

Código A.6: Implementação do método `close` para comunicações TLS no arquivo `modbus-tcp.c`

Na Listagem A.6 apresenta-se a implementação do método `close`, que envia um alerta de fechamento da conexão pelo protocolo TLS, na linha 8 ou 10, para indicar que a conexão TLS não receberá e nem enviará mais dados. Na linha 12 a chamada ao método `shutdown` indica que o *socket* TCP também não receberá ou enviará dados e, por fim, as estrturas alocadas para a conexão TLS são liberadas na linha 16.

```
1  #if defined(USE_TLS)
2  static int _modbus_tls_flush(modbus_t *ctx)
3  {
4    int rc, rc_sum = 0;
5    char devnull[MODBUS_TCP_MAX_ADU_LENGTH];
6    fd_set rset;
7    struct timeval tv = {.tv_sec = 0, .tv_usec = 0};
8    modbus_tls_t *ctx_tls = (modbus_tls_t *)ctx->backend_data;
9
10 #if defined(USE_OPENSSL)
11   if((rc = SSL_pending(ctx_tls->ssl)) > 0) {
12     rc_sum = SSL_read(ctx_tls->ssl, devnull, rc);
13   }
14 #elif defined(USE_MBEDTLS)
15   if((rc = mbedtls_ssl_get_bytes_avail(&ctx_tls->ctx)) > 0) {
16     rc_sum = mbedtls_ssl_read(&ctx_tls->ctx, devnull, rc);
17   }
18 #endif
19
20   FD_ZERO(&rset);
21   FD_SET(ctx->s, &rset);
22
23   do {
24     rc = select(ctx->s+1, &rset, NULL, NULL, &tv);
25
```

```
26      switch(rc) {
27        case -1:
28          if(errno == EINTR) {
29            continue;
30          }
31          return -1;
32
33        case 1:
34  #if defined(USE_OPENSSL)
35          rc = SSL_read(ctx_tls->ssl, devnull,
    ↪ MODBUS_TCP_MAX_ADU_LENGTH);
36  #elif defined(USE_MBEDTLS)
37          rc_sum = mbedtls_ssl_read(&ctx_tls->ctx, devnull,
    ↪ MODBUS_TCP_MAX_ADU_LENGTH);
38
39  #endif
40          if(rc > 0) {
41            rc_sum += rc;
42          }
43
44          rc = -1;
45      }
46    } while(rc < 0);
47
48    return rc_sum;
49  }
```

Código A.7: Implementação do método `flush` para comunicações TLS no arquivo `modbus-tcp.c`

O método `flush` é detalhado na Listagem A.7. Na linha 11 ou na linha 15 é checado se existe algum dado no *buffer* interno da camada TLS, lendo esta quantidade na linha 12 ou 16. Em seguida, é feito um `select` no *socket* da conexão para verificar se há dados a serem recebidos pelo processo, utilizando zero como valor de *timeout* a fim de que o método `select` não bloqueie. Caso existam dados, uma chamada ao método de leitura é realizada na linha 35 ou

37.

```
1  #if defined(USE_TLS)
2  static int _modbus_tls_select(modbus_t *ctx, fd_set *rset, struct
       ↪ timeval *tv, int length_to_read)
3  {
4    modbus_tls_t *ctx_tls = (modbus_tls_t *)ctx->backend_data;
5
6    /*Check SSL buffer for pending data*/
7  #if defined(USE_OPENSSL)
8    if(SSL_pending(ctx_tls->ssl) > 0) {
9      return 1;
10   }
11 #elif defined(USE_MBEDTLS)
12   if(mbedtls_ssl_get_bytes_avail(&ctx_tls->ctx) > 0) {
13     return 1;
14   }
15 #endif
16
17   /*Run normal TCP select*/
18   return _modbus_tcp_select(ctx, rset, tv, length_to_read);
19 }
20 #endif
```

Código A.8: Implementação do método `select` para comunicações TLS no arquivo `modbus-tcp.c`

Por fim, na Listagem A.8 é apresentada a implementação do método `select`, em que o *buffer* interno do protocolo TLS é checado, na linha 8 ou 12, e em seguida utiliza-se a implementação do método `select` do *backend* TCP (linha 18) para verificar se há dados ainda não lidos pela aplicação para o *socket* informado.

APÊNDICE B - ALTERAÇÕES DA FREEMODBUS

Para que a FreeModbus realize comunicações utilizando o protocolo TLS foi necessário implementar uma camada de portabilidade que utilize os métodos fornecidos pela biblioteca mbedTLS. Como base para esta implementação utilizou-se a adaptação para Linux, que utiliza a interfaces de *sockets* POSIX e é composta por um cabeçalho, `port.h`, e dos fontes `portevent.c`, `portother.c` e `porttcp.c`.

As funções modificadas desta camada de abstração foram `xMBTCPPortInit`, `vMBTCPPortClose`, `xMBPortTCPPool`, `xMBTCPPortSendResponse`, `prvvMBPortReleaseClient` e `prvbMBPortAcceptClient`, todos pertencentes ao arquivo `porttcp.c`. A seguir será descrito o funcionamento de cada um destes métodos.

```
1  BOOL
2  xMBTCPPortInit( USHORT usTCPPort )
3  {
4    CHAR       usPort[6];
5    if( usTCPPort == 0 )
6    {
7      snprintf(usPort, sizeof(usPort), "%hu", MB_TCP_DEFAULT_PORT);
8    }
9    else
10   {
11     snprintf(usPort, sizeof(usPort), "%hu", usTCPPort);
12   }
13
14   mbedtls_entropy_init(&entropy);
15   mbedtls_ctr_drbg_init(&drbg);
16
17   if(mbedtls_ctr_drbg_seed(&drbg, mbedtls_entropy_func, &entropy,
      ↪ NULL, 0) == MBEDTLS_ERR_CTR_DRBG_ENTROPY_SOURCE_FAILED) {
18     goto drbg_fail;
19   }
20
```

```
21  mbedtls_x509_crt_init(&cert);
22
23  if(mbedtls_x509_crt_parse(&cert, SERVER_CERT, strlen(SERVER_CERT
     ↪ ) + 1) != 0) {
24    goto x509_fail;
25  }
26
27  mbedtls_pk_init(&pk);
28
29  if(mbedtls_pk_parse_key(&pk, SERVER_KEY, strlen(SERVER_KEY) + 1,
     ↪  NULL, 0) != 0) {
30    goto pk_fail;
31  }
32
33  if(mbedtls_x509_crt_parse(&cert, CA_CERT, strlen(CA_CERT) + 1)
     ↪ != 0) {
34    goto pk_fail;
35  }
36
37  mbedtls_ssl_config_init(&cfg);
38  mbedtls_ssl_conf_ca_chain(&cfg, cert.next, NULL);
39
40  if(mbedtls_ssl_config_defaults(&cfg, MBEDTLS_SSL_IS_SERVER,
     ↪ MBEDTLS_SSL_TRANSPORT_STREAM, MBEDTLS_SSL_PRESET_DEFAULT) !=
     ↪  0) {
41    goto cfg_fail;
42  }
43
44  mbedtls_ssl_conf_authmode(&cfg, MBEDTLS_SSL_VERIFY_OPTIONAL);
45  mbedtls_ssl_conf_rng(&cfg, mbedtls_ctr_drbg_random, &drbg);
46
47  if(mbedtls_ssl_conf_own_cert(&cfg, &cert, &pk) != 0) {
48    goto cfg_fail;
49  }
50
51  mbedtls_ssl_init(&ctx);
```

```
52
53    if(mbedtls_ssl_setup(&ctx, &cfg) != 0) {
54      goto ssl_fail;
55    }
56
57    //mbedtls_net_init(&srv);// done by dhcp_thread
58
59    if(mbedtls_net_bind(&srv, NULL, usPort, MBEDTLS_NET_PROTO_TCP)
       ↪ != 0) {
60      goto net_fail;
61    }
62
63    FD_ZERO(&allset);
64    FD_SET(srv.fd, &allset);
65
66    return TRUE;
67
68  net_fail:
69    mbedtls_net_free(&srv);
70  ssl_fail:
71    mbedtls_ssl_session_reset(&ctx);
72  cfg_fail:
73    mbedtls_ssl_config_free(&cfg);
74  pk_fail:
75    mbedtls_pk_free(&pk);
76  x509_fail:
77    mbedtls_x509_crt_free(&cert);
78  drbg_fail:
79    mbedtls_ctr_drbg_free(&drbg);
80
81    mbedtls_entropy_free(&entropy);
82
83    return FALSE;
84  }
```

Código B.1: Implementação do método `xMBTCPPortInit`

O método `xMBTCPPortInit`, listado no Còdigo B.1, inicializa a pilha Modbus. A porta a ser escutada pelo servidor Modbus fornecida como argumento ao método é transformada em uma *string*, como requerido pelo método `mbedtls_net_bind`. As estruturas `mbedtls_entropy_context` e `mbedtls_ctr_drbg_context` são inicializadas nas linhas 14 e 15 e o gerador de números pseudoaleatórios é alimentado na linha 17.

Em seguida o certificado local, sua chave e o certificado do CA são carregados respectivamente nas linhas 23, 29 e 33, e a cadeia de certificados confiáveis é montada na linha 38.

Na linha 40 a estrutura de configurações da sessão TLS é inicializada com os valores padrão e na linha 44 configura-se para que o cliente deste servidor TLS também seja autenticado. Na linha 45 é definido o gerador de números pseudoaleatórios a ser utilizado pela sessão TLS e na linha 47 é definido a cadeia de certificados a ser utilizada.

Por fim as configurações são aplicadas à sessão TLS na linha 53 e o servidor começa a escutar por clientes com a chamada a `mbedtls_net_bind`, na linha 59.

```
1  void
2  vMBTCPPortClose(  )
3  {
4    // Close all client sockets.
5    if( cli.fd != INVALID_SOCKET )
6    {
7      prvvMBPortReleaseClient(  );
8    }
9    // Close the listener socket.
10   if( srv.fd != INVALID_SOCKET )
11   {
12     mbedtls_net_free(&srv);
13     mbedtls_ssl_session_reset(&ctx);
14     mbedtls_ssl_config_free(&cfg);
15     mbedtls_pk_free(&pk);
```

```
16      mbedtls_x509_crt_free(&cert);
17      mbedtls_ctr_drbg_free(&drbg);
18      mbedtls_entropy_free(&entropy);
19    }
20  }
```

Código B.2: Implementação do método `vMBTCPPortClose`

O método `vMBTCPPortClose` pode ser observado no Código B.2. Este método desaloca os recursos alocado por `xMBTCPPortInit` e possívelmente por `prvbMBPortAcceptClient`. Entre as linhas 5 a 8 é verificado se o servidor Modbus está conectado a algum cliente, liberando esta conexão com uma chamada ao método `prvvMBPortReleaseClient` caso isso seja verdade. Nas linhas 12 a 18 os recursos alocados nas estruturas `mbedtls_net_context`, `mbedtls_ssl_context`, `mbedtls_ssl_config`, `mbedtls_pk_context`, `mbedtls_x509_crt`, `mbedtls_ctr_drbg_context` e `mbedtls_entropy_context` são liberados.

```
1  BOOL
2  xMBPortTCPPool( void )
3  {
4    int       n;
5    fd_set      fread;
6    struct timeval  tval;
7
8    tval.tv_sec = 0;
9    tval.tv_usec = 5000;
10   int       ret;
11   USHORT      usLength;
12
13   if( cli.fd == INVALID_SOCKET )
14   {
15     /* Accept to client */
16     if( ( n = select( srv.fd + 1, &allset, NULL, NULL, NULL ) ) <
     ↪ 0 )
17     {
```

```
18        return TRUE;
19      }
20
21      if( FD_ISSET( srv.fd, &allset ) )
22      {
23        ( void )prvbMBPortAcceptClient(  );
24      }
25    }
26
27    while( TRUE )
28    {
29      if(cli.fd == INVALID_SOCKET) {
30        mbedtls_ssl_session_reset(&ctx);
31        return TRUE;
32      }
33
34      if(!mbedtls_ssl_get_bytes_avail(&ctx)) {
35        FD_ZERO( &fread );
36        FD_SET( cli.fd, &fread );
37        if( ( ( ret = select( cli.fd + 1, &fread, NULL, NULL, &tval
    ↪ ) ) == SOCKET_ERROR )
38          || !ret )
39        {
40          if(errno == EAGAIN || errno == EINTR) {
41            continue;
42          } else {
43            return FALSE;
44          }
45        }
46      }
47      if( ret > 0 )
48      {
49        if( FD_ISSET( cli.fd, &fread ) )
50        {
51          do {
52            ret = mbedtls_ssl_read(&ctx, &aucTCPBuf[usTCPBufPos],
```

```
    ↪ usTCPFrameBytesLeft);
53        } while(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
    ↪ MBEDTLS_ERR_SSL_WANT_WRITE);
54
55        if(ret <= 0) {
56          do {
57            mbedtls_ssl_close_notify(&ctx);
58          } while(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
    ↪ MBEDTLS_ERR_SSL_WANT_WRITE);
59
60          mbedtls_net_free(&cli);
61          mbedtls_ssl_session_reset(&ctx);
62
63          return TRUE;
64        }
65
66        usTCPBufPos += ret;
67        usTCPFrameBytesLeft -= ret;
68        if( usTCPBufPos >= MB_TCP_FUNC )
69        {
70          /* Length is a byte count of Modbus PDU (function code +
    ↪  data) and the
71           * unit identifier. */
72          usLength = aucTCPBuf[MB_TCP_LEN] << 8U;
73          usLength |= aucTCPBuf[MB_TCP_LEN + 1];
74
75          /* Is the frame already complete. */
76          if( usTCPBufPos < ( MB_TCP_UID + usLength ) )
77          {
78            usTCPFrameBytesLeft = usLength + MB_TCP_UID -
    ↪ usTCPBufPos;
79          }
80          /* The frame is complete. */
81          else if( usTCPBufPos == ( MB_TCP_UID + usLength ) )
82          {
83            ( void )xMBPortEventPost( EV_FRAME_RECEIVED );
```

```
84              return TRUE;
85            }
86            /* This can not happend because we always calculate the
    ↪ number of bytes
87             * to receive. */
88            else
89            {
90              assert( usTCPBufPos <= ( MB_TCP_UID + usLength ) );
91            }
92          }
93        }
94      }
95    }
96    return TRUE;
97  }
```

Código B.3: Implementação do método `xMBPortTCPPool`

O método `xMBPortTCPPool` é chamado por `xMBPortEventGet`, interno à FreeModbus, quando não há nenhum novo evento na fila de eventos da biblioteca. Este método é responsável por aceitar uma nova conexão, caso não exista nenhum cliente conectado, ou receber os dados enviados pelo cliente atualmente conectado.

Primeiramente, na linha 13 é verificado se existe algum cliente conectado. Se não existir, uma chamada a `select` é feita na linha 16 para verificar se há alguma conexão pendente, de forma a evitar bloqueios na chamada à `accept`. O resultado da chamada a `select` é checado com a macro `FD_ISSET`, na linha 21. Caso exista uma conexão pendente, o método `prvbMBPortAcceptClient` é invocado, na linha 23, para tratar esta nova conexão.

Em seguida, entre as linhas 27 a 95 existe um laço que tenta realizar a leitura de um quadro Modbus completo. Nas linhas 29 a 32 verifica-se pela desconexão do cliente atual, liberando os recursos com uma chamada ao

método `mbedtls_ssl_session_reset` na linha 30 caso isso ocorra.

Entre as linhas 34 a 46 é checado se existem dados a serem recebidos na conexão atual. Primeiramente é verificado se há dados não lidos no *buffer* do protocolo TLS, com uma chamada ao método `mbedtls_ssl_get_bytes_avail` na linha 34. Caso não existam dados, checa-se o *buffer* do protocolo TCP, com a utilização do método `select` na linha 37. Se também não existirem dados neste *buffer* o laço é reiniciado pelo comando `continue`, na linha 41.

Se houverem dados ainda não lidos em um dos dois *buffers*, um laço de leitura é executado entre as linhas 51 a 53, com o tratamento de possíveis erros retornados por `mbedtls_ssl_read` entre as linhas 55 a 64. A posição da escrita no *buffer* de recepção do quadro Modbus é então atualizada na linha 66 e a quantidade de dados restantes para completar o quadro é atualizada na linha 67.

Se a posição do *buffer* de recepção do quadro indicar que o tamanho do quadro Modbus já foi recebido, este tamanho é calculado entre as linhas 72 e 73. Por fim, entre as linhas 76 a 91 é verificado se o quadro já foi recebido por completo. Caso isso seja verdade, o evento de quadro recebido é postado na fila de eventos, na linha 83, por meio de uma chamada ao método `xMBPortEventPost`. Do contrário, a quantidade de *bytes* restantes é novamente calculada na linha 78 e a execução retorna ao início da laço para a leitura de mais dados.

```
1 BOOL
2 xMBTCPPortSendResponse( const UCHAR * pucMBTCPFrame, USHORT
    ↪ usTCPLength )
3 {
4   BOOL       bFrameSent = FALSE;
5   BOOL       bAbort = FALSE;
6   int       res;
7   int       iBytesSent = 0;
8   int       iTimeOut = MB_TCP_READ_TIMEOUT;
```

```
9
10  do
11  {
12    res = mbedtls_ssl_write(&ctx, &pucMBTCPFrame[iBytesSent],
     ↪ usTCPLength - iBytesSent);
13
14    if(res < 0)
15      res = -1;
16
17    switch ( res )
18    {
19    case -1:
20      if( iTimeOut > 0 )
21      {
22        iTimeOut -= MB_TCP_READ_CYCLE;
23        vTaskDelay(MB_TCP_READ_CYCLE/1000);
24      }
25      else
26      {
27        bAbort = TRUE;
28      }
29      break;
30    case 0:
31      prvvMBPortReleaseClient(  );
32      bAbort = TRUE;
33      break;
34    default:
35      iBytesSent += res;
36      break;
37    }
38  }
39  while( ( iBytesSent != usTCPLength ) && !bAbort );
40
41  bFrameSent = iBytesSent == usTCPLength ? TRUE : FALSE;
42
43  return bFrameSent;
```

```
44 }
```

Código B.4: Implementação do método `xMBTCPPortSendResponse`

O método `xMBTCPPortSendResponse` pode ser observado no Código B.4. Estre as linhas 10 a 38 há um laço que chama o método `mbedtls_ssl_write` (linha 12) até que todo o quadro passado por argumento seja enviado, algum erro seja detectado ou o envio exceda o tempo máximo para a transmissão.

A detecção de erros é feita entre as linhas 17 a 37. Caso o método `mbedtls_ssl_write` indique algum erro, é verificado se tempo máximo de transmissão já foi atingido (linha 20), e então sinalizado por meio da variável booleana `bAbort` na linha 27 caso isso seja verdade. Do contrário, a *task* dorme por `MB_TCP_READ_TIMEOUT` microsegundos.

Caso a chamada a `mbedtls_ssl_write` indique que nenhum dado foi escrito o cliente é desconectado, na linha 31. Se nenhum erro for detectado, a quantidade de dados escritos é somada. Por fim, na linha 41 é verificado se todo o quandro foi escrito, de forma a gerar o retorno apropriado para o método, na linha 43.

```
1  prvvMBPortReleaseClient(  )
2  {
3    int ret;
4
5    do {
6      ret = mbedtls_ssl_read(&ctx, &aucTCPBuf[0], MB_TCP_BUF_SIZE);
7    } while(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
       ↪ MBEDTLS_ERR_SSL_WANT_WRITE);
8
9    do{
10     ret = mbedtls_ssl_close_notify(&ctx);
11   } while(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
       ↪ MBEDTLS_ERR_SSL_WANT_WRITE);
12
```

```
13   mbedtls_net_free(&cli);
14   mbedtls_ssl_session_reset(&ctx);
15
16   cli.fd = INVALID_SOCKET;
17 }
```

Código B.5: Implementação do método `prvvMBPortReleaseClient`

O Código B.5 apresenta uma listagem da implementação do método `prvvMBPortReleaseClient`, que realiza a desconexão de um cliente. Entre as linhas 5 a 7 é realizado um laço que chamada o método `mbedtls_ssl_read` até que reste nenhum dado não lido no *buffer* da pilha TCP ou da biblioteca mbedTLS. Em seguida, entre as linhas 9 a 11 o método `mbedtls_ssl_close_notify` é invocado para notificar ao cliente que nenhuma dado será escrito ou lido desta conexão TLS. Os recursos da conexão são liberados na linha 13 e a sessão TLS é reinicializada na linha 14. Por fim, na linha 16 garante-se que o descritor do *socket* da conexão possua um número inválido, de forma que a próxima chamada a `xMBPortTCPPool` chame `prvbMBPortAcceptClient`.

```
1  prvbMBPortAcceptClient(  )
2  {
3    int ret;
4
5    if(cli.fd != INVALID_SOCKET) {
6      return FALSE;
7    }
8
9    do {
10     ret = mbedtls_net_accept(&srv, &cli, NULL, 0, NULL);
11   } while(ret == MBEDTLS_ERR_SSL_WANT_READ);
12
13   if(ret != 0) {
14     mbedtls_net_free(&cli);
15     return FALSE;
16   }
17
```

```
18    mbedtls_ssl_set_bio(&ctx, &cli, mbedtls_net_send,
        ↪ mbedtls_net_recv, NULL);
19
20    do {
21      ret = mbedtls_ssl_handshake(&ctx);
22    } while(ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
        ↪ MBEDTLS_ERR_SSL_WANT_WRITE);
23
24    if(ret != 0) {
25      mbedtls_ssl_session_reset(&ctx);
26      mbedtls_net_free(&cli);
27      return FALSE;
28    }
29
30    usTCPBufPos = 0;
31    usTCPFrameBytesLeft = MB_TCP_FUNC;
32
33    return TRUE;
34  }
```

Código B.6: Implementação do método `prvbMBPortAcceptClient`

Por fim, o método `prvbMBPortAcceptClient` é listado no Código B.6. Este método é utilizado para aceitar a conexão de um novo cliente, utilizando uma chamada ao método `mbedtls_net_accept` na linha 10. O *socket* da sessão TLS é então definido na linha 18 e o *handshake* é realizado entre as linhas 20 a 22. Entre as linhas 24 e 28 verifica-se o resultado do *handshake*, inicializando as variáveis globais `usTCPBufPos` e `usTCPFrameBytesLeft` caso ocorra nenhum erro.

APÊNDICE C - SAÍDA DO PROGRAMA DE TESTES

Quadro 4: Tempo de conexão para as suítes criptográficas testadas

Suíte	Tempo de Conexão (ms)
TCP	1.157395
TLS-RSA-WITH-NULL-MD5	1101.243593
TLS-RSA-WITH-NULL-SHA	1095.212708
TLS-RSA-WITH-NULL-SHA256	1096.880676
TLS-ECDHE-RSA-WITH-NULL-SHA	3476.506457
TLS-RSA-WITH-DES-CBC-SHA	1105.787551
TLS-RSA-WITH-3DES-EDE-CBC-SHA	1094.456978
TLS-DHE-RSA-WITH-DES-CBC-SHA	6032.204893
TLS-DHE-RSA-WITH-3DES-EDE-CBC-SHA	6042.888227
TLS-RSA-WITH-AES-128-CBC-SHA	1102.636041
TLS-DHE-RSA-WITH-AES-128-CBC-SHA	6035.074894
TLS-RSA-WITH-AES-256-CBC-SHA	1108.055885
TLS-DHE-RSA-WITH-AES-256-CBC-SHA	6040.181925
TLS-RSA-WITH-AES-128-CBC-SHA256	1098.957239
TLS-RSA-WITH-AES-256-CBC-SHA256	1095.097916
TLS-RSA-WITH-CAMELLIA-128-CBC-SHA	1095.418698
TLS-DHE-RSA-WITH-CAMELLIA-128-CBC-SHA	6042.136247
TLS-DHE-RSA-WITH-AES-128-CBC-SHA256	6030.953487
TLS-DHE-RSA-WITH-AES-256-CBC-SHA256	6036.908383
TLS-RSA-WITH-CAMELLIA-256-CBC-SHA	1096.347187
TLS-DHE-RSA-WITH-CAMELLIA-256-CBC-SHA	6032.558747
TLS-RSA-WITH-AES-128-GCM-SHA256	1096.446614
TLS-RSA-WITH-AES-256-GCM-SHA384	1112.888593
TLS-DHE-RSA-WITH-AES-128-GCM-SHA256	6047.732446
TLS-DHE-RSA-WITH-AES-256-GCM-SHA384	6049.934060
TLS-RSA-WITH-CAMELLIA-128-CBC-SHA256	1103.809271

Suíte	**Tempo de Conexão (ms)**
TLS-DHE-RSA-WITH-CAMELLIA-128-CBC-SHA256	6038.011143
TLS-RSA-WITH-CAMELLIA-256-CBC-SHA256	1102.686562
TLS-DHE-RSA-WITH-CAMELLIA-256-CBC-SHA256	6057.344425
TLS-ECDHE-RSA-WITH-3DES-EDE-CBC-SHA	3517.059009
TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA	3547.802863
TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA	3546.866978
TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA256	3512.084322
TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA384	3455.522603
TLS-ECDHE-RSA-WITH-AES-128-GCM-SHA256	3568.283957
TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384	3478.661248
TLS-ECDHE-RSA-WITH-CAMELLIA-128-CBC-SHA256	3540.049946
TLS-ECDHE-RSA-WITH-CAMELLIA-256-CBC-SHA384	3501.175988
TLS-ECDHE-RSA-WITH-CAMELLIA-128-GCM-SHA256	3506.567342
TLS-ECDHE-RSA-WITH-CAMELLIA-256-GCM-SHA384	3527.431613
TLS-RSA-WITH-AES-128-CCM	1098.459114
TLS-RSA-WITH-AES-256-CCM	1095.377500
TLS-DHE-RSA-WITH-AES-128-CCM	6021.800154
TLS-DHE-RSA-WITH-AES-256-CCM	6038.617290
TLS-RSA-WITH-AES-128-CCM-8	1095.944843
TLS-RSA-WITH-AES-256-CCM-8	1097.387031
TLS-DHE-RSA-WITH-AES-128-CCM-8	6042.940154
TLS-DHE-RSA-WITH-AES-256-CCM-8	6038.043904

Quadro 5: Latência, *jitter* e *Goodput* para operações de leitura de bobinas (0x01)

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TCP	0.847240	0.999133	14.720208	137.3291	244
TLS-RSA-WITH-NULL-MD5	1.044531	1.168570	14.600677	137.3291	208
TLS-RSA-WITH-NULL-SHA	1.080313	1.218312	17.320364	167.84668	200
TLS-RSA-WITH-NULL-SHA256	1.305937	1.469770	17.680104	137.3291	166
TLS-ECDHE-RSA-WITH-NULL-SHA	1.097396	1.220309	17.423177	137.3291	200
TLS-RSA-WITH-DES-CBC-SHA	1.787343	1.938580	15.525886	61.03516	125
TLS-RSA-WITH-3DES-EDE-CBC-SHA	2.987448	3.097159	22.033750	106.81152	78
TLS-DHE-RSA-WITH-DES-CBC-SHA	1.800469	1.947247	20.656407	106.81152	125
TLS-DHE-RSA-WITH-3DES-EDE-CBC-SHA	3.000052	3.101423	21.915469	106.81152	78
TLS-RSA-WITH-AES-128-CBC-SHA	1.370208	1.534516	17.763281	106.81152	159
TLS-DHE-RSA-WITH-AES-128-CBC-SHA	1.359479	1.523290	15.384895	61.03516	160
TLS-RSA-WITH-AES-256-CBC-SHA	1.402709	1.568367	16.939063	137.3291	155
TLS-DHE-RSA-WITH-AES-256-CBC-SHA	1.420260	1.565401	16.987344	137.3291	155
TLS-RSA-WITH-AES-128-CBC-SHA256	1.606042	1.748896	16.257292	76.29395	139
TLS-RSA-WITH-AES-256-CBC-SHA256	1.636407	1.795964	18.030990	106.81152	135
TLS-RSA-WITH-CAMELLIA-128-CBC-SHA	1.686458	1.826565	15.968542	106.81152	133

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TLS-DHE-RSA-WITH-CAMELLIA-128-CBC-SHA	1.677396	1.827125	18.919167	671.38672	133
TLS-DHE-RSA-WITH-AES-128-CBC-SHA256	1.600677	1.750600	20.098802	122.07031	139
TLS-DHE-RSA-WITH-AES-256-CBC-SHA256	1.643802	1.798715	16.009791	137.3291	135
TLS-RSA-WITH-CAMELLIA-256-CBC-SHA	1.820417	1.969104	18.191042	106.81152	123
TLS-DHE-RSA-WITH-CAMELLIA-256-CBC-SHA	1.808020	1.967473	17.296770	122.07031	124
TLS-RSA-WITH-AES-128-GCM-SHA256	1.455000	1.608110	15.991146	137.3291	151
TLS-RSA-WITH-AES-256-GCM-SHA384	1.522135	1.659404	17.969584	91.55273	147
TLS-DHE-RSA-WITH-AES-128-GCM-SHA256	1.474948	1.610598	17.205833	122.07031	151
TLS-DHE-RSA-WITH-AES-256-GCM-SHA384	1.536406	1.659751	15.858021	137.3291	147
TLS-RSA-WITH-CAMELLIA-128-CBC-SHA256	1.902708	2.053068	19.968542	91.55273	118
TLS-DHE-RSA-WITH-CAMELLIA-128-CBC-SHA256	1.900573	2.056364	15.645104	122.07031	118
TLS-RSA-WITH-CAMELLIA-256-CBC-SHA256	2.063855	2.204506	16.126458	137.3291	110
TLS-DHE-RSA-WITH-CAMELLIA-256-CBC-SHA256	2.046250	2.202389	18.405937	106.81152	110
TLS-ECDHE-RSA-WITH-3DES-EDE-CBC-SHA	2.973541	3.102034	21.824323	76.29395	78
TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA	1.374479	1.533167	17.637187	76.29395	159
TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA	1.407292	1.567085	15.217291	137.3291	155
TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA256	1.607083	1.750327	20.481355	122.07031	139
TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA384	2.588073	2.720836	16.451302	152.58789	89
TLS-ECDHE-RSA-WITH-AES-128-GCM-SHA256	1.451979	1.608846	18.864532	106.81152	151

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384	1.522031	1.658808	18.012604	106.81152	147
TLS-ECDHE-RSA-WITH-CAMELLIA-128-CBC-SHA256	1.900677	2.055561	16.162240	122.07031	118
TLS-ECDHE-RSA-WITH-CAMELLIA-256-CBC-SHA384	2.961770	3.124988	22.826355	106.81152	78
TLS-ECDHE-RSA-WITH-CAMELLIA-128-GCM-SHA256	1.792500	1.945954	20.447344	76.29395	125
TLS-ECDHE-RSA-WITH-CAMELLIA-256-GCM-SHA384	1.974115	2.117681	18.069218	137.3291	115
TLS-RSA-WITH-AES-128-CCM	1.372135	1.496077	17.804427	137.3291	163
TLS-RSA-WITH-AES-256-CCM	1.472292	1.599861	19.584792	122.07031	152
TLS-DHE-RSA-WITH-AES-128-CCM	1.361719	1.496853	15.643906	1373.29102	163
TLS-DHE-RSA-WITH-AES-256-CCM	1.450416	1.601374	17.893489	122.07031	152
TLS-RSA-WITH-AES-128-CCM-8	1.376094	1.488607	15.308802	106.81152	164
TLS-RSA-WITH-AES-256-CCM-8	1.460625	1.599616	18.453489	137.3291	152
TLS-DHE-RSA-WITH-AES-128-CCM-8	1.361354	1.489575	19.757969	122.07031	163
TLS-DHE-RSA-WITH-AES-256-CCM-8	1.457865	1.598959	19.818750	122.07031	152

Quadro 6: Latência, *jitter* e *Goodput* para operações de leitura de registradores (0x03)

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TCP	0.825989	0.908300	14.582292	76.29395	268

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TLS-RSA-WITH-NULL-MD5	0.925938	1.083499	14.488907	137.3291	225
TLS-RSA-WITH-NULL-SHA	0.987760	1.127380	19.140729	137.3291	216
TLS-RSA-WITH-NULL-SHA256	1.218646	1.381712	17.669635	91.55273	176
TLS-ECDHE-RSA-WITH-NULL-SHA	0.993594	1.131157	16.449114	167.84668	215
TLS-RSA-WITH-DES-CBC-SHA	1.686927	1.857248	18.216719	122.07031	131
TLS-RSA-WITH-3DES-EDE-CBC-SHA	2.911667	3.009299	22.187500	122.07031	81
TLS-DHE-RSA-WITH-DES-CBC-SHA	1.703542	1.860821	18.061823	152.58789	131
TLS-DHE-RSA-WITH-3DES-EDE-CBC-SHA	2.905209	3.013880	22.137240	91.55273	81
TLS-RSA-WITH-AES-128-CBC-SHA	1.272292	1.432373	19.352656	122.07031	170
TLS-DHE-RSA-WITH-AES-128-CBC-SHA	1.275573	1.425097	16.885729	137.3291	171
TLS-RSA-WITH-AES-256-CBC-SHA	1.320052	1.479978	17.658073	167.84668	164
TLS-DHE-RSA-WITH-AES-256-CBC-SHA	1.305782	1.478894	17.742708	183.10547	165
TLS-RSA-WITH-AES-128-CBC-SHA256	1.511198	1.662688	20.290937	91.55273	146
TLS-RSA-WITH-AES-256-CBC-SHA256	1.557709	1.702038	20.633125	137.3291	143
TLS-RSA-WITH-CAMELLIA-128-CBC-SHA	1.585364	1.736536	20.375885	122.07031	140
TLS-DHE-RSA-WITH-CAMELLIA-128-CBC-SHA	1.579740	1.735389	20.209427	122.07031	140
TLS-DHE-RSA-WITH-AES-128-CBC-SHA256	1.525104	1.662233	17.750156	137.3291	146
TLS-DHE-RSA-WITH-AES-256-CBC-SHA256	1.569740	1.702280	20.671771	457.76367	143
TLS-RSA-WITH-CAMELLIA-256-CBC-SHA	1.736406	1.881220	15.491667	91.55273	129

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TLS-DHE-RSA-WITH-CAMELLIA-256-CBC-SHA	1.719688	1.882825	18.097657	137.3291	129
TLS-RSA-WITH-AES-128-GCM-SHA256	1.381719	1.512552	19.821875	61.03516	161
TLS-RSA-WITH-AES-256-GCM-SHA384	1.416614	1.570583	17.868906	152.58789	155
TLS-DHE-RSA-WITH-AES-128-GCM-SHA256	1.386510	1.512523	15.803072	91.55273	161
TLS-DHE-RSA-WITH-AES-256-GCM-SHA384	1.437708	1.571127	14.939166	137.3291	155
TLS-RSA-WITH-CAMELLIA-128-CBC-SHA256	1.820417	1.965541	19.871459	76.29395	124
TLS-DHE-RSA-WITH-CAMELLIA-128-CBC-SHA256	1.819063	1.970141	19.636042	1358.03223	123
TLS-RSA-WITH-CAMELLIA-256-CBC-SHA256	1.957865	2.113737	19.581875	106.81152	115
TLS-DHE-RSA-WITH-CAMELLIA-256-CBC-SHA256	1.958073	2.115998	19.627239	152.58789	115
TLS-ECDHE-RSA-WITH-3DES-EDE-CBC-SHA	2.915677	3.014670	22.373021	152.58789	80
TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA	1.263854	1.432215	20.250260	152.58789	170
TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA	1.328594	1.478714	20.278594	152.58789	165
TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA256	1.494323	1.663661	17.759948	152.58789	146
TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA384	2.466407	2.627804	16.178646	137.3291	92
TLS-ECDHE-RSA-WITH-AES-128-GCM-SHA256	1.377865	1.512412	16.867396	91.55273	161
TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384	1.426875	1.570855	17.741041	122.07031	155
TLS-ECDHE-RSA-WITH-CAMELLIA-128-CBC-SHA256	1.807864	1.963484	20.885312	91.55273	124
TLS-ECDHE-RSA-WITH-CAMELLIA-256-CBC-SHA384	2.960833	3.030411	16.437240	122.07031	80
TLS-ECDHE-RSA-WITH-CAMELLIA-128-GCM-SHA256	1.712969	1.859637	16.526042	122.07031	131

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TLS-ECDHE-RSA-WITH-CAMELLIA-256-GCM-SHA384	1.865834	2.020557	15.898386	61.03516	120
TLS-RSA-WITH-AES-128-CCM	1.261198	1.410783	15.231042	167.84668	173
TLS-RSA-WITH-AES-256-CCM	1.375625	1.509703	19.210678	137.3291	161
TLS-DHE-RSA-WITH-AES-128-CCM	1.253594	1.410325	15.205417	137.3291	173
TLS-DHE-RSA-WITH-AES-256-CCM	1.358854	1.509709	15.565468	122.07031	161
TLS-RSA-WITH-AES-128-CCM-8	1.244375	1.403446	16.474740	152.58789	173
TLS-RSA-WITH-AES-256-CCM-8	1.372396	1.508312	15.142292	137.3291	161
TLS-DHE-RSA-WITH-AES-128-CCM-8	1.256458	1.404130	14.765833	152.58789	173
TLS-DHE-RSA-WITH-AES-256-CCM-8	1.375886	1.505061	18.398282	137.3291	162

Quadro 7: Latência, *jitter* e *Goodput* para operações de escrita e leitura de registradores (0x17)

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TCP	0.931146	1.071460	14.496979	198.36426	220
TLS-RSA-WITH-NULL-MD5	1.069687	1.255274	18.611666	106.81152	188
TLS-RSA-WITH-NULL-SHA	1.120209	1.308745	14.605469	244.14063	180
TLS-RSA-WITH-NULL-SHA256	1.475990	1.639954	15.227552	122.07031	144
TLS-ECDHE-RSA-WITH-NULL-SHA	1.142083	1.308300	14.561041	198.36426	180

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TLS-RSA-WITH-DES-CBC-SHA	2.459270	2.594665	15.898750	30.51758	91
TLS-RSA-WITH-3DES-EDE-CBC-SHA	4.606302	4.718325	24.218906	152.58789	50
TLS-DHE-RSA-WITH-DES-CBC-SHA	2.473073	2.595533	16.630052	91.55273	91
TLS-DHE-RSA-WITH-3DES-EDE-CBC-SHA	4.537761	4.718335	24.122865	106.81152	50
TLS-RSA-WITH-AES-128-CBC-SHA	1.567187	1.739909	15.062761	198.36426	135
TLS-DHE-RSA-WITH-AES-128-CBC-SHA	1.578646	1.740288	15.045885	213.62305	135
TLS-RSA-WITH-AES-256-CBC-SHA	1.639062	1.817283	21.347031	213.62305	130
TLS-DHE-RSA-WITH-AES-256-CBC-SHA	1.639792	1.818305	20.787291	152.58789	129
TLS-RSA-WITH-AES-128-CBC-SHA256	1.865729	2.042518	15.699792	183.10547	115
TLS-RSA-WITH-AES-256-CBC-SHA256	1.964427	2.132060	15.443177	228.88184	110
TLS-RSA-WITH-CAMELLIA-128-CBC-SHA	2.142605	2.318372	16.081354	183.10547	101
TLS-DHE-RSA-WITH-CAMELLIA-128-CBC-SHA	2.149167	2.317946	19.109687	152.58789	101
TLS-DHE-RSA-WITH-AES-128-CBC-SHA256	1.870625	2.044705	16.036875	213.62305	115
TLS-DHE-RSA-WITH-AES-256-CBC-SHA256	1.965104	2.135502	18.451355	244.14063	110
TLS-RSA-WITH-CAMELLIA-256-CBC-SHA	2.429063	2.584332	16.347500	228.88184	91
TLS-DHE-RSA-WITH-CAMELLIA-256-CBC-SHA	2.460105	2.585171	15.988333	198.36426	91
TLS-RSA-WITH-AES-128-GCM-SHA256	1.898855	2.070634	15.273281	61.03516	114
TLS-RSA-WITH-AES-256-GCM-SHA384	1.982395	2.162933	15.793698	91.55273	109
TLS-DHE-RSA-WITH-AES-128-GCM-SHA256	1.902917	2.073853	20.251458	61.03516	113

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TLS-DHE-RSA-WITH-AES-256-GCM-SHA384	2.003333	2.162383	18.340000	45.77637	109
TLS-RSA-WITH-CAMELLIA-128-CBC-SHA256	2.475312	2.630726	19.892917	244.14063	89
TLS-DHE-RSA-WITH-CAMELLIA-128-CBC-SHA256	2.501250	2.630828	18.675833	244.14063	89
TLS-RSA-WITH-CAMELLIA-256-CBC-SHA256	2.738073	2.895111	16.249479	198.36426	81
TLS-DHE-RSA-WITH-CAMELLIA-256-CBC-SHA256	2.736458	2.896201	16.513281	213.62305	81
TLS-ECDHE-RSA-WITH-3DES-EDE-CBC-SHA	4.560677	4.713835	24.194844	122.07031	50
TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA	1.552396	1.740524	16.073542	183.10547	135
TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA	1.648958	1.819382	20.896146	213.62305	129
TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA256	1.863021	2.043439	15.342084	198.36426	115
TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA384	3.020417	3.182694	16.430468	106.81152	74
TLS-ECDHE-RSA-WITH-AES-128-GCM-SHA256	1.900782	2.073526	15.300208	45.77637	113
TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384	1.996250	2.163875	15.481406	76.29395	109
TLS-ECDHE-RSA-WITH-CAMELLIA-128-CBC-SHA256	2.497865	2.630371	22.249375	198.36426	89
TLS-ECDHE-RSA-WITH-CAMELLIA-256-CBC-SHA384	3.837969	3.935777	23.131407	45.77637	60
TLS-ECDHE-RSA-WITH-CAMELLIA-128-GCM-SHA256	2.562240	2.689873	16.257240	91.55273	87
TLS-ECDHE-RSA-WITH-CAMELLIA-256-GCM-SHA384	2.825886	2.969010	16.735105	61.03516	79
TLS-RSA-WITH-AES-128-CCM	1.710521	1.885349	20.814739	76.29395	125
TLS-RSA-WITH-AES-256-CCM	1.902708	2.070918	18.790625	76.29395	114
TLS-DHE-RSA-WITH-AES-128-CCM	1.711302	1.884345	20.846771	61.03516	125

Suíte	Latência (ms)			*Jitter* (ns)	*Goodput*
	Minima	Média	Máxima		
TLS-DHE-RSA-WITH-AES-256-CCM	1.898177	2.070340	16.048438	61.03516	114
TLS-RSA-WITH-AES-128-CCM-8	1.714427	1.884079	16.570990	45.77637	125
TLS-RSA-WITH-AES-256-CCM-8	1.891875	2.066999	15.374688	61.03516	114
TLS-DHE-RSA-WITH-AES-128-CCM-8	1.714844	1.883232	19.966458	91.55273	125
TLS-DHE-RSA-WITH-AES-256-CCM-8	1.893178	2.067055	15.699479	30.51758	114

Quadro 8: *Overhead* para operações de leitura de bobinas (0x01), leitura de registradores (0x17) e escrita e leitura de registradores (0x17) nas suítes criptográficas testadas

Suíte	Leitura de Bobinas		Leitura de Registradores		Escrita e Leitura de Registradores	
	Req.	Res.	Req.	Res.	Req.	Res.
TCP	66	313	66	313	313	305
TLS-RSA-WITH-NULL-MD5	87	334	87	334	334	326
TLS-RSA-WITH-NULL-SHA	91	338	91	338	338	330
TLS-RSA-WITH-NULL-SHA256	103	350	103	350	350	342
TLS-ECDHE-RSA-WITH-NULL-SHA	91	338	91	338	338	330
TLS-RSA-WITH-DES-CBC-SHA	103	351	103	351	351	343
TLS-RSA-WITH-3DES-EDE-CBC-SHA	103	351	103	351	351	343

Suíte	Leitura de Bobinas		Leitura de Registradores		Escrita e Leitura de Registradores	
	Req.	Res.	Req.	Res.	Req.	Res.
TLS-DHE-RSA-WITH-DES-CBC-SHA	103	351	103	351	351	343
TLS-DHE-RSA-WITH-3DES-EDE-CBC-SHA	103	351	103	351	351	343
TLS-RSA-WITH-AES-128-CBC-SHA	111	367	111	367	367	351
TLS-DHE-RSA-WITH-AES-128-CBC-SHA	111	367	111	367	367	351
TLS-RSA-WITH-AES-256-CBC-SHA	111	367	111	367	367	351
TLS-DHE-RSA-WITH-AES-256-CBC-SHA	111	367	111	367	367	351
TLS-RSA-WITH-AES-128-CBC-SHA256	123	379	123	379	379	363
TLS-RSA-WITH-AES-256-CBC-SHA256	123	379	123	379	379	363
TLS-RSA-WITH-CAMELLIA-128-CBC-SHA	111	367	111	367	367	351
TLS-DHE-RSA-WITH-CAMELLIA-128-CBC-SHA	111	367	111	367	367	351
TLS-DHE-RSA-WITH-AES-128-CBC-SHA256	123	379	123	379	379	363
TLS-DHE-RSA-WITH-AES-256-CBC-SHA256	123	379	123	379	379	363
TLS-RSA-WITH-CAMELLIA-256-CBC-SHA	111	367	111	367	367	351
TLS-DHE-RSA-WITH-CAMELLIA-256-CBC-SHA	111	367	111	367	367	351
TLS-RSA-WITH-AES-128-GCM-SHA256	95	342	95	342	342	334
TLS-RSA-WITH-AES-256-GCM-SHA384	95	342	95	342	342	334
TLS-DHE-RSA-WITH-AES-128-GCM-SHA256	95	342	95	342	342	334
TLS-DHE-RSA-WITH-AES-256-GCM-SHA384	95	342	95	342	342	334

Suíte	Leitura de Bobinas		Leitura de Registradores		Escrita e Leitura de Registradores	
	Req.	Res.	Req.	Res.	Req.	Res.
TLS-RSA-WITH-CAMELLIA-128-CBC-SHA256	123	379	123	379	379	363
TLS-DHE-RSA-WITH-CAMELLIA-128-CBC-SHA256	123	379	123	379	379	363
TLS-RSA-WITH-CAMELLIA-256-CBC-SHA256	123	379	123	379	379	363
TLS-DHE-RSA-WITH-CAMELLIA-256-CBC-SHA256	123	379	123	379	379	363
TLS-ECDHE-RSA-WITH-3DES-EDE-CBC-SHA	103	351	103	351	351	343
TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA	111	367	111	367	367	351
TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA	111	367	111	367	367	351
TLS-ECDHE-RSA-WITH-AES-128-CBC-SHA256	123	379	123	379	379	363
TLS-ECDHE-RSA-WITH-AES-256-CBC-SHA384	139	395	139	395	395	379
TLS-ECDHE-RSA-WITH-AES-128-GCM-SHA256	95	342	95	342	342	334
TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384	95	342	95	342	342	334
TLS-ECDHE-RSA-WITH-CAMELLIA-128-CBC-SHA256	123	379	123	379	379	363
TLS-ECDHE-RSA-WITH-CAMELLIA-256-CBC-SHA384	139	395	139	395	395	379
TLS-ECDHE-RSA-WITH-CAMELLIA-128-GCM-SHA256	95	342	95	342	342	334
TLS-ECDHE-RSA-WITH-CAMELLIA-256-GCM-SHA384	95	342	95	342	342	334
TLS-RSA-WITH-AES-128-CCM	95	342	95	342	342	334
TLS-RSA-WITH-AES-256-CCM	95	342	95	342	342	334
TLS-DHE-RSA-WITH-AES-128-CCM	95	342	95	342	342	334

Suíte	Leitura de Bobinas		Leitura de Registradores		Escrita e Leitura de Registradores	
	Req.	Res.	Req.	Res.	Req.	Res.
TLS-DHE-RSA-WITH-AES-256-CCM	95	342	94	342	342	334
TLS-RSA-WITH-AES-128-CCM-8	87	334	87	334	334	326
TLS-RSA-WITH-AES-256-CCM-8	87	334	87	334	334	326
TLS-DHE-RSA-WITH-AES-128-CCM-8	87	334	87	334	334	326
TLS-DHE-RSA-WITH-AES-256-CCM-8	87	334	87	334	334	326