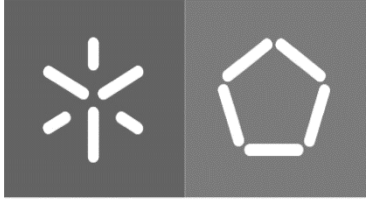
Universidade do Minho
Escola de Engenharia

Eduardo Gil Ferraz Soares Mendes

Extending the Security Spectrum of ARM TrustZone Technology

Maio de 2017



Universidade do Minho
Escola de Engenharia

Eduardo Gil Ferraz Soares Mendes

Extending the Security Spectrum of ARM TrustZone Technology

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Adriano Tavares
Professor Doutor Sandro Pinto

Maio de 2017

Declaração do Autor

Nome: Eduardo Gil Ferraz Soares Mendes
Correio Eletrónico: a65318@alunos.uminho.pt
Cartão de Cidadão: 14359668

Titulo da dissertação: Extending the Security Spectrum of ARM TrustZone technology
Ano de conclusão: 2017
Orientador: Professor Doutor Adriano Tavares

Designação do Mestrado: Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Eletrónica Industrial e Computadores
Área de Especialização: Sistemas Embebidos
Escola de Engenharia
Departamento de Eletrónica Industrial

Universidade do Minho, _____ / _____ / _____

Assinatura: _____

Agradecimentos

Todos nós somos um pedaço de cada pessoa que nos rodeia. Quero agradecer o facto de pertencer à família ESRG, em particular ao meu orientador Professor Doutor Adriano Tavares e dirigir-lhe uma especial palavra de apreço por tudo que me proporcionou a nível académico, mas principalmente a nível pessoal. Agradecer carinhosamente ao meu outro orientador, Professor Doutor Sandro Pinto, que para além de um grande mentor é um amigo. Obrigado por todo acompanhamento e por nestes quase dois anos seres o expoente da engenharia na minha vida: um grande exemplo a seguir. Mostrar também apreço ao Jorge Pereira, por ser um grande investigador e amigo: obrigado por me ajudares quando mais precisei.

Agradecer incondicionalmente aos meus putos: André Oliveira e João Silva por ter tido a honra de ter realizado este percurso académico a vosso lado, por sermos uma equipa, por sermos grandes amigos. Não poderia ter ficado mais grato por ter tido a oportunidade de fazer ERASMUS com vocês. Ficaremos para sempre com aquelas maravilhosas recordações. Numa forma geral, quero agradecer também aos GRANDES malucos (no qual eu também faço parte) e craques do ESRG: Mr. Gomes, Mr. Salgado, Monte, Carlos, Raphael, e aos restantes. Obrigado a todos!

A minha família não necessita de agradecimentos, pois tudo o que faço todos os dias é principalmente para eles e por eles. Amo-vos incondicionalmente: Pai, Mãe, irmã e irmão. Sem vocês nada sou e a vosso lado, a pessoa mais feliz deste mundo. À Daniela Miranda obrigado por todos os momentos mágicos e inesquecíveis que temos partilhado, por seres uma guerreira e despertares esse sentimento também em mim, por seres a minha âncora e me concederes clarividência, obrigado por me fazeres crescer constantemente.

Por fim, obrigado a todos os meus amigos incríveis, pois sem vocês realizar esta dissertação não seria possível.

Resumo

Os sistemas embebidos têm sido os grandes impulsionadores para o desenvolvimento tecnológico da sociedade e a sua presença está enraizada e proliferada em diversos setores da sociedade, como nos transportes, na medicina ou nos bens de uso comum como é o caso dos *smartphones*. A metamorfose e aumento de complexidade são consequências desta proliferação e, atualmente, a sua direção evolutiva possui como principal foco a conexão de todos os dispositivos à Internet. Mais do que nunca a exposição de dados valiosos e pessoais é elevada e suscetível a vários tipos de ataques, apetecivelmente atrativos a diversos géneros de *hackers*.

A segurança num sistema embebido é, hoje em dia, um fator diferenciador de peso. Até então, era implementada através da extensão de módulos em *software*, tipicamente nas etapas mais tardias do estágio de desenvolvimento do sistema. No entanto, dada a importância do requisito de segurança nos sistemas atuais, passou a ser crucial e necessário a sua implementação desde as primeiras etapas de desenvolvimento, ou seja, desde o *hardware*, de forma a garantir um *root-of-trust*.

O *hardware* é a solução mais fiável para a segurança nos sistemas embebidos se for capaz de garantir um ambiente seguro, vigilante e resiliente face a ataques. A tecnologia de segurança ARM TrustZone está integrada em milhões de processadores utilizados em *smartphones* e tem sido alvo de estudo por parte de analistas de segurança, que provaram algumas das suas vulnerabilidades, nomeadamente no seu canal de comunicação. A presente dissertação propõe a criação de uma solução de segurança capaz de introduzir uma camada de monitorização em *hardware* por forma a estender o espectro de segurança concedido pela tecnologia TrustZone. Os resultados obtidos certificam a metodologia empregue na elaboração da solução, que se baseou no estudo de casos reais e se foca numa monitorização do canal de comunicação da tecnologia TrustZone.

Palavras Chave: Sistemas Embebidos, Segurança, Monitorização, *Hardware*, ARM TrustZone;

Abstract

Embedded systems have been the main cause for the society's technological evolution and they are deeply present in many sectors of society, like in transportation systems, medical equipments or consumer goods, like smartphones. Embedded system's metamorphosys and increase in its complexity are consequences of this proliferation and nowadays the evolution is towards the connection of all devices through the Internet. More than never, valuable and personal data are exposed and vulnerable to various kinds of attacks, very attractive to various kinds of hackers.

The security requirement is seen today as a major factor. Until now, security was implemented through the extension of software modules, typically in the latter stages of the system's development. However, due to the importance of the security in the modern systems, is crucial and necessary its implementation from the earlier stages of the device's development, as well as, directly on the hardware in order to guarantee a complete root-of-trust.

The most effective way to achieve security is through hardware, but only if it is capable of providing a secure, vigilant and resilient environment on the presence of attacks. ARM TrustZone technology is present in millions of smartphone's processors and has been target of studies performed by several security analysts that had proven some of its vulnerabilities, in particular on its communication channel. This master thesis proposes the creation of a security solution that introduces a monitor layer, in hardware, capable of extending the security spectrum of TrustZone technology. The obtained results validate the metodology that was adopted to develop the security solution. It was based on the study of real world cases while focusing on the monitoring of the communication channel.

Key words: Embedded Systems, Security, Monitoring, Hardware, ARM TrustZone;

Índice

Índice **xiii**

Lista de Acrónimos **xvii**

Lista de Figuras **xxi**

Lista de Tabelas **xxiii**

Lista de Listas **xxv**

1 Introdução **1**
1.1 Contextualização 1
1.2 Motivação e Objetivos 2
1.3 Estrutura da Dissertação 3

2 Literatura e Estado da Arte **5**
2.1 Sistemas Embebidos 5
2.2 Segurança nos Sistemas Embebidos 6
2.2.1 Considerações e Conceitos básicos 8
2.2.2 Características e Vulnerabilidades 10
2.2.3 Ataques e Defesas 11
2.2.3.1 *Software* 12
2.2.3.2 *Hardware* 13
2.2.4 Virtualização 16
2.2.5 *Hardware* para a Segurança 17
2.2.5.1 *Trusted Platform Module* 17
2.2.6 *Trusted Execution Environment* 18
2.2.7 ARM TrustZone 18
2.2.7.1 Soluções baseadas em TrusZone 20

3 Especificação do Sistema **25**
3.1 Arquitetura *Advanced RISC Machine* (ARM) 25
3.1.1 Conceitos básicos 25

3.1.1.1 Extensões do Processador . . . 28
3.1.2 ARM TrustZone . . . 28
3.1.2.1 Arquitetura do *Hardware* . . . 28
3.1.2.2 Arquitetura do *Software* . . . 33
3.2 AXI . . . 35
3.2.1 AXI-*Lite* . . . 36
3.3 GlobalPlatform API . . . 37
3.3.1 TEE *Client* API . . . 37
3.4 Plataforma de Desenvolvimento . . . 38
3.4.1 Zynq-7000 *All Programmable* SoC . . . 38
3.5 *Toolchain* de Desenvolvimento . . . 39
3.5.1 Vivado Design Suite . . . 40
3.5.2 Xilinx SDK . . . 40
3.5.3 ARM Fast Models . . . 41

4 Implementação do L-TEE **43**
4.1 Visão Geral do L-TEE . . . 43
4.1.1 Mundo Seguro . . . 44
4.1.1.1 T-RTOS . . . 44
4.1.1.2 Monitor . . . 45
4.1.1.3 COMON . . . 45
4.1.2 GPOS . . . 45
4.1.3 Comunicação entre Mundos . . . 46
4.2 Gestão temporal e de interrupções . . . 46
4.3 Proteção da memória . . . 48
4.4 Proteção dos dispositivos . . . 48
4.5 Comutação entre ambientes . . . 49
4.6 Proteção da infraestrutura de *debug* . . . 49
4.7 Comunicação . . . 50
4.7.1 TEE . . . 50
4.7.2 REE . . . 51
4.7.2.1 GlobalPlatform TEE *Client* API . . . 51
4.7.2.2 *Device driver* TrustZone . . . 53

5 Metodologia e Implementação do COMON **55**
5.1 Espectro de Segurança . . . 55
5.2 Monitorização do canal de comunicação . . . 58
5.3 Deteção não intrusiva da SMC . . . 60

5.4 Investigação 60
5.4.1 Barramento AXI e visibilidade à PL 61
5.4.2 Componentes CoreSight na Zynq 62
5.5 *Communication Monitor* - COMON 65
5.5.1 Proteção do Canal 66

6 Resultados **73**
6.1 Validação da Comunicação 73
6.2 Ataque ao LTEE 76
6.2.1 Modelo e Considerações 77
6.2.2 *Ataque Code Execution* 78
6.2.2.1 Análise da *stack* da comunicação 78
6.2.2.2 Execução do ataque 85
6.2.3 Proteção do COMON 88
6.3 Análise de Segurança 89

7 Conclusões e Perspetivas **91**
7.1 Conclusões 91
7.2 Limitações e Trabalho Futuro 92

Lista de Acrónimos

AMBA *Advanced Microcontroller Bus Architecture*

APB *Advanced Peripheral Bus*

API *Application Programming Interface*

ARM *Advanced RISC Machine*

AXI *Advanced eXtendible Interface*

CA aplicação cliente

CAIC *Control Availability Integrity Confidentiality*

CIA *Confidentiality Integrity Availability*

CISC *Complex Instruction Set Computing*

COMON *Communication Monitor*

CP coprocessadores

CP15 *System Control Coprocessor*

CPU *central processing unit*

DoS negação de serviço

FIQ *fast interrupt request*

FPGA *Field-Programmable Gate Array*

GPOS sistema operativo de propósito geral

GIC *Generic Interrupt Controller*

ICDs *In-Circuit-Debugers*

IT *Information Technology*

IoT Internet das Coisas

IPs *Intlectual Properties*

IRQ *interrupt request*

ISA *Instruction Set Architecture*

JTAG *Joint Test Action Group*

L-TEE *Lightweight Trusted Execution Environment*

LTZVisor *Lightweight* TrustZone-*assisted Hypervisor*

MMU *Memory Management Unit*

OEMs *Original Equipment Manufacturers*

OT *Operation Technology*

PCs computadores pessoais

PL *Programmable Logic*

PS *Processing System*

REE *Rich Execution Environment*

RISC *Reduced Instruction Set Computing*

RTOS sistema operativo de tempo-real

SO Sistema Operativo

SCR *Secure Configuration Register*

SMC *Secure Monitor Call*

SoC *System on Chip*

TA serviço seguro

TCB *Trust Computing Base*

TCG *Trust Computing Group*

TEE *Trusted Execution Environment*

TPM *trusted platform module*

TZAPI TrustZone *Application Programming Interface*

TZASC TrustZone *Address Space Controller*

TZMA TrustZone *Memory Adapter*

VMs máquinas virtuais

Lista de Figuras

2.1 Segurança através de *Software* 8
2.2 Abordagem da segurança por *Hardware* em camadas 9
2.3 Arquitetura geral do FreeTEE 21
2.4 Arquitetura geral do SeCReT 22
2.5 Arquitetura geral do LTZVisor 23
3.1 Mundo Seguro e Não-Seguro 29
3.2 Zynq-7000 *All Programmable SoC* 39
4.1 L-TEE: Arquitetura genérica 44
4.2 *Path* de comunicação 47
4.3 Configuração da memória no LTEE 48
5.1 Arquitetura *Harvard* na Zynq 62
5.2 Arquitetura CoreSight na Zynq 63
5.3 Deteção de uma intrução através do PTM 64
5.4 Contagem do número de eventos através da PMU 65
5.5 Acoplamento do COMON à Zynq 66
5.6 Sub-componentes do COMON 67
5.7 Máquina de estados do comportamento do *driver* 68
5.8 Máquina de estados representativa da monotorização 69
6.1 Abertura de sessão da comunicação 74
6.2 Realização da funcionalidade eco 75
6.3 Visão geral do ataque *code execution* 85

Lista de Tabelas

2.1 Características vs Vulnerabilidades 12
2.2 Ataques vs Contramedidas 15
3.1 Modos do processador da arquitetura ARMv7 26
3.2 Conjunto total de registos da arquitetura ARMv7 27
3.3 *Vector table* 28
3.4 *Bits* de proteção do barramento AXI 32
3.5 Sinais por cada canal AXI-*Lite* 36
3.6 Sinais AXI-*Lite* 36
4.1 Funções TEECAPI 52
5.1 Espectro de segurança da tecnologia TrustZone 58

Lista de Listas

4.1 Suporte à comunicação no Monitor 50
4.2 Criação de um serviço seguro 51
5.1 Configuração da interrupção do COMON 70
5.2 Rotina de atendimento à interrupção do COMON 70
6.1 Estrutura principal da comunicação 77
6.2 Superfície de ataque no Monitor 79
6.3 Variáveis relevantes da rotina tz_api_call 79
6.4 Variáveis que permitem a execução da get_api_context() 80
6.5 Rotina detentora de *write primitives* 81
6.6 Implementação do serviço seguro ECHO_SS 82
6.7 Invocação da função vulnerável 83
6.8 Função vulnerável 84

Introdução

Neste capítulo é contextualizado o tema da presente dissertação, que se enquadra na segurança dos sistemas embebidos. Posteriormente o autor demonstra quais as razões motivacionais ao tema e os objetivos a que se propõe para o desenvolvimento da presente dissertação.

1.1 Contextualização

Nos Estados Unidos, algures nos anos 50, durante a transição da era industrial para a presente era da informação, o visionário americano, John Naisbitt [1], disse: "*The source of power is not money in the hands of few, but information in the hands of many*". A Internet desempenhou um papel preponderante para o mundo da forma que o conhecemos e, nos dias que correm, ela fortalece economias, une laços entre nações e até melhora a vida de cada um de nós. Paralelamente, os sistemas embebidos também sofreram, desde a sua utilização em rudimentares máquinas de calcular, até aos poderosos processadores atuais, uma enorme metamorfose. Desde a criação do primeiro processador, o 4004 de 4-*bits* da Intel em 1971, os sistemas embebidos explodiram comercialmente, proliferando nos diversos setores da sociedade. À medida que a sociedade caminha para a era da Internet das Coisas (IoT), os sistemas embebidos evoluem tendo em vista a sua conexão com milhares de dispositivos através da Internet.

Os sistemas embebidos, como por exemplo os *smartphones*, proporcionam aos utilizadores diversas experiências concedidas por um vasto conjunto de funcionalidades: *email*, banco e comércio *online* ou entretenimento como a música e vídeo, contudo o total aproveitamento das capacidades destes dispositivos está condicionada, pois as respostas às exigências dos consumidores dependem do risco que os donos de conteúdo comercial estão dispostos a aceitar. Assim sendo, a máxima extração das potencialidades destes dispositivos não depende somente da sua alta performance e eficiência energética, mas também, do nível de segurança que estes são capazes de assegurar.

A segurança é cada vez mais, considerada um fator de peso no momento de compra dos consumidores tradicionais, indústrias e governos. O cidadão comum

reivindica a proteção e a privacidade dos seus dados pessoais, as indústrias exigem apertados requisitos de segurança em torno dos seus produtos ou serviços e, por sua vez, alguns governos têm reagido enfurecidamente a ameaças reais e poderosas, que comprometem algumas das suas infraestruturas [2].

Um ataque ao sistema embebido pode ser executado através de *software* [3], da monitorização do *hardware* [4] (*side-channel*), da adulteração do *hardware* (Trojan), entre outros. A segurança deve assegurar que recursos valiosos contidos no sistema não são copiados, danificados ou tornados indisponíveis [5]. Para poder conceder tal nível de proteção perante ameaças recentes e cada vez mais complexas, a mentalidade dos arquitetos destes sistemas passou a considerar a segurança como sendo um fator crítico e a ter em conta durante todo o processo de desenvolvimento. Até então, esta era implementada apenas com a extensão de módulos em *software* em etapas mais tardias do estágio de desenvolvimento. Porém, dada a importância do requisito de segurança nos sistemas atuais, passou a ser crucial e necessário a sua implementação desde as primeiras etapas, isto é, durante o desenvolvimento do *hardware*.

O mercado tem correspondido à crescente exigência na segurança, como por exemplo a tecnologia TrustZone é uma incisiva e forte resposta a estas exigências, por parte de um fabricante que é considerado um dos "*big players*" no desenho de processadores - ARM.

1.2 Motivação e Objetivos

A forte presença e proliferação dos sistemas embebidos em quase todos os domínios da sociedade provocou uma dependência enorme das sociedades para com estes sistemas. Para chegar a este patamar, os sistemas embebidos sofreram uma fantástica metamorfose e aumento de complexidade. Atualmente a direção evolutiva é no sentido da interconectividade segura e inteligente entre todos os dispositivos, sendo a Internet o principal meio de conexão e comunicação entre eles. No entanto, a Internet é uma fonte diversificada de ameaças, que desejam, entre outros, a obtenção de informações sensíveis (credenciais bancárias ou chaves criptográficas) e o controlo ou paragem do sistema.

A forma de incorporar a segurança nos sistemas embebidos foi repensada, no entanto, apesar dos esforços, não se previu que recentes tipos ataques, quer por *software* quer diretos ao *hardware*, violariam o *root-of-trust* [6].

É neste enquadramento que surge a presente dissertação. O trabalho que se pretende desenvolver é sobre a tecnologia de segurança ARM TrustZone. Não

obstante do alto grau de segurança que a tecnologia garante, tem sido alvo de estudo por parte de analistas que provaram algumas das suas vulnerabilidades. Assim sendo, a dissertação tem como principal objetivo a implementação de um mecanismo capaz de alargar o espetro de segurança concedido pela tecnologia TrustZone. Para esse feito os seguintes objetivos têm que ser alcançados:

- Estudo da tecnologia de modo a percepcionar teoricamente as suas vulnerabilidades.
- Definir o espetro de segurança da tecnologia TrustZone, ou seja, identificar quais tipos de ataques a tecnologia concede ou não proteção.
- Com base nas conclusões anteriores, desenvolver um periférico em *hardware* que implemente mecanismos de proteção capazes de estender o espectro de segurança.

1.3 Estrutura da Dissertação

A presente dissertação está enquadrada no âmbito da segurança e, propõe-se a criar uma solução de segurança específica para sistemas baseados na tecnologia TrustZone. A introdução e contextualização ao tema, como também, a enumeração do objetivos da dissertação é realizado no Capítulo 1. No Capítulo 2 são abordados conceitos essenciais sobre a segurança em sistemas embebidos, assim como, *frameworks* em *software* que tiram partido da tecnologia. No Capítulo 3, é apresentado a arquitetura alvo - ARM, detalhada a tecnologia central do trabalho - TrustZone, bem como apresentados conceitos, tecnologias e ferramentas necessárias ao desenvolvimento do trabalho que é proposto. Seguidamente, no Capítulo 4, é introduzido a *framework* desenvolvida - *Lightweight Trusted Execution Environment* (L-TEE). Esta serviu de base de desenvolvimento da solução de segurança em *hardware* - o *Communication Monitor* (COMON). Devido a isso, no Capítulo 5, o autor aborda a investigação realizada e que esteve na base das decisões de implementação do COMON. Os resultados extraídos são apresentados no Capítulo 6 e, por fim, no Capítulo 7, o autor realiza uma apreciação sobre as suas conclusões assim como o trabalho que faz sentido realizar no futuro.

Literatura e Estado da Arte

A presente dissertação está enquadrada no âmbito da segurança em sistemas embebidos, deste modo, neste capítulo são abordados conceitos essenciais sobre esta temática. Na secção 2.1 é descrito o conceito de sistemas embebidos aceite pela comunidade científica atual. Na secção 2.2 é contextualizado o tema da segurança nestes sistemas.

Relativamente à segurança, são introduzidos primeiramente conceitos básicos, que pretendem clarificar algumas ideias, assim como servir de suporte à leitura da restante dissertação. O tópico seguinte, relaciona algumas características inerentes aos sistemas embebidos com algumas das suas vulnerabilidades, que em certos casos funcionam como incentivo ao atacante. Posteriormente, é feita a caracterização dos tipos de ataques, bem como respetivas contramedidas, que existem atualmente. Por fim, são explicadas um conjunto de tecnologias tradicionais usadas para incorporar segurança nestes sistemas. O capítulo termina com a tecnologia alvo da dissertação e alguns exemplos da sua aplicabilidade.

2.1 Sistemas Embebidos

Descrever sucintamente o que são sistemas embebidos não é um tarefa fácil, quer pela ambiguidade existente ao nível da literatura, quer pela metamorfose e evolução a que têm sido sujeitos ao longo destes anos. Apesar disso, geralmente define-se um sistema embebido como sendo um sistema dedicado que, contrariamente a um sistema de propósito geral, executa apenas um leque reduzido de tarefas, sejam elas de maior ou menor grau de complexidade [7]. Tipícamente regem-se por requisitos temporais, ou seja, o pedido de uma tarefa deve ser atendido num período de tempo pré-definido.

Como Koopman classificou [8] "*embedded computing is more readily defined by what it is not (it is not generic application software executing on the main CPU of a "desktop computer") than what it is*". O termo "embebido", ficou popular pelo departamento da defesa Americano, que citava que estes estão embebidos num sistema maior, na qual, a sua principal função não é computacional [7]. Consequentemente, isso exclui os computadores pessoais (PCs) de serem considerados

sistemas embebidos. No entanto, tendo em conta as mais diversas áreas onde se encontram os sistemas embebidos e em resposta às necessidades da sociedade, estes têm-se estendido para áreas de maior diversidade e heterogeneidade, fazendo com que se aproximem cada vez mais, a nível de complexidade e funcionalidades, dos PCs. Exemplo disso são os telemóveis, cuja funcionalidade inicial era a de comunicar com outros dispositivos, e nos dias de hoje, com o aparecimento dos *smartphones*, podemos comunicar com outros dispositivos, gozar do vasto entretimento que nos transmite (i.e., jogos, fotografia e vídeo, *email*, *web browsing*) e até fazer uso do GPS. O paradigma concreto sobre definição de sistema embebido não se encontra estático e há uma tendência para que existam alterações com o evoluir da sociedade e suas necessidades

Quase não existem setores na sociedade moderna que pudessem sobreviver sem os sistemas embebidos. Eles existem em todas as indústrias, nos transportes em geral, bens de consumo, entretenimento, câmaras fotográficas e de vídeo. Existem também em vestuário, calçado, acessórios como óculos e também em implantes humanos! A sua presença tem-se tornado ubíqua, pois cada vez menos nos apercebemos presença dos mesmos.

Por vezes são desenvolvidos para obedecer a requisitos temporais. O comportamento de sistemas de tempo real tem de respeitar a respetiva funcionalidade mas também um conjunto de restrições temporais definidas externamente, por isso é comum classificar estes sistemas com base no tempo de reação e ambiente onde estão inseridos. As restrições temporais podem ser, *hard*: exige que os *deadlines* sejam sempre satisfeitos, caso contrário a integridade do sistema ou humana pode estar em risco; ou *soft*: caso os *deadlines* não sejam cumpridos apenas a performance geral do sistema é afetada.

2.2 Segurança nos Sistemas Embebidos

Nos dias que correm, a segurança dos sistemas embebidos requer, cada vez mais, uma atenção especial, dada a sua elevada importância. Em ambientes industriais, estes sistemas estão associados a infraestruturas chave da sociedade (i.e., produção de energia, fornecimento de água, serviços de transporte). Assim sendo, a ausência ou falhas dos mesmo pode culminar em consequências desastrosas, que podem inclusive comprometer vidas humanas. Por sua vez, os sistemas embebidos, devido à sua evolução, possuem hoje em dia, um papel ativo num mundo cada vez mais conectado, onde antigamente apenas os PCs pertenciam. A todo momento,

estes tipos de sistemas, lidam com dados valiosos (i.e., credenciais bancárias, conteúdo comercial), tornando-os também alvos atrativos. Quando atacados com sucesso, a imagem das empresas associadas pode ficar denegrida e consequentemente, resultar em elevadas percas financeiras.

A forma como a segurança é idealizada nos dias de hoje, está a ser marcada à medida que a sociedade entra na era IoT. Um mundo onde todas as "coisas" estão interconectadas através da Internet, obrigará a que todos os negócios aceitem a Internet e que retirem o melhor partido desta (i.e., eficiência na produção, conectividade), com vista a acompanharem a exigências de um mercado atual, extremamente competitivo.

No passado, a segurança era abordada de forma separada. Os ambientes industriais não eram vítimas dos perigos da Internet. Estes possuíam redes internas baseadas em outras tecnologias que não a Internet, estando deste modo protegidas de qualquer ataque exterior. A superfície e vetores de ataque eram portanto muito mais contidos. No entanto, novos conceitos de negócio começavam a explorar as funcionalidades únicas da Internet. Estas empresas tiveram que se adaptar através, por exemplo, da criação de departamentos dedicados a gerir tudo que englobasse estas novas funcionalidades. Rapidamente estes departamentos começaram a deparar-se com problemas de segurança, pois a comunidade de *hackers* foi se tornando cada vez mais especializada (e perigosa), motivada pelos valores que as empresas movimentavam.

Com o advento e exigências da era IoT, tanto o ambiente industrial e suas tecnologias, também conhecido por *Operation Technology* (OT), como todos os serviços baseados na Internet, também conhecido por *Information Technology* (IT), estão conectados à Internet. De modo a adaptar-se, o ambiente OT começou a migrar funcionalidades para esta nova realidade. Contudo, este não pode ariscar-se a ser alvo de ataques que comprometam as suas infraestruturas [2]. Deste modo, a idealização da segurança é feita, hoje em dia, considerando as duas áreas distintas – OT e IT. Tendo como base as propriedades de segurança, existem dois modelos que guiam as decisões e políticas de segurança: *Confidentiality Integrity Availability* (CIA) para o ambiente IT e *Control Availability Integrity Confidentiality* (CAIC) para OT [9]. O modelo CIA é explicado detalhadamente na secção 2.2.1, no entanto, é possível observar que ambos se interceptam em vários aspetos, mas possuem uma ordem de prioridade distinta. O novo conceito introduzido para OT, é o Controlo (*Control*). Este representa a Fiabilidade que é imprescindível para garantir o bom funcionamento dos sistemas neste tipo de ambiente.

A disponibilidade surge em segundo lugar, pois caso um serviço não seja disponibilizado as pessoas podem sofrer consequências graves.

A segurança em sistemas embebidos era, no passado, considerada como sendo um requisito extra adicionada através de módulos em *software*, Figura 2.1. Hoje em dia, é algo a considerar desde a fase conceptual do sistema, ou seja, vem incorporada no próprio *hardware*.

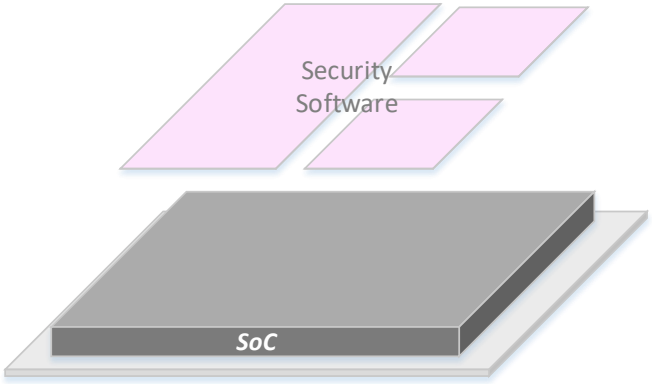


Figura 2.1: Segurança através de *Software*

O *hardware* é encarado como sendo a solução mais fiável para a segurança nos sistemas embebidos, se for capaz de conceder um ambiente seguro, vigilante e resiliente (Figura 2.2) face aos possíveis vectores e metodologias de ataques. Um ambiente seguro é visto como uma camada de "*trust*", proporcionadora de mecanismos (i.e., isolamento físico de memória, etc) com objetivo de garantir ao sistema integridade, autenticidade, entre outros requisitos. A vigilância visa detetar intrusões que transpassem a primeira camada, por forma a conceder ao sistema uma capacidade de reação. A camada de resiliência permite ao sistema a capacidade de voltar à sua operação estável.

2.2.1 Considerações e Conceitos básicos

A incorporação da segurança é uma tarefa árdua, tanto no desenvolvimento de *software* como no *hardware*, pois este é um requisito que tipicamente diverge de outros requisitos e métricas do sistema: a 1) funcionalidade de um sistema resume-se aos meios que este fornece para realizar um conjunto de tarefas. Em *software*, um produto é avaliado pelo número de funcionalidades que possui, traduzindo consequentemente num aumento de complexidade, o que vai contra aos princípios da segurança; a 2) usabilidade de um produto, ou seja, a facilidade que um utilizador possui em usufruir da funcionalidade também vai contra a segurança, tome-se como consideração, por exemplo, mecanismos de autenticação (i.e., *passwords*); o

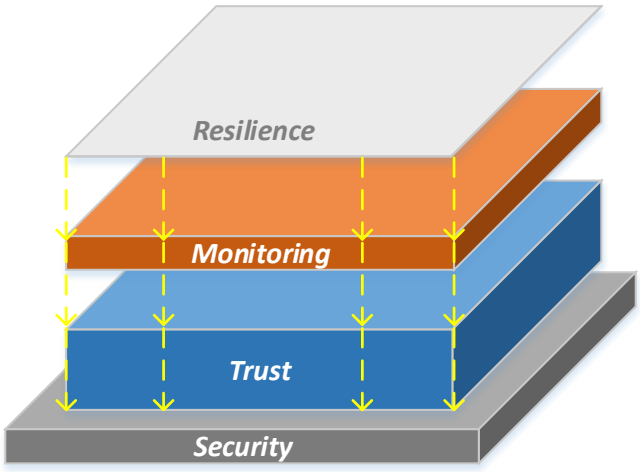


Figura 2.2: Abordagem da segurança por *Hardware* em camadas

3) desempenho continua a ser um entrave à segurança, nomeadamente devido às custosas operações em *software* de módulos criptográficos ou mesmo mecanismos de introspeção; Por fim, a necessidade de segurança reduz a 4) simplicidade de um produto, o que se traduz em maiores custos de produção e manutenção.

O *designer* deve ter em consideração que não é possível criar-se um sistema totalmente seguro, pois com tempo, determinação e recursos adequados, um atacante pode quebrar a segurança de qualquer sistema. Ao invés de tentar reduzir as vulnerabilidades a zero, deve-se ter em conta o fator de risco, isto é, a probabilidade que um sistema tem de ser comprometido. Um sistema muito vulnerável, contudo pouco distribuído, possui pouca exposição a ataques e por isso uma probabilidade baixa de ser atacado. Em contrapartida, um sistema valioso e disseminado (i.e., sistema bancário) é suscetível a ser atacado e por isso deve ser robusto. A probabilidade de um ataque com sucesso é medida através do nível de ameaça que o sistema está sujeito e do grau de vulnerabilidade que este possui. Considerando o exemplo do sistema bancário, este quando comprometido pode resultar em elevada perca monetária ou até dos próprios clientes. De maneira que, outro elemento que é considerado para o cálculo do fator de risco é o impacto que resulta da falta de segurança.

O tema da segurança é controverso pois carece de uma definição concreta. Um dos maiores dilemas são os conceitos de segurança e fiabilidade, pois apesar de parecerem distintos, não possuem atualmente um perímetro bem definido. No entanto, num ambiente IT, o objetivo da segurança é garantir três atributos: confidencialidade, integridade e disponibilidade (CIA). O primeiro, a confidencialidade (ou privacidade), consiste em não permitir que informação seja acedida por alguém não autorizado. É comum em *passwords* e chaves criptográficas. A

integridade garante a ausência de modificações nos dados, por parte de intervenientes não autorizados. É fundamental para mecanismos como *secure boot* de uma plataforma. Por último, a disponibilidade, está relacionada com a capacidade do sistema facultar atempadamente a informação. Sendo relevante face a ataques de negação de serviço (DoS), que tentam privar o sistema a um recurso crítico ou funcionalidade. Aos três atributos, pode ser adicionado a autenticidade, que está associada com a genuinidade da informação. É importante para que o sistema seja capaz de detetar a genuinidade da informação ou *software*, antes de o colocar em execução. Um conceito igualmente importante é o de vulnerabilidade. Esta é um defeito ou falha relevante para a segurança do sistema, passível de ser explorado por um atacante. Um ataque é o ato de se proceder à ativação de uma vulnerabilidade e quando este é bem sucedido está-se perante uma intrusão (*exploit*). À área num sistema computacional que pode ser alvo de ataques, dá-se o nome de superfície de ataque. Por fim, um termo mais lato é o vetor de ataque, pois abrange o tipo de vulnerabilidade, a forma de como o ataque é realizado ou o componente do sistema que é explorado.

2.2.2 Características e Vulnerabilidades

As características naturais dos sistemas embebidos podem por vezes ser aspetos que lhes propiciam ser vítimas de ataques que colocam a sua segurança em risco. A limitação a nível de recursos tem influência no menor poder de processamento comparativamente aos PCs, o que leva a que, por vezes, não executem *software* adequado de segurança (antivírus). A sua vertente *low-power*, significa que grande parte da sua energia é canalizada para o processamento das funções principais, o que em certos casos exclui as funcionalidades segurança. A localização onde são utilizados tem um papel preponderante para a sua segurança. Quando usados em localizações remotas e agrestes (ambientes industriais), atualizar o seu *firmware* é difícil. Por outro lado, os que se encontram em locais de fácil acesso, estão sujeitos a ataques físicos diretos ao *hardware*.

O aspeto fundamental dos sistemas embebidos desta geração é a conexão à Internet, esta é um oásis de ameaças e a maior fonte de ataques aos sistemas embebidos. Uma particularidade recentemente demonstrada por estudos internacionais é o papel negativo que algumas empresas têm tido no fabrico de sistemas embebidos [10, 6]. Tal facto deve-se à tendência económica mundial, que potencia a conceção por entidades não confiáveis, que comprometem a segurança do sistema logo nos primeiros estágios de desenvolvimento.

As características supracitadas podem ser encaradas, noutra ótica, como pré-condições para a realização de ataques, motivados pelos recursos de elevado valor contidos no sistema e conhecidas vulnerabilidades na sua segurança. O escoamento de energia, é um ataque que explora o fato de estes sistemas se alimentarem por baterias. Este pode ser efetuado sobrecarregando o sistema com vários sensores e periféricos que consequentemente diminuem a vida útil do sistema. Associado ainda às restrições a nível dos recursos e também de custo, está a carência de mecanismos de segurança eficazes. A criptografia é um desses mecanismos, muito comum, todavia em diversos casos a sua utilização é inapropriada. Existem ataques quer por *software* quer ao *hardware* criptográfico, que exploram falhas desses mecanismos, quer seja na sua própria implementação ou no protocolo que usam para comunicar com o exterior. A proximidade física para com o atacante permite a realização de ataques onde o contacto direto com o sistema é necessário, como em ataques *side-channel*, *microprobing*, engenharia inversa e ataques ao microcontrolador por meio de infraestruturas de depuração. A introdução de informação falsa através dos sensores ou a reescrita da memória física podem alterar o correto funcionamento do sistema.

O *software*, Sistema Operativo (SO), aplicação ou *middleware*, que executa sobre a plataforma é portador de inúmeros *bugs* de programação. Geralmente é aceite que num programa de *software* com 1000 linhas de código, existam entre 5 a 50 bugs, dependendo dos métodos de desenvolvimento usados [11]. Através da Internet ou do sistema de ficheiros, é possível a injeção de ataques de *software*, que exploram as vulnerabilidades e resultam na execução de código arbitrário ou paragem de todo o sistema.

Como foi referido, fabricantes não confiáveis, podem comprometer a segurança do sistema através de alterações malignas no *hardware*. Estes são defeitos intencionais que tornam o sistema vulnerável a ataques *trojan*, que ativam o *hardware* maligno com o objetivo de roubar informação sensível ou provocar a falha.

A Tabela 2.1 ilustra de forma simples e resumida, a correspondência entre as características mencionadas e suas vulnerabilidades.

2.2.3 Ataques e Defesas

No presente documento os ataques que afetam os sistemas embebidos foram divididos em duas categorias: *software* e *hardware*. Associado a cada tipo são também apresentados contramedidas tipicamente utilizadas.

Tabela 2.1: Características vs Vulnerabilidades

Vulnerabilidades	Características: *Limitação de Recursos*	*Localização do Sistema*	*Conexão à Internet*	*Fabrico Duvidoso*
Erros de Programação			x	
Mecanismos de Proteção	x	x		
Modificações no *Hardware*				x

2.2.3.1 *Software*

Os ataques em *software* são, de entre todos os tipos, os que possuem menos custos tanto para a sua execução como para o seu desenvolvimento e por isso são normalmente os mais comuns. Podem ser divididos em duas categorias: ataques técnicos e ataques de engenharia social. Os primeiros são fruto de vulnerabilidades nos componentes do sistema, os restantes, são dirigidos a vulnerabilidades existentes nos próprios utilizadores, a título de exemplo, a invasão de um dispositivo através de credenciais ingenuamente fornecidas por utilizador legitimo.

Um programa malicioso (*malware*) pode atingir a vítima de vários modos. Um vírus é um programa que se propaga anexado a *software* legitimo ou num meio de armazenamento portátil e que é executado quando a vitima lhe acede. Um cavalo de Troia, é *malware* dissimulado de programa legítimo e que por isso é executado pela vítima. Os referidos ataques baseiam-se na falta de credulidade da vítima, contudo, um *worm* propaga-se automaticamente sem intervenção das suas vítimas, geralmente através da Internet. Este tipo de *malware* ataca vulnerabilidades no *software* e propaga-se de dispositivo em dispositivo continuamente, selecionando os seus alvos com base na análise de vulnerabilidades [11] (para as quais foram projetados) que garantam a sua propagação. Um ataque que explore tipos de vulnerabilidades técnicas pode ter vários resultados. Um *buffer overflow*, talvez o *bug* mais famoso em segurança de *software*, resulta na escrita arbitrária da memória.

Em última instância pode conceder ao atacante a execução de *malware* (ataque de execução de código), ou apenas resultar na paragem do sistema (*crash*). O impacto do ataque está dependente maioritariamente do cuidado no desenvolvimento do *software*, no entanto, o desenvolvedor pode recorrer a ferramentas que automatizam o processo de deteção de erros.

2.2.3.2 *Hardware*

No presente documento dividiu-se os ataques através de *hardware* em três tipos, conforme a literatura atual: físicos, *side-channel* e *trojan*.

Ataques Físicos

Os ataques físicos necessitam de elevados recursos financeiros, temporais, assim como também, de um elevado grau de conhecimento por parte do atacante. São ataques invasivos que utilizam ferramentas como: osciloscópio (tradicional ou de varrimento eletrónico) ou microscópios de alta resolução. Algumas espécies são o *microprobing* e engenharia inversa. O primeiro recorre a máquinas de corte a *laser* para entrar em contacto direto com a superfície do *chip* e tem como objetivos, por exemplo, observar o conteúdo armazenado em memória ou manipular o valor em registos e assim interferir com o comportamento do componente. A engenharia inversa baseia-se em compreender a estrutura interna do *hardware* e emular a sua funcionalidade. Nos *chips* modernos, compostos por milhares de portas lógicas, este tipo de ataque é uma tarefa desafiante. Contrariamente aos anteriores, existem tipos mais baratos e simples. São efetuados diretamente ao microcontrolador, através de *In-Circuit-Debugers* (ICDs) ou interfaces programáveis como portas séries ou *Joint Test Action Group* (JTAG) [12].

Relativamente às defesas para mitigar a eficácia deste tipo de ataques, é comum incorporar-se um encapsulamento endurecido e resistente ao dispositivo final. O revestimento terá, contudo, um impacto no custo e complexidade em prol da segurança que visa dificultar a invasão, adulteração ou cópia dos dados aos atacantes. Os ataques diretos ao microcontrolador via interfaces programáveis podem ser evitados, se as mesmas forem removidas do produto final ou recorrendo a mecanismos que privam estas estruturas de interferir com o comportamento do sistema.

Side-channel

Os ataques *side-channel* a par com os de *software* são também muito comuns.

O material que utilizam não é complexo, apenas um PC, osciloscópios ou alguns componentes eletrónicos. Não é invasivo, pois baseia-se na análise das propriedades do sistema enquanto este realiza algum processamento que seja de interesse do atacante (i.e., manuseamento de chaves criptográficas). As propriedades típicas do sistema são: a energia consumida, o tempo de processamento as radiações eletromagnéticas que emite, entre outros. A análise energética é medida através do consumo energético no tempo e requer apenas algum conhecimento de eletrónica e processamento de sinal. Esta possui dois métodos estatísticos conhecidos: *Simple Power Analisys* e *Differential Power Analisys* para a obtenção da chave. A análise através de radiação eletromagnética é semelhante à anterior, todavia ao invés de utilizar um resistência, utiliza uma bobine para determinar precisamente a sua posição sobre o *chip*.

A análise temporal é um ataque capaz de identificar uma chave através de padrões temporais. Uma contramedida para este tipo em específico, é a inserção de instruções inúteis no código (*dummy instructions*) por forma a gerar atrasos aleatórios que confundem o atacante no momento da análise estatística dos dados. Deve ser realizada exaustivamente pois caso contrário, o atacante não terá dificuldades em reordenar os dados e executar o ataque. Para combater a analise energética, é comum mascarar a execução de código com a adição de ruído para, de igual modo, confundir o analista. Além disso, existem técnicas utilizadas para reduzir a proporção de sinal vs ruído, que dificulta a tarefa de realizar um perfil energético fidedigno. A análise energética pode também ser mitigada pelo desenho de instruções específicas, cujo o seu consumo energético é independente dos dados que processam, tornando de igual modo difícil a identificação das respetivas instruções, por meio do seu perfil energético [13].

Trojan

Os ataques *trojan* são considerados os ataques mais "*state-of-the-art*", levando a que o conhecimento neste âmbito seja escasso de pouca consolidação em torno da comunidade cientifica. Baseiam-se em modificações malignas no *hardware*, sob a forma de portas lógicas ou pequenos circuitos, introduzidos por fabricantes não confiáveis durante o desenvolvimento inicial do sistema. As modificações resultam em comportamento indesejado do *hardware* ou fornecem um canal para fuga de informação sensível (*backdoor*). São considerados como sendo ataques furtivos, pois são extremamente difíceis de detetar e a intenção maliciosa para qual foram projetados apenas é desencadeada durante a utilização do sistema "no terreno", mediante um conjunto predefinido de variáveis ou entradas.

Não existem atualmente, soluções concretas que ataquem diretamente este problema. A causa para este cenário, deve-se à recência destes ataques o que por sua vez provoca a escassez de informação fidedigna nesta matéria. É apenas concreto, que não existe uma única solução capaz de proteger o sistema de toda a variedade de *trojans* atuais. Por isso a comunidade científica tem dividido os esforços para a proteção a estes ataques em três abordagens: 1) abordagens de deteção, 2) abordagens de prevenção e 3) abordagens de monitorização. O primeiro tipo foca-se em técnicas ativadoras de *trojans*, por exemplo, por meio de técnicas *side-channel*. Estas técnicas revelam-se ineficazes devido à complexidade que alguns *trojans* podem adquirir, com elevado número de entradas ativadoras. No que toca às abordagens de prevenção, o sistema deve ser desenvolvido considerando este fenómeno, não desprezando as estratégias de deteção e incorporar mecanismos que facilitem a deteção de *trojans*. A monitorização pode ser usada, para em tempo real, desligar partes do sistema após a deteção maligna ou contornar esse problema e possibilitar de novo uma operação segura [10], concedendo deste modo uma capacidade resiliente ao sistema.

Tabela 2.2: Ataques vs Contramedidas

		Ataques			
		Malware	Físicos	*Side-channel*	*Trojans*
Contramedidas	Normas para *Software* Seguro	x			
	Encapsulamento Robusto		x		
	Dummy Instructions			x	
	Adição de Ruído			x	
	Instruções Especiais			x	
	Contramedidas *Trojans*				x

2.2.4 Virtualização

"O termo virtualização está muito relacionado com o da abstração" [11]. Nesta secção, este termo é usado para designar a abstração proporcionada por um hipervisor. Estes componentes abstraem os recursos da plataforma e permitem a execução e co-existência de vários SOs na mesma plataforma física. Estes encontram-se confinados em máquinas virtuais (VMs) que são geridas pelo hipervisor. Os hipervisores substituem os SO como um ponto nevrálgico do sistema [11], pois tendem a ser menos complexos o que permite assegurar mais facilmente a sua segurança.

Os primeiros hipervisores surgiram perto dos anos 60 do século passado, nos *mainframes* da IBM. O uso da virtualização era motivado pelo facto de permitir que a máquina de elevado custo pudesse ser partilhada por vários utilizadores e aplicações. Nos anos 80 e 90, com o aparecimento dos microcomputadores, que apesar de não muito potentes, já executavam o seu próprio SO e o uso dos hipervisores foi escasseando. Isto deveu-se ao fato de os SOs já suportarem a execução de múltiplos programas. Nos finais dos anos 90, começou o ressurgimento a que agora estamos a assistir, contudo os SOs virtualizados, segundo estas técnicas, eram significativamente mais lentos comparativamente com os SOs não virtualizados. Este cenário começou a tornar-se mais animador, por volta de 2001/02, quando dos esforços por parte da comunidade científica resultou o Xen. O Xen possui uma melhor performance face às outras técnicas, pois baseia-se numa técnica chamada para-virtualização. Começou a dar-se uma explosão comercial novamente em torno da virtualização, onde até os próprios fabricantes de processadores começaram a fornecer suporte em *hardware* para aliviar a perda de desempenho.

Existem dois tipos de virtualização:

- Virtualização tipo I ou nativa, o hipervisor corre diretamente no *hardware*.
- Virtualização tipo II ou hospedada,o hipervisor executa no SO nativo.

A para-virtualização consiste em substituir instruções sensíveis e privilegiadas dos SOs das VMs por chamadas ao sistema (*hypercalls*) que emulem o mesmo comportamento, tendo no entanto um nível de privilégio mais alto. É por isso necessário realizar a para-virtualização dos SOs, ou seja, realizar algumas modificações, para executarem sobre os hipervisores.

A virtualização pode ser usada para criar vários níveis de proteção dentro de um sistema com o objetivo de restringir a superfície de ataque, ou seja, caso uma intrusão consiga comprometer algumas partes do sistema, não consiga comprometer partes críticas ou até todo o sistema: 1) uma VM está confinada à visibilidade

(recursos) concedida pelo hipervisor; 2) os SO executam em VM distintas, o que evita que um intrusão num SO se propague aos restantes.

2.2.5 *Hardware* para a Segurança

A ideia de utilizar *hardware* para complementar a segurança nos sistemas computacionais, surgiu sensivelmente em finais dos anos 60 do século passado, altura em que surgiam também os primeiros problemas ao nível da segurança informática. Uma das soluções mais famosas, devido à sua elevada utilização durante a última década até hoje, são os *smartcards*. Têm sido adotados como uma solução clássica em aplicações embebidas, como é o caso dos cartões multibanco, de débito e também em dispositivos móveis nos cartões SIM. São *chips* dedicados de pequenas dimensões. Estes componentes são adequados para as referidas aplicações por serem certificados e por concederem um elevado nível de proteção contra ataques físicos (*tamperproof*). Contém processadores de baixa performance e pequenas quantidades de memória, o que restringe bastante o seu leque de funcionalidades.

Soluções através de *hardware* criptográfico, têm uma vasta história, pois vêm acompanhado desde o início dos problemas relacionados com a confidencialidade da informação. São aceleradores em *hardware* para as morosas operações criptográficas. Contrariamente às soluções mencionadas no início da secção, estas abdicam de uma elavada robustez *tamperproof* em virtude da poupança, performance e conveniência da sua incorporação no produto final.

2.2.5.1 *Trusted Platform Module*

A *Trust Computing Group* (TCG) é um consórcio de empresas como a Microsoft, Intel, IBM, HP e AMD, que em 2000 se uniram com o objetivo de melhorar a segurança nos PCs. O *trusted platform module* (TPM) é o principal componente desenvolvido pelo consórcio e encontra-se em inúmeros modelos de PCs. Este componente é um cripto-processador, composto por unidades criptográficas em *hardware*, assim como, por *software* para inicialização e comunicação, também desenvolvidos e certificados pelo consórcio.

As aplicações deste componente são essencialmente o armazenamento seguro de chaves criptográficas e a verificação da integridade do sistema. Durante o processo de inicialização de uma plataforma (*boot*), ocorre uma sequência de verificações entre todos os componentes intervenientes neste processo. O TPM substitui o componente mais crítico desta sequência, o primeiro, que não pode ser comprometido de maneira alguma: o primeiro componente não possui um outro, anterior, que

faça a sua verificação de integridade. O TPM por se assumir confiável, torna-se o *root-of-trust* da sequência, concedendo *secure boot* ao sistema.

2.2.6 *Trusted Execution Environment*

O *Trusted Execution Environment* (TEE) é um conceito relativamente atual e simboliza um ambiente de execução seguro para as aplicações com maiores requisitos de segurança, e consequentemente com maior probabilidade de serem comprometidas. Os dados sensíveis são armazenados, processados e protegidos no TEE. Forçando a execução protegida de código autenticado, o TEE garante a confidencialidade, autenticidade, privacidade, integridade e controlo de acesso aos dados. Um SO vulnerável executa num ambiente distinto - *Rich Execution Environment* (REE). Aplicações que estão dependentes de infraestruturas perigosas como a Internet, executam no REE, isolados do TEE. O TEE é destinado à execução de serviços que lidam com dados ou recursos sensíveis, estando estes protegidos contra ataques de *software* provenientes do REE.

As duas principais razões para a existência do TEE são: 1) o aumento do número de serviços que requerem elevada segurança; 2) o elevado crescimento do número de utilizadores que exigem que o sistema possua mecanismos de autenticação mais robustos, e de proteção perante ataques por *software*. Para finalizar, o TEE acrescenta valor aos *service providers*, pois: 1) protege as suas aplicações de valor acrescentado assim como 2) a gestão dos seus conteúdos e serviços; 3) concede a autenticação dos seus utilizadores ou associados legítimos; no que toca a *Original Equipment Manufacturers* (OEMs), 4) protege as propriedades do dispositivo contra ataques de *software* executados por utilizadores ou aplicações maliciosas.

2.2.7 ARM TrustZone

A tecnologia TrustZone, também denominadas por extensões de segurança (*security extensions*), foram introduzidas nas arquiteturas ARM de forma gradual. A arquitetura ARMv6, através do processador ARM1176, foi a primeira a ser detentora da tecnologia. Esta disseminou-se posteriormente para os processadores de perfil aplicacional Cortex-A, e recentemente, encontra-se nos processadores Cortex-M. Apesar do sucesso e aceitação que se tem assistido em torno da tecnologia atualmente, nos primeiros anos após o seu aparecimento (2003), a adesão por parte do público em geral foi baixa, tal deve-se ao secretismo em torno da tecnologia consentido pelo seu fabricante. Contudo e como já foi referido, a tecnologia é atualmente detentora de uma aplicabilidade inegável, sendo o seu nome,

do ponto de vista de *marketing*, usado por fabricantes para "rotular" a segurança nos seus produtos. A tecnologia TrustZone é um conjunto de extensões no *hardware* de várias subpartes do sistema computacional ARM. Estas são incorporadas durante as primeiras fases de conceção, permitindo à solução de segurança ser mais flexível, comparativamente com outras soluções de perfil funcional estático (i.e., TPM). A flexibilidade deve-se ao fato de o comportamento de um sistema de segurança baseado na TrustZone ser definido por *software* ao invés de estar *hard-wired*.

A inovação trazida pela tecnologia reside na divisão da execução do processador em dois mundos distintos, fortemente isolados entre si: mundo seguro e mundo não-seguro. O isolamento é alcançado devido às particularidades arquiteturais, que como já foi referido, se estendem por várias partes do sistema: processador, hierarquias de memória (sistema interno e externo), barramento, infraestruturas de *debug* e periféricos. A tecnologia TrustZone propaga o controlo de acesso através do barramento aos vários componentes do sistema, o que permite um nível de segurança em cada transação do processador. Este nível de granularidade torna o sistema mais robusto contrastando com outras soluções puramente em *software*, como a virtualização. Esta última, recorre à *Memory Management Unit* (MMU) para atribuir permissões de acesso que apenas se situam ao nível da *page table*.

A tecnologia é extremamente adequada ao desenvolvimento de TEEs, dependendo dos recursos da plataforma, executa no mundo não-seguro um sistema operativo de propósito geral (GPOS). Estes são componentes de risco para o sistema, pois possuem um elevado número de vulnerabilidades inerentes à sua complexidade e dimensão. O isolamento da TrustZone assegura que mesmo estando o GPOS comprometido, este não compromete o *software* que executa no domínio seguro. O mundo seguro foi desenhado para conceder uma elevada proteção ao *software* que execute sobre o mesmo. A *Trust Computing Base* (TCB) de um sistema TrustZone está confinada maioritariamente ao domínio seguro, e por isso tudo que resida neste mundo é crítico para a segurança e bom funcionamento do sistema. Deve existir um cuidado especial no desenvolvimento do *software* seguro, no sentido de reduzir a sua complexidade e consequentemente o seu número de vulnerabilidades.

Com esta tecnologia a ARM criou bons alicerces para o desenvolvimento de um ecossistema altamente seguro. No entanto, estes alicerces devem ser cimentados com *software* cuidadosamente codificado. A tecnologia TrustZone incentiva uma vasta cadeia de valor, que dado o enquadramento na segurança e a sua demanda atual, obriga a todos os parceiros (OEMs, *service providers*, *developers*)

uma exigente e profissional cooperação.

2.2.7.1 Soluções baseadas em TrusZone

Serão seguidamente apresentadas *frameworks* que exploram a tecnologia TrustZone, sendo estas o FreeTEE e o *Lightweight* TrustZone-*assisted Hypervisor* (LTZVisor), ambas desenvolvidas *in-house*, o SierraTEE da SierraWare e o SeCReT.

FreeTEE

O FreeTEE, representado na Figura 2.3, é uma *framework* que tira partido das extensões de segurança para implementar um TEE que se destaca pela à sua característica *real-time* [14].

No mundo seguro, o FreeRTOS é responsável por executar tarefas de tempo-real e por conceder serviço seguros a aplicações clientes que executam no Linux, no mundo normal. Aos serviços seguros foram atribuídos uma prioridade inferior às tarefas de tempo-real, por forma a preservar o comportamento nativo do FreeRTOS. Quando o FreeRTOS não se encontra a executar qualquer tarefa (*idle*), o controlo de execução é enviado ao ACVisor através da exceção *Secure Monitor Call* (SMC). O ACVisor por sua vez realiza o *context-switch* (guarda o estado do FreeRTOS e restaura o contexto do Linux), ficando o GPOS com total controlo de execução. Por forma a garantir o comportamento nativo do FreeRTOS, as interrupções FIQ estão associadas ao ACVisor. Sempre que ocorre um *tick* do FreeRTOS, o *context-switch* é realizado na ordem inversa e o controlo de execução é devolvido ao FreeRTOS onde este verificará a existência de tarefas prioritárias (*real-time*) prontas a executas. O *overhead* deste processo é mínimo, preservando o determinismo do sistema operativo de tempo-real (RTOS).

Aplicações no Linux devem poder comunicar com serviços seguros, para isso o FreeTEE, é complacente com a especificação TrustZone *Application Programming Interface* (TZAPI). A TZAPI usufrui de um *device driver* que se encontra em *kernel space* no Linux, responsável por implementar todos os detalhes da comunicação.

SierraTEE

A SierraWare desenvolveu e mantém o projeto *OpenVirtualization*, *open source*, que por sua vez mantém o SierraTEE, tanto para plataformas virtuais como plataformas físicas, como ARM Cortex-A8, Cortex-A9 e Cortex-A15.

O SierraTEE tira partido da tecnologia TrustZone, para a criar um ambiente de execução seguro. O *kernel* foi elaborado de forma otimizada para possuir um

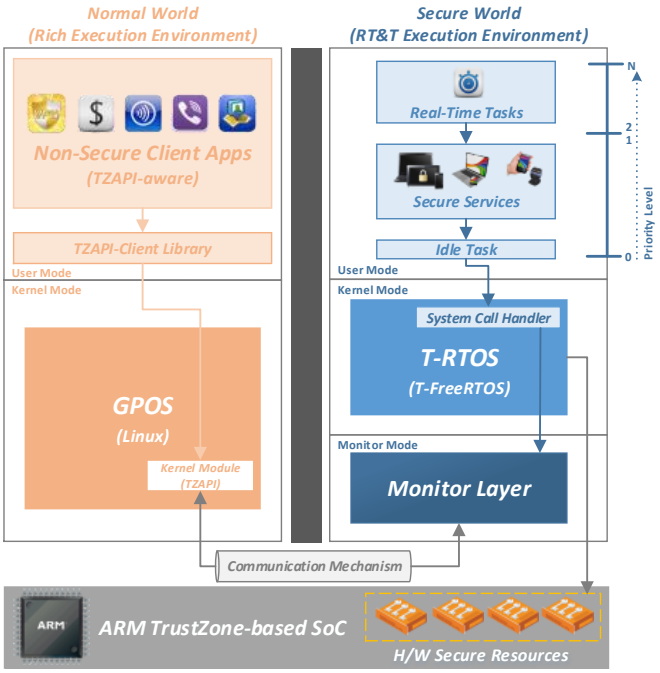


Figura 2.3: Arquitetura geral do FreeTEE

pequeno tamanho e elevada performance. Deste modo, permite a execução de um *kernel* minimalista em "paralelo" com um SO tradicional. A comunicação com o *kernel* seguro é possível devido à existência de um *driver* no GPOS. Existe também uma camada Monitor, alojada no domínio seguro, responsável por comutar e realizar o *context-switch* entre os dois OSs [15].

As aplicações do GPOS executam sem qualquer privilégio, em modo *user*, o *kernel* do SO em *system mode*. Por sua vez o *kernel* seguro executa em modo monitor, o mais privilegiado. Assim sendo, uma aplicação contaminada mesmo tendo privilégios *rooted* não conseguirá ter acesso às regiões protegidas dentro do sistema. O fato de o SierraTEE tirar partido da tecnologia TrustZone, permite proteger completamente o *kernel* seguro e periféricos de qualquer código que execute no mundo normal.

As especificações definidas pela GlobalPlatform são adotadas neste projeto, o que estimula o desenvolvimento de serviços seguros e aplicações cliente. De notar também que esta *framework* foi pioneira no uso da TZAPI de forma *open source*, o que serviu posteriormente de base para o desenvolvimento da *Application Programming Interface* (API) da GlobalPlatform para aplicações clientes.

SeCReT

O SeCReT surge da necessidade de melhorar a segurança em volta da tecnologia TrustZone, e sucintamente constrói um canal de comunicação seguro entre o

REE e o TEE. A *framework* é compatível com arquiteturas ARM Cortex-A15 *dual core*.

A TrustZone não possui mecanismos de autenticação aos seus recursos, [16] Uma aplicação cliente (no REE) para utilizar os recursos na TrustZone cria um canal de comunicação com a mesma. O canal é baseado em memória partilhada, alocada no REE, e é vulnerável a um atacante com privilégios ao nível do *kernel* deste ambiente. O atacante pode criar um processo maligno que continuamente invoque pedidos manipulados (SMCs) com a finalidade de descobrir vulnerabilidades no *kernel* seguro. O SeCReT constrói de um canal de comunicação seguro, garantindo que apenas processos legítimos têm acesso aos recursos críticos alojados TrustZone.

A *framework* propõe dois requisitos de segurança de maneira a reduzir a superfície de ataque da tecnologia TrustZone: 1) os acessos aos recursos críticos são restritos a processos que integram uma lista pré-definida, mantida no mundo seguro; 2) as mensagens trocadas através do canal são encriptadas com uma chave (armazenada na TrustZone) para comunicação segura. As chaves usadas durante uma sessão de comunicação são atribuídas simetricamente a ambos os mundos, no entanto, o SeCReT protege a chave no REE, que é considerado como não seguro. O acesso ao local de memória que contém a chave é vigiado pelo SeCReT, que apenas concede acesso à chave a processos legítimos. A Figura 2.4 apresenta o funcionamento geral do SeCReT.

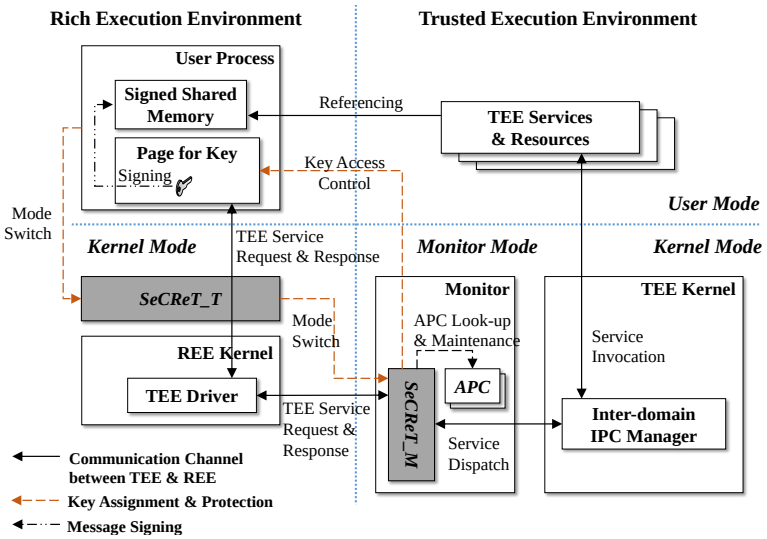


Figura 2.4: Arquitetura geral do SeCReT

LTZVisor

O LTZVisor é uma *framework* desenvolvida *in-house* e encontra-se apresentada na Figura 2.5. Esta é resultado da investigação académica realizada em torno da tecnologia TrustZone para assistir a virtualização em aplicações de tempo-real.

Dado o *expertise* dos autores nesta área, o LTZVisor foi construído tendo como base três princípios: 1) implementação minimalista, ambicionando a segurança geral da solução. O facto da *framework* tirar o máximo partido do suporte em *hardware* concedido pela tecnologia, assim como, adotar uma configuração estática das VMs, assegura uma TCB reduzida ao sistema; 2) princípio do menos privilegiado, ou seja, os diferentes níveis de privilégios concedidos pela tecnologia são explorados para assegurar o isolamento entre o ambiente de tempo-real e outro ambiente não crítico; 3) recorrendo a um escalonamento assimétrico, os autores mitigam um problema que é ortogonal à virtualização de sistemas de tempo-real: hierarquia do escalonamento, isto é, assegurar o determinismo de tarefas de tempo-real escalonadas numa VM, esta por sua vez escalonada pelo hipervisor.

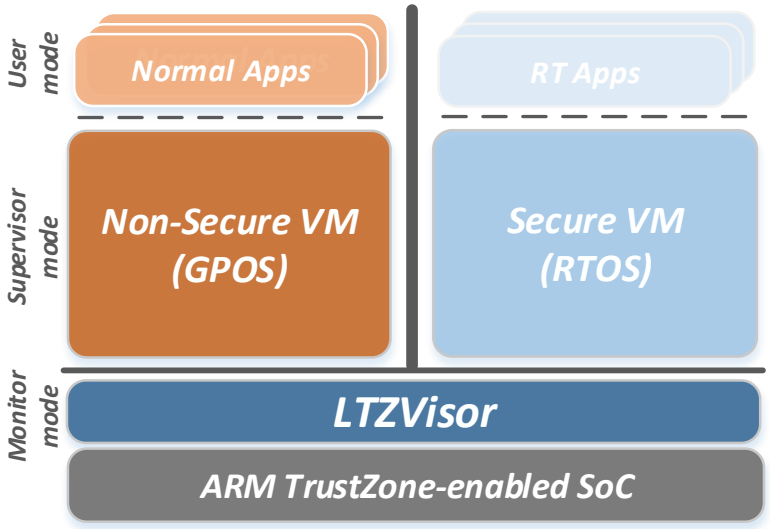


Figura 2.5: Arquitetura geral do LTZVisor

Especificação do Sistema

No presente capítulo é introduzida algumas noções sobre a arquitetura alvo - ARM (secção 3.1), assim como é detalhado a tecnologia principal da dissertação - ARM TrustZone (secção 3.1.2). Posteriormente a discussão foca-se no barramento *Advanced eXtendible Interface* (AXI) (secção 3.2) e GlobalPlatform API (secção 3.3), por serem tecnologias determinantes para a dissertação. Na parte final, aborda-se as ferramentas necessárias ao desenvolvimento (secção 3.4 e 3.5).

3.1 Arquitetura ARM

A ARM segue a filosofia *Reduced Instruction Set Computing* (RISC). A filosofia RISC baseia-se na utilização de um conjunto reduzido, mas poderoso, de instruções. As características que fazem parte da identidade da arquitetura RISC são: 1) tempo de execução num ciclo máquina, mesmo a elevadas frequências de relógio; 2) utilização do *pipeline*, para o processamento simultâneo de instruções nos seus diferentes estágios; 3) elevado número de registos, para evitar elevadas interações com a memória. A arquitetura RISC coloca sobre o compilador um elevado esforço para a geração de instruções simples perante código complexo. Em contraste, uma arquitetura *Complex Instruction Set Computing* (CISC) depende mais do *hardware* para as funcionalidades das instruções, o que por sua vez torna esta arquitetura mais complexa.

Os processadores ARM não seguem, no entanto, arquiteturas puramente RISC. Utilizam instruções com ciclos de relógio variáveis, em particular para o acesso à memória. Os sistemas embebidos regem-se mais pela eficiência da sua performance e baixo consumo energético, do que propriamente por elevadas velocidades de processamento, como a pura filosofia RISC ambiciona.

3.1.1 Conceitos básicos

A arquitetura ARM possui um conjunto de dezasseis registos (r0 a r15) de 32-*bits* para propósito geral. Os primeiros quinze registos podem ser utilizados para o armazenamento de informação genérica, contudo o valor do décimo sexto (r15) é alterado à medida que o processador executa instruções. Ao subconjunto de

registos r13, r14 e r15 é atribuído uma identificação própria, devido à sua função específica: r13 - *stack pointer* (sp), armazena o topo da *stack*; r14 - *linker register* (lr), contém o endereço de retorno quando é invocada uma rotina; r15 - *program counter* (pc). Existem também dois registos, o *current status processor register* (cpsr) e o *saved status processor register* (spsr). O spsr armazena o valor cpsr relativo ao modo em que o processador executou anteriormente. O cpsr fornece informação sobre o modo atual de execução e estado do processador.

O processador ARM opera em diferentes modos que determinam o nível de privilégio da operação atual. Existem até três níveis de privilégios: PL0, PL1 e PL2. O PL0 é o nível privilegiado para as execuções não privilegiadas. O PL1 concede execução privilegiada em todos os modos, à exceção do modo hipervisor. PL2, concede acesso a todas as características do sistema.

Antes da introdução das extensões de segurança TrustZone, o processador ARM possuía sete modos. Haviam seis modos privilegiados e um modo não-privilegiado - *user mode*. Cada modo possui um subconjunto de registos próprios, fixos no *central processing unit* (CPU). Denominam-se de registos banqueados, pois encontram-se mascarados durante períodos de execução e geralmente são usados para guardar o sp e lr do respetivo modo. A introdução das extensões TrustZone, criou dois estados de segurança que são independentes do nível de privilégio e modo do processador. Um novo modo *monitor* faz a gestão do acesso a esses estados, e aos vários modos existentes em cada um deles. As extensões de virtualização da ARM adicionaram também um novo modo privilegiado, *hypervisor*, para possibilitar a operação de múltiplos SO na mesma plataforma. Na Tabela 3.1 encontram-se descritos todos os modos e respetivos níveis de privilégios existentes na arquitetura ARMv7 com extensões de segurança e virtualização.

Tabela 3.1: Modos do processador da arquitetura ARMv7

Modo	**Função**	**Nível de Privilégio**
User (USR)	Modo de execução não privilegiado para as aplicações	PL0
FIQ	Entrada através de um interrupção FIQ	PL1
IRQ	Entrada através de um interrupção IRQ	PL1
Supervisor (SVC)	Entrada no reset ou na execução de uma instrução SVC	PL1
Monitor (MON)	Implementado como parte das extensões de segurança	PL1
Abort (ABT)	Entrada derivado de um acesso falhado à memória	PL1
Hypervisor (HYP)	Implementado como parte das extensões de virtualização	PL2
Undef (UND)	Entrada na execução de uma instrução indefinida ou não suportada	PL1
System (SYS)	Entrada através da escrita no registo cpsr	PL1

O banco de registos de um processador ARM, com extensões de segurança e virtualização, é composto no total por quarenta e três registos, dos quais vinte e seis são escondidos da aplicação em diferentes intervalos no tempo. Os registos

banqueados encontram-se disponíveis quando o processador se encontra num modo específico de execução. Estes registos encontram-se destacados na Tabela 3.2. À exceção do modo *system*, todos os outros modos possuem registos banqueados. Quando o modo do processador comuta, o registo banqueado do novo modo irá substituir o respetivo registo existente. Os restantes modos, à exceção do modo *hypervisor*, possuem os registos sp, lr e spsr baqueados. O modo *fast interrupt request* (FIQ) possui banqueados, para além dos já referidos, o subconjunto de registos de r8 a r12.

Tabela 3.2: Conjunto total de registos da arquitetura ARMv7

USR	**SYS**	**SVC**	**ABT**	**UND**	**IRQ**	**MON**	**HYP**	**FIQ**
r0								
r1								
r2								
r3								
r4								
r5								
r6								
r7								
r8								r8_fiq
r9								r9_fiq
r10								r10_fiq
r11								r11_fiq
r12								r12_fiq
r13 (sp)		sp_svc	sp_abt	sp_und	sp_irq	sp_mon	sp_hyp	sp_fiq
r14 (lr)		lr_svc	lr_abt	lr_und	lr_irq	lr_mon	r14 (lr)	lr_fiq
r15 (pc)								
CPSR								
SPSR		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_mon	SPSR_hyp	SPSR_fiq

O modo atual de execução do processador é alterado sobre controlo de *software* privilegiado (i.e., escrita, no cpsr) ou quando ocorre uma exceção. Uma exceção é uma interrupção da execução atual do processador. As exceções podem ser interrupções externas assíncronas (i.e. FIQ e IRQ), acessos inválidos à memória (i.e., *Aborts*), *Reset* da plataforma ou através da execução de instruções específicas (i.e., SVC, SMC, SWI). Na presença de uma exceção, o processador comuta para o modo associado e executa uma rotina própria que lida com a exceção - *exception handlers*. Estes *entry-points* são estruturados em vetores e encontram-se armazenadas em tabelas, que possuem um endereço fixo. Vetores associados às determinas exceções estão por isso localizados em *offsets* fixos relativos ao endereço da tabela de vetores - *vector table*.

A Tabela 3.3 apresenta as diversas exceções suportadas por um processador ARM.

Uma interrupção *interrupt request* (IRQ) não é capaz de interromper a execução do processador quando este se encontra a executar em modo FIQ. A interrupção IRQ encontram-se desativadas no processador enquanto este processa uma interrupção FIQ. Assim sendo, as FIQs são consideradas as mais prioritárias.

Tabela 3.3: *Vector table*

Exceção	***Offset***	Descrição
Reset	0x00	Usada para executar código de *boot*, após *power on*.
Undefined instruction	0x04	Usada quando não é possível descodificar de uma instrução
Software interrupt	0x08	Invocado após a execução da instrução SWI
Prefetch abort	0x0C	Falha em fazer o *fetch* de uma instrução da memória
Data abort	0x10	Falha em aceder aos dados em memória
Reserved	0x14	Sem finalidade, reservado
Interrupt request	0x18	Usado por *hardware* externo
Fast Interrupt Request	0x1C	Usado por *hardware* externo crítico

A exceção FIQ possui também vantagens no que toca à velocidade do seu processamento. A sua posição na *vector table* liberta o processador do esforço de realizar um *branch* para um endereço específico e como possui mais registos baqueados que o modo IRQ, o *context-switch* entre os modos é realizado de forma mais rápida.

3.1.1.1 Extensões do Processador

A ARM acopolado aos seus processadores envolve *harware* de suporte a multiplas funções. As principais extensões dos processadores são as *caches*, a MMU e os coprocessadores. Com a *cache* o processador consegue um aumento significativo de performance, mas tendo o custo de retirar determinismo ao sistema. A MMU organiza todas a estruturas de memória e, ao mesmo tempo, protege-as de aplicações que as tentam aceder sem as devidas permissões. Por último, os coprocessadores estendem, de uma forma elegante, o *Instruction Set Architecture* (ISA) da arquitetura. São unidades secundárias de processamento que lidam com instruções, enquanto o processador continua o seu processamento. Existem 16 coprocessadores (CP0 a CP15). O *System Control Coprocessor* (CP15) é principalmente usado para gerir *caches* e configurar a MMU. O coprocessadores (CP)14 é destinado à depuração e o CP10 e CP11 são utilizados para as tecnologia NEON e *Floating Point*, respetivamente.

3.1.2 ARM TrustZone

Nesta secção é abordada a tecnologia TrustZone de forma técnica. É explicado de forma profunda, e em primeiro lugar, a arquitetura em *harware* (secção 3.1.2.1) e, seguidamente, a arquietura em *software* (secção 3.1.2.2).

3.1.2.1 Arquitetura do *Hardware*

A arquitetura TrustZone implicou mudanças ao nível do processador e seus periféricos internos (i.e., hierarquias e mecanismos de gestão da memória, sistema

de interrupção). Estas funcionam como um 33º *bit* que propaga a coerência de segurança a todo o sistema.

Processador

Um novo modo Monitor foi incorporado à arquitetura, como se observa na Figura 3.1. Este modo criou dois estados de segurança ortogonais ao nível de privilégio e modo do processador. Assim sendo, este componente é a interface entre dois novos "mundos": Seguro e Não-seguro, e os tradicionais modos que existem, de forma independente, em cada um deles.

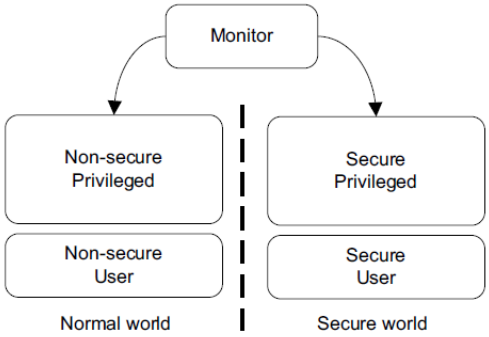


Figura 3.1: Mundo Seguro e Não-Seguro

Um conjunto de registos banqueados foram adicionados tanto ao modo Monitor (sp_mon, lr_mon, e o spsr_mon) como a vário coprocessadores. Na Tabela 3.2 podem ser observados os registos introduzidos pela ARM para suporte ao novo modo Monitor.

Secure Monitor Call

O ISA ARMv7 foi estendido com a instrução SMC, que é o mecanismo usado pelo mundo não-seguro para entrar no modo Monitor de forma síncrona. Contudo também pode fazê-lo de forma assíncrona, através exceções geradas pelo *hardware*. O *trap* das IRQ, FIQ, *external Data Abort* e *prefetch Data Abort* para o modo Monitor pode ser realizado através da configuração do registo *Secure Configuration Register* (SCR) para esse efeito. A SMC usa um parâmetro de 4-*bits* codificada na própria instrução, podendo ser usados como base para a construção de protocolos de comunicação entre os dois domínios.

Secure Configuration Register

O registo SCR é um registo do CP15, introduzido com as extensões de segurança, que só pode ser acedido no mundo seguro em modo privilegiado. Uma tentativa de acesso indevida resulta numa exceção *Undifined Instruction*.

Os campos deste registo, permitem ao monitor ter controlo sobre o mundo seguro e não-seguro. Através deste registo, é possível identificar e alterar o mundo de execução do processador, assim como alterar o comportamento do processador aquando da ocorrência de interrupções externas. As características de com maior relevo deste registo são as seguintes:

- Como foi referido, permite alterar o mundo de execução do processador, através da escrita no NS *bit*. Em modo monitor, independentemente do valor deste *bit*, o estado de execução é sempre considerado como seguro (NS é 0). O valor deste *bit* têm influência na visibilidade de vários registos, destinados a cada mundo em particular no CP15;

- Possibilita o *trap* das interrupções FIQ, IRQ e *Data Abort* para o modo monitor, através da escrita nos bits FIQ, IRQ e *external abort* respetivamente. Por defeito, o comportamento de execução do processador não sofre modificações, significando que o atendimento à exceção será realizado no modo associado;

- Impede o mundo não-seguro de modificar o valor dos *bits* F e A do cpsr. Ou seja, o mundo-não seguro pode ser impossibilitado de interferir com as interrupções FIQs e *Data Aborts*;

Derivado do papel que o modo monitor desempenha no sistema, é recomendado pela ARM, que apenas este modo altere o *bit* NS. O SCR permite configurar características comportamentais do sistema que têm implicações na sua segurança.

Gestão da memória

As extensões de segurança influenciaram os componentes pertencentes à gestão da memória (i.e., MMU, *cache* e memória externa). A forma elegante como a ARM as introduziu, não prejudicou no entanto, a performance dos seus processadores. O *software* lida com um espaço de endereçamento virtual, a MMU realiza a tradução desse espaço de endereçamento em endereços físicos, fazendo uso de *translation tables* alocadas em memória. Esta é também responsável por conferir atributos à memória, como permissões de acesso. A *cache* é uma memória de rápido acesso que suaviza o esforço do processador quando este tem intenções de aceder à memória externa.

Numa arquitetura TrustZone, cada processador virtual possui uma MMU virtual. Uma das novidades é o novo bit *Non Secure Table Identifier* (NSTID) incorporado em todas as entradas existentes na *Translation Lookaside Buffer* (TLB). A função da TLB é armazenar as entradas das *translation tables*. Devido ao novo

bit, é possível a coexistência de entradas respetivas às *translations tables* de cada mundo. Nas *caches*, este tipo de coexistência também passou a ser possível com um *bit Non-Secure* (NS) para cada linha de *cache*, que grava o respetivo mundo a que pertencem. Como referido, a performance do sistema não foi afetada devido às características da TLB e da *cache*, que permitem a coexistência de informações relativas aos dois mundos, descartando a necessidade de realizar o *flush* das entradas da TLB, das linhas de *cache* e das instruções existentes no *pipeline* a cada comutação de mundos.

O TrustZone *Address Space Controller* (TZASC) é responsável por verificar a segurança das transações à memória. Possui uma interface AXI *master* responsável por prolongar a coerência ao nível da segurança através do sinais AWPROT[1] e ARPROT[1] (Tabela 3.4). O TZASC suporta até dezasseis regiões, para cada, disponibiliza um registo de configuração das permissões de acesso, que depois usa para aprovar ou reprovar a transação. O controlador TZASC é principalmente usado para particionar a memória (segura ou não-segura). Existe ainda, o TrustZone *Memory Adapter* (TZMA) que oferece uma funcionalidade similar mas relativa à memória externa, como a memória ROM.

As permissões de acesso à memória concedidas por este controlador, originam uma das características mais peculiares da tecnologia TrustZone, a separação virtual da memória.

Gestão das interrupções

No que respeita às interrupções, a tecnologia TrustZone introduziu algumas alterações no modelo de interrupções tradicional da ARM: o *trap* das interrupções FIQ e IRQ para o monitor pode ser realizado sem intervenção de código intermediário de cada mundo. Apenas existe o *trap* para o Monitor se o mundo de origem da interrupção não corresponder ao mundo atual de execução. Neste caso, o *hardware* encaminha a execução para o Monitor, que realizara a comutação de contextos entre os mundos e fornece o controlo de execução ao mundo restaurado. Este, por fim, lida localmente com a interrupção.

A ARM aconselha a atribuição das interrupções FIQs ao mundo seguro e as IRQs ao não-seguro, isto é possível devido à existência de três tabelas de exceções, pertencentes ao modo Monitor, mundo seguro e mundo não-seguro. A localização de cada tabela pode ser definida em *run-time*, através da escrita no registo VBAR apropriado, alojado no CP15.

Por fim, o *Generic Interrupt Controller* (GIC) suporta a existência de fontes de interrupção seguras e não-seguras. Permite a configuração das interrupções

seguras com maior prioridade que as não seguras, e fornece diferentes modelos de configuração que permitem a atribuição das IRQs e FIQs para fontes de interrupção seguras ou não-seguras.

Barramento AXI

O protocolo AXI da especificação *Advanced Microcontroller Bus Architecture* (AMBA) integra sinais de permissões de acesso que podem ser usados para proteger o sistema contra transações ilegais (Tabela 3.4). Os sinais ARPROT[2:0] e AWPROT[2:0] definem as permissões de acessos em transações de leitura e escrita, respetivamente.

Um dos atributos de proteção concedidos pelo barramento, diretamente relacionado com a TrustZone, são a identificação das transações como sendo seguras ou não-seguras, através do *bit* AxPROT[1]. Por forma a manter a consistência com os outros sinais da tecnologia, a especificação considera a transação como sendo não-segura quando este *bit* é 1.

Tabela 3.4: *Bits* de proteção do barramento AXI

ARPROT[2:0] AWPROT[2:0]	Valor	Nível de Proteção
[0]	1	acesso privilegiado
	0	acesso não privilegiado
[1]	1	acesso não seguro
	0	acesso seguro
[2]	1	*instruction access*
	0	*data access*

Um periférico *master* AXI pode suportar estados de operação seguros e não-seguros e estender este conceito de segurança até ao acesso à memória. O valor do *bit* depende do estado do *master* quando este realiza uma transação. O *slave* e o barramento devem interpretar o valor do *bit* por forma a garantirem o isolamento de segurança. Contudo, a própria implementação do protocolo tem definido que uma transação de um *master* não-seguro a um *slave* seguro resultará em falha. A falha pode passar silenciosamente despercebida ou gerar um erro (SLVERR ou DECERR), dependendo da configuração do protocolo do barramento ou do *design* do *slave*.

Barramento *Advanced Peripheral Bus*

A tecnologia TrustZone recorre ao protocolo *Advanced Peripheral Bus* (APB) para prolongar o estado de segurança aos periféricos externos ao *System on Chip*

(SoC). O ambiente de segurança é desta feita estendido a problemas não apenas relacionados com o processamento seguro de dados. O componente AXI-*to*-APB funciona como uma ponte, anexando do barramento APB ao barramento do sistema (AXI).

Suporte à infraestrutura de depuração

Anteriormente à introdução da TrustZone, a infraestrutura de depuração era dotada de apenas dois sinais de controlo de acesso ou visibilidade à depuração no processador:

- *Invasive Debug Enable* (DBGEN) - Define se o *debug* invasivo é permitido ou não, em todo o processador. *Degug* invasivo caracteriza-se por interferir com a execução do processador, como o que ocorre através de JTAG;
- *Non Invasive Debug Enable* (NIDEN) - Controla se o *debug* não invasivo é permitido ou não, em todo o processador. Um *debug* não invasivo não interfere com a execução do processador, apenas recolhe informação sobre o estado de execução como o que ocorre em componentes de instrumentação passiva de *trace*;

O suporte concedido com o advento das extensões de segurança, apresenta-se principalmente na forma de dois sinais, que permitem o mesmo nível de configurabilidade, mas exclusivos ao mundo seguro. São eles o *Secure Priviledge Invasive Debug* (SPIDEN)e o *Secure Priviledge Non Invasive Debug* (SPNIDEN).

Através do estado destes sinais todos os componentes associados à infraestrutura de *debug*, como o monitorizador de performance, componentes de instrumentação passiva ou ativa, entre outros, encontram-se confinados à visibilidade adjacente a estes. Este suporte no *debug*, é uma eficaz proteção contra poderosos, mas simples ataques de *hardware*. Contudo, é importante notar que, estes sinais podem permitir ao mundo não-seguro afetar ou monitorizar o mundo seguro e por isso a ativação destes sinais apenas deve ser realizada mediante um ambiente considerado seguro.

3.1.2.2 Arquitetura do *Software*

Nesta secção são levantadas algumas considerações sobre a possível implementação do *software*, e de que forma este pode retirar eficazmente partido da proteção concedida pela TrustZone.

Software do mundo seguro

O mundo seguro requer uma implementação em *software* que tire partido dos recursos protegidos na tecnologia TrustZone. Não existe uma arquitetura de *software* única para o mundo seguro, no entanto, uma abordagem poderosa e complexa como um SO facilita o aproveitamento eficiente dos recursos. O SO proporciona um ambiente de execução a múltiplos serviços seguros e permite o *download*, em *run-time*, de novas aplicações de segurança ou *patches* que complementam a segurança destas.

A arquitetura da tecnologia TrustZone possui algumas particularidades específicas que potenciam a existência de um RTOS no mundo seguro. A atribuição das interrupções FIQ ao mundo seguro e as IRQ ao não-seguro, é a forma aconselhada para garantir o comportamento determinístico no mundo seguro. As interrupções FIQ, nesta abordagem, funcionam como porta de entrada periódica para o mundo seguro. É garantido pelo *hardware* que o mundo não-seguro não consegue controlar ou influenciar a respetiva interrupção.

Camada Monitor

Como já foi referido, o papel do Monitor é realizar a interface entre os dois mundos. Esta camada gere as mudanças de contexto: assegura que o estado do mundo onde o processador se encontra é gravado e, restaura o estado do mundo em que o processador irá executar. O Monitor deve armazenar os estados do mundo seguro em localizações seguras de memória, onde o mundo não-seguro não terá qualquer acesso. Concretamente, o que necessita de ser restaurado ou gravado depende diretamente do *hardware* da plataforma, contudo tipicamente são: 1) os registos de propósito geral; 2) os registos usados em alguns coprocessadores; 3) a configuração do CP15 que seja dependente ao mundo.

O Monitor é um componente crítico do sistema, pois interage consecutivamente com um ambiente que não oferece quaisquer garantias de segurança. Por isso, esta camada dever ser desenvolvida de forma minimalista e, durante a sua execução, não deve possuir as interrupções habilitadas, uma vez que lidar com o comportamento destas interrupções adicionaria uma complexidade considerável a esta camada, o que não é aconselhável.

3.2 AXI

O AXI, atualmente na versão AXI4, é parte do especificação de barramentos AMBA 3.0 (AMBA). Foi lançado em 1996 e, desde então, tem vindo a ser reformulado e otimizado, sendo que atualmente é considerado pela ARM como sendo o protocolo *standard* para comunicações em SoCs. Existem três tipos de interfaces: AXI-*Lite*, AXI-*Full* e AXI *Stream*. As interfaces diferem entre si, pois são específicas para diferentes aplicações.

O barramento AXI possui como principais características: 1) fases de controlo separadas das fases de dados. Numa conexão a fase de controlo é a primeira que ocorre, consiste na passagem de dados para controlo da transação. Na segunda fase ocorre a transferência dos dados própriamente ditos: 2) identificadores de *bytes* a escrever na *word* propriamente dita, ou seja, permite a escrita desalinhada de dados; 3) possibilidade de transações em modo *burst*, e por último, 4) canais de escrita e leitura separados.

As interfaces AXI dividem-se em duas categorias tendo em conta o seu modo de endereçamento que ocorre durante transferência de dados: *memory-mapped* e *point-to-point*. A primeira é endereçável (acessível através de endereços em memória) e contêm cinco canais, enquanto que as *point-to-point* não o são e, possuem apenas um canal de dados, sendo ainda a única interface unidirecional.

Estas interfaces apresentam ainda dois modos dentro de cada uma delas: o modo *master* e o modo *slave*. A interface em modo *master* caracteriza-se por controlar as transações, ou seja, é esta que decide a direção das transações e é responsável por começar cada uma das transações. Por outro lado, o modo *slave* apenas se limita a responder às transações invocadas pelo *master*, sendo a única com a responsabilidade (nas interfaces *memory-mapped*) de dar uma resposta positiva ou negativa às transações de escrita.

As interfaces *memory-mapped* (AXI-*Lite*, AXI-*Full*), como anteriormente referenciado, possuem cinco canais de fluxo de dados: dois usados para escrita no periférico, dois para leitura do periférico, e ainda um quinto canal. O quinto, depende do modo (*master* ou *slave*) e é usado para responder às transações de escrita invocadas por parte do *master*. Nas transações de leitura não existe *feedback* do *master*, pois este simplesmente poderia recomeçar a transação em caso de insucesso. Para cada par de canais (escrita ou leitura), um representa os endereços e outro representa os dados, e em ambos existe sinais para o controlo das transações. No caso de uma interface *point-to-point* (AXI-*Stream*) existe apenas um canal disponível que irá conter tanto os sinais de controlo como os dados.

3.2.1 AXI-*Lite*

Esta interface é, como foi supracitado, uma interface *memory-mapped*. Das interfaces em estudo caracteriza-se por ser a mais simples, com menos sinais de controlo, sendo por isso a interface de mais fácil implementação (em *hardware*), e também a mais intuitiva e fácil de usar do ponto de vista do utilizador.

Por forma a garantir a sua simplicidade e um número reduzido de sinais de controlo, esta interface apenas garante uma transação de dados por conexão (*burst* de tamanho 1). Para cada conexão será necessário realizar o *handshake*, enviar o endereço, enviar a *word* de dados e esperar pelo *acknowledge* da transação. Algumas das características específicas a esta interface são: 1) canal de dados com tamanho de 32 ou 64 *bits* (preferencialmente 32-*bits*, recomendado pela ARM); 2) sinal *strobe* pode ser opcional pela interface modo *slave*; 3) Transações *burst* de tamanho 1;

Na Tabela 3.5 são organizados os sinais da interface AXI-*Lite* por canal, enquanto na Tabela 3.6 são retirados os sinais da tabela anterior e são explicados os seus propósitos nas diferentes transações.

Tabela 3.5: Sinais por cada canal AXI-*Lite*

Globais	**Canal de escrita de endereços**	**Canal de escrita de dados**	**Canal de resposta à escrita**	**Canal de leitura de endereços**	**Canal de leitura de dados**
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
	AWADDR	WDATA	BRESP	ARADDR	RDATA
	AWPROT	WSTRB		ARPROT	RRESP

Tabela 3.6: Sinais AXI-*Lite*

Sinal	**Origem do sinal *Master* ou *Slave***	**Descrição**
ACLK	*Clock*	Sinal de *Clock*
ARESETn	*Reset*	Sinal de *Reset* (activo a *low*)
AWVALID/AWVALID	*Master*	Indica que as informações de endereços e controlo são válidas
AWREADY/ARREADY	*Slave*	Indica que o *Slave* está pronto para aceitar os sinais de controlo
AWADDR/ARADDR	*Master*	Valores do endereço de escrita/leitura
AWPROT/AWPROT	*Master*	Indica o nível de privilegio e segurança da transação
WVALID	*Master*	Indica que as informações de controlo e dados são válidas
WREADY	*Slave*	Indica que o *Slave* está pronto para aceitar os sinais de controlo e os dados para escrita
WDATA	*Master*	Dados escritos para o *Slave* AXI
WSTRB	*Master*	Sinalizador dos *bytes* válidos nos dados escritos (WDATA)
BVALID	*Slave*	Indica que o sinal de BRESP é válido
BREADY	*Master*	*Master* está pronto para receber o sinal de BRESP
BRESP	*Slave*	Indica o estado da transação de escrita (Sucesso ou Insucesso)
RVALID	*Slave*	Indica que o *Slave* está pronto para enviar os dados para leitura
RREADY	*Master*	Indica que o *Master* está pronto para receber os dados para
RDATA	*Slave*	Dados para leitura
RRESP	*Slave*	Indica estado da transferência de dados (Sucesso ou Insucesso)

Em termos de protocolo existem três fases distintas (AR ou R – operações de leitura, AW ou W – operações de escrita):

- *Handshake* – São assertados os valores de controlo necessários para a realização da transferência por parte do *master* e assim que os sinais tomarem os valores desejados é colocado a *high* o sinal AR/AW-VALID. O *slave* acaba a fase colocando o sinal AR/AW-READY a *high*, dando início à transferência de dados;

- Transferência de Dados – A transferência de dados ocorre quando ambos, o *slave* e o *master* verificam compatibilidade nos valores dos sinais W/R-VALID e W/R-READY a *high*. Exclusivamente nesta interface, estes sinais são realizados no *handshake*, pois apenas existe uma transferência de uma trama de dados nesta transação;

- *Acknowledge* – Após realizada a transferência, o sinal BRESP é levado a *low* para uma resposta de sucesso de transferência (após devido "*handshake*" de BVALID e BREADY);

3.3 GlobalPlatform API

A GlobalPlatform é líder de mercado no desenvolvimento e manutenção de especificações para TEEs. Possui um portfólio de *Application Programming Interfaces* (APIs) considerável, que permitem a compatibilidade do código fonte entre diferentes TEEs. A necessidade em criar APIs para TEEs surgiu devido à fragmentação no mercado a nível de coerência de APIs de segurança, o que, no ponto de vista da GlobalPlatform é um dos principais problemas na segurança. Contudo, a própria ARM foi pioneira no desenvolvimento deste tipo de APIs através da TZAPI, que surgiu aquando da introdução da tecnologia TrustZone no mercado.

A especificações definem os conceitos de aplicação cliente (CA) e serviço seguro (TA). A CA executa num ambiente externo ao TEE que não possui garantias de segurança (REE) e a TA é executada no próprio TEE. Para comunicar com a TA, a CA recorre à *Client* API, existente no REE.

3.3.1 TEE *Client* API

A TEE *Client* API (TEECAPI) define um conjunto de funções que fornecem uma comunicação eficiente de aplicações clientes com serviços seguros. Sobre esta especificação podem ser implementadas funcionalidades de mais alto nível, como por exemplo, a instalação de novas TAs em *run-time*. Uma CA, usa a TEECAPI para estabelecer comunicação com o TEE, estabelecer uma sessão com a respetiva TA, configurar a memória partilhada na comunicação e enviar à TA

comandos específicos para invocar um serviço seguro ou fechar silenciosamente a comunicação.

3.4 Plataforma de Desenvolvimento

Na secção que se segue é introduzida a plataforma de desenvolvimento (secção 3.4.1), indispensável para o desenvolvimento da dissertação.

3.4.1 Zynq-7000 *All Programmable* SoC

Todos os dispositivos da família Xilinx Zynq-7000 são baseados na arquitetura *All Programmable SoC* (AP SoC). A arquitetura possui, tal como se observa na Figura 3.2, o *Processing System* (PS) baseada num processador *dual* ou *single core(s)* ARM Cortex-A9, assim como a *Programmable Logic* (PL) baseada na *Field-Programmable Gate Array* (FPGA) de baixo consumo Xilinx Artix-7 Series. A Zynq-7000 permite aos utilizadores de sistemas tradicionais *Application-Specific Integrated Circuit* (ASICS) acompanharem as exigências atuais de programação, pois combina num único dispositivo a versatilidade associada às FPGAs com o poder de processamento de um processador *standard*, *onboard hard-wired*. A heterogeneidade da arquitetura é adequada a aplicações únicas de *hardware-software co-design*, como *high-performance computing* (i.e., processamento de imagem em visão por computador). A Xilinx, juntamente com seus parceiros, fornecem suporte através de uma vasta oferta de *soft Intlectual Properties* (IPs), assim como *device drivers* para os periféricos da PL e PS, compatíveis com Linux ou para aplicações *bare-metal*. Disponibilizam também, um conjunto de ambientes de desenvolvimento para o rápido desenvolvimento de *software* e *hardware*.

A *Application Processor Unit* (APU), encontra-se na PS e contém essencialmente o CPU e periféricos, como por exemplo, memórias *caches* L1 e L2, MMU, NEON e CoreSight. O(s) processador(es) da PS são responsáveis pela inicialização de toda a plataforma e por programar a FPGA da PL. A PS engloba também múltiplas interfaces para conexão com os aceleradores em *hardware* na lógica da PL e o componente que permite essa interligação é o *Interconnect*. Os portos de interfaces de comunicação, usam o protocolo AXI e existem disponíveis: 1) quatro portos de alta performance (4xHP), 2) quatro portos de propósito geral, dois *masters* e dois *slaves* (2xMGP, 2xSGP); 3) um porto *Accelerator Coherency Port* (ACP), que se destaca pela menor latência de acesso às memórias internas do CPU e pode ser usado para coerência entre as *caches* e a PL.

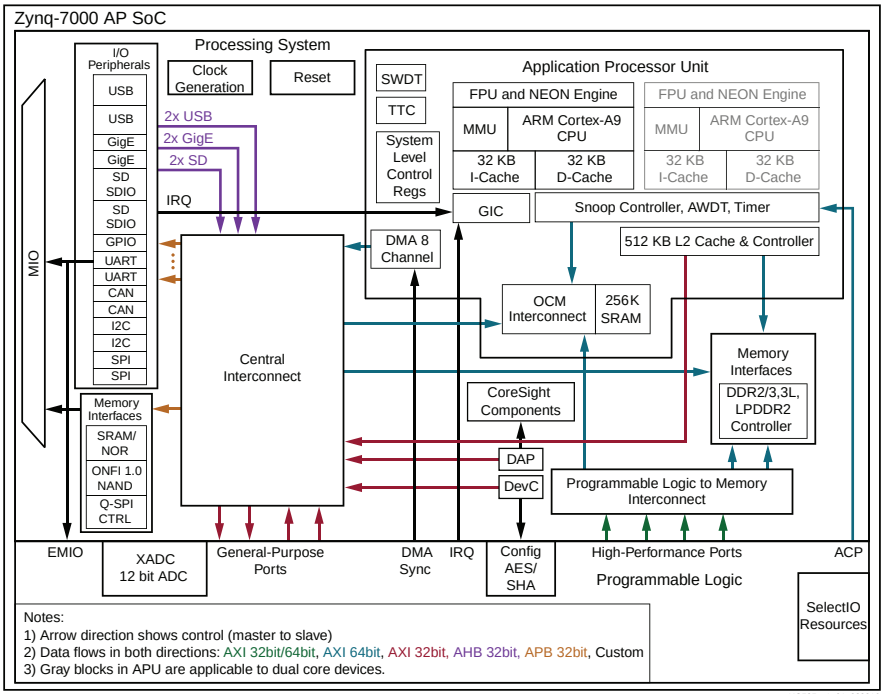


Figura 3.2: Zynq-7000 *All Programmable SoC*

A escolha da plataforma Zynq-7000 para o desenvolvimento da dissertação deveu-se a diversos motivos: 1) a Zynq-7000 é uma das poucas plataformas que possui a especificação da sua implementação da TrustZone disponível e acessível aos desenvolvedores; 2) incorpora nos processadores a tecnologia de segurança TrustZone; 3) é uma plataforma híbrida, onde a sua FPGA pode ser explorada para a criação de periféricos; 4) utiliza o protocolo AXI, tanto nos barramentos internos da APU como para a interface com a PL, significando que a coerência a nível de segurança é estendida a toda a totalidade da arquitetura; 5) a *toolchain* disponibilizada pelo fabricante para o desenvolvimento de *hardware* na PL, permite o a criação de periféricos TrustZone-*Aware*, concedendo ao *designer* total liberdade para o desenvolvimento de sistemas heterogéneos seguros.

3.5 *Toolchain* de Desenvolvimento

As ferramentas utilizadas para o desenvolvimento da dissertação são abordadas nesta secção. O Vivado Design Suite e o SDK da Xilinx, ambas criadas pela Xilinx, são dadas a conhecer nas subsecções 3.5.1 e 3.5.2. A presente secção é finalizada com a ferramenta ARM Fast Models .

3.5.1 Vivado Design Suite

O Vivado Design Suite é um conjunto de ferramentas que funcionam de forma integrada para aumentar a produtividade durante as fases de *design*, integração e implementação de um sistema na arquitetura Zynq-7000 AP SoC. O Vivado foi criado para lidar com os desafios de desenvolvimento impostos pela complexidade atual dos dispositivos Xilinx.

As ferramentas complementares do obsoleto Xilinx ISE Design Suite, foram agrupadas no Vivado. O seu ambiente gráfico, é na verdade uma evolução do PlanAhead. É intuitivo e de simples aprendizagem a novos utilizadores, contudo oferece todo o poder de personalização a utilizadores mais avançados. O Manage IP foi outra das ferramentas importadas para o Vivado, conhecido no ISE como CoreGenerator. Concede um ambiente para a criação personalizável de IP *cores*, onde posteriormente e de forma automática é gerada a implementação do *hardware*, também disponível ao utilizador. O IP Integrator é a principal área de interação com Vivado. Através de intuitivas interfaces, o utilizador é capaz de adicionar uma vasta gama de IP *cores*, realizar a sua personalização individual, assim como a interligação entre todos os blocos de forma automática.

As ferramentas do Vivado aceleram a implementação de um *design* e otimizam-no segundo diversas métricas, em cada fase de desenvolvimento, por forma a reduzir iterações e acelerar o *time-to-market*.

3.5.2 Xilinx SDK

O Xilinx *Software Development Kit* (SDK) é um *Integrated Development Environment* (IDE) baseado no *standard open-source* da Eclipse. É, no entanto, específico para o desenvolvimento de *software* nos processadores da plataformas Xilinx. O XSDK possui um editor de código C/C++ rico e com várias ferramentas de suporte para um desenvolvimento simples e automatizado do *software*, podendo este processo ser realizado de forma paralela com o desenvolvimento de *hardware* no Vivado.

O XSDK está adaptado para usar os *designs* em *hardware* criados pelo IP Integrator do Vivado, para poder programar tanto a FPGA como os processadores da plataforma. No Vivado encontra-se refletida informação configuracional relativa a ambas as partes da plataforma (PS-PL). O XSDK adquire os ficheiros relativos à pré-inicialização da PS, ao processador alvo e periféricos mapeados no *hardware*, e ainda, um ficheiro usado para programar toda a FPGA. O XSDK recorre a vários *plug-ins* para programar o(s) processador(es) e FPGA, permitindo a sua

programação independente através de USB ou, todas as partes da plataforma através de um cartão de memória externo.

3.5.3 ARM Fast Models

ARM Fast Models vem descartar a necessidade de uma plataforma física durante o desenvolvimento de um sistema computacional. Esta ferramenta facilita o acesso às plataformas que por vezes são inexistentes ou demasiado dispendiosas, oferecendo plataformas virtuais totalmente validadas. Na sua base encontram-se modelos de processadores e periféricos que simulam o comportamento desse *hardware*, contudo apenas do ponto de vista do programador. O comportamento do *software* que se obtém nestes modelos é equivalente ao que um programador vislumbraria no *hardware* físico. O Fast Models simula de forma otimizada a execução do comportamento do *software*, no entanto, em prol desta característica, a ferramenta não é capaz de determinar a performance do *software*, por não conceder quaisquer garantias sobre as interações de baixo nível do *hardware*, como ciclos de relógio.

O Fast Models oferece um portfólio de modelos de alta performance dos mais recentes IPs da ARM (CPUs e periféricos). Os blocos permitem a construção de plataformas virtuais, ou utilização de plataformas virtuais já prontas, como é o caso da *Virtual Express*. As características mais relevantes do ARM Fast Models são: 1) modelos de ISAs da ARM funcionalmente precisos e validados; 2) modela tecnologias avançadas da ARM, como *caches*, MMU, virtualização, VFP e TrustZone; 3) inicializa qualquer SO, incluindo Linux, Android e Windows em segundos.

A ferramenta System Canvas reduz o tempo de *design* de plataformas virtuais, devido à sua intuitiva interface baseada em blocos, que permite a um programador inexperiente realizar o *design*, configuração e construção de plataformas virtuais. As plataformas são depois transformadas em modelos de simulação pelo System Generator. Este incorpora as interfaces necessárias para permitir a conexão de *debuggers standards*. Fazendo uso das capacidades de *debug* e *trace* existentes nas plataformas virtuais, o Model Debugger do Fast Models, acede a essas capacidades e permite a inspeção de qualquer componente da plataforma virtual. Para o desenvolvimento de uma aplicação o Model Debugger inclui: 1) funcionalidades *standard* de depuração, *step-by-step*, *step-in*, *step-out*, *step-over* e *breakpoints* nas intruções; 2) *disassembley* do código e visão sobre o estado dos registo e memória; 3) janela de *watch* do estado das variáveis locais e globais da aplicação; 4) suporta a depuração em ambientes *multicore*.

Implementação do L-TEE

No presente capítulo é abordado a implementação do L-TEE, *framework* em *software* que implementa o TEE. Numa primeira abordagem é dado a conhecer ao leitor as escolhas efetuadas para cada um dos componentes do L-TEE, secção 4.1. Após esta fase, a implementação é descrita de forma funcional nas secções 4.2, 4.3, 4.4, 4.5, 4.6 e 4.7. A divisão foi realizada em vários requisitos que, na ótica do autor, são importantes para a segurança e correto funcionamento do TEE. Em cada tópico, são também apresentados alguns detalhes técnicos de implementação.

4.1 Visão Geral do L-TEE

Na presente secção irão ser descritos todos os componentes do L-TEE. A *framework* é uma adaptação do LTZVisor (secção 2.2.7.1), orientada às necessidades da dissertação, isto é, o desenvolvimento de um TEE e suas funcionalidades de comunicação, por forma a tornar-se um sistema oportuno para o desenvolvimento e integração de um periférico de segurança monitorizador.

O L-TEE é um sistema *dual-OS single core*, que permite a coexistência de dois SOs distintos sobre a mesma plataforma. Neste caso, cada SO representa um ambiente de execução com características de segurança distintas - TEE e REE, proporcionado pelas extensões de segurança. Para além do isolamento, a tecnologia TrustZone concede as bases para construção do canal de comunicação, permitindo que aplicações cliente no REE comuniquem com serviços seguros, exclusivos do TEE.

A Figura 4.1 apresenta os componentes que compõem o L-TEE, assim como o COMON, isto é, o periférico que se pretende desenvolver. Conforme se pode ver na figura, à direita observa-se no mundo não-seguro, o REE, este é composto maioritariamente por um GPOS (Linux). Em *user mode* do Linux irão executar aplicações clientes (NSCApps) capazes de usufruir dos recursos do TEE. A especificação TEECAPI da GlobalPlatform existe na forma de uma biblioteca (*library*) que se encontra igualmente no modo não privilegiado. Esta tira partido das funcionalidades de baixo nível, fornecidas por um *device driver*, que implementa o canal de comunicação propriamente dito. O TEE encontra-se à direita da

Figura 4.1 e é composto por um *trusted* RTOS (T-RTOS) e pelo Monitor, este último, apenas responsável por gerir coerentemente a execução dos SOs. O T-RTOS (FreeRTOS) possui serviços seguros (SSvcs) específicos de aplicações clientes, responsáveis por fornecer funcionalidades sensíveis na qual estas não possuem permissões de acesso. Para finalizar, acoplado ao processador observa-se o COMON que é responsável por monitorizar a comunicação que ocorre no mundo seguro.

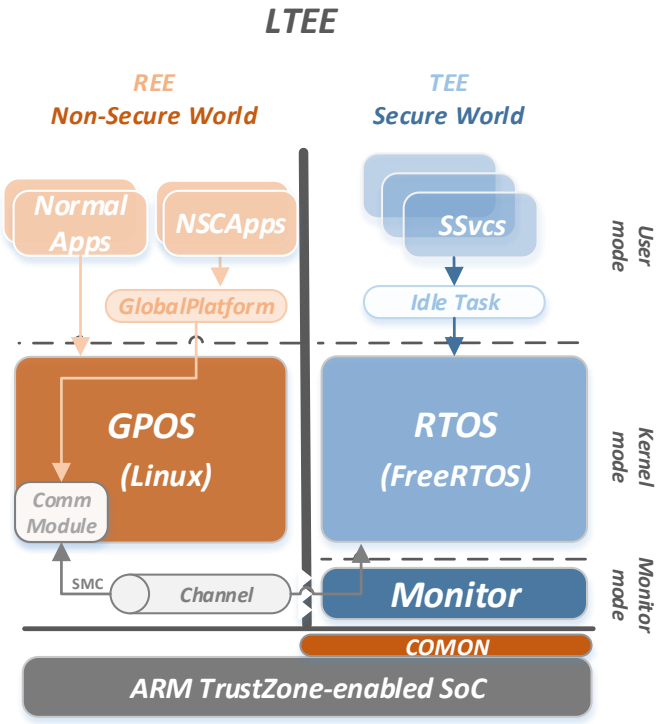


Figura 4.1: L-TEE: Arquitetura genérica

4.1.1 Mundo Seguro

O mundo seguro, na presente dissertação, compreende vários componentes, sendo estes o T-RTOS, o Monitor e o COMON. Estes são descritos respetivamente nas secções 4.1.1.1, 4.1.1.2 e 4.1.1.3.

4.1.1.1 T-RTOS

No mundo seguro, como recomendado pela ARM, deve ser existir um SO minimalista e robusto, pois este representará grande parte da TCB do sistema. Um RTOS enquadra-se nestes requisitos, pois possuem um baixo *footprint* de memória e a tarefa de encontrar vulnerabilidades na sua codificação pode ser realizada pelo próprio programador de forma manual. O FreeRTOS foi eleito, contudo nesta configuração não é utilizado para funcionar como um RTOS puro, de maneira a

provar o conceito, este componente foi modificado para fornecer funcionalidades multi-tarefa e de gestão de memória, por forma a fornecer os serviços mínimos de um *kernel* seguro.

4.1.1.2 Monitor

O Monitor executa em modo monitor e é a porta de entrada e interface para o mundo seguro e para o mundo não-seguro. É o componente que permite a gestão entre mundos, devendo preservar e restaurar o estado dos SOs quando ocorre a troca entre domínios, assim como, transmitir os pedidos para comunicação por parte SO do mundo seguro. Esta camada deverá ser simples por dois motivos: o primeiro prende-se com performance na disponibilização das funcionalidades seguras do T-RTOS. O Monitor deve ser projetado de forma a minimizar o *overhead* que existirá na comutação entre mundos e o segundo motivo, deve-se ao caráter inseguro do GPOS, que obriga a que esta camada seja minimalista, por forma a possuir o menor número de *bugs*.

4.1.1.3 COMON

O COMON é um periférico de monitorização que irá ser desenvolvido com o intuito de aumentar a segurança em torno do canal de comunicação da tecnologia TrutZone, que *by design* se encontra totalmente desprotegido. A monitorização permitirá ao T-RTOS detetar intrusões originárias do REE, que ambicionam utilizar os SSvcs no TEE com a finalidade de aceder aos recursos críticos da TrustZone. Este componente será detentor de uma interface AXI pois é a única interface na qual periféricos na PL se podem acoplar aos processadores na PS da Zynq. Além disso, as ferramentas de desenvolvimento estão otimizadas para a geração de periféricos com este tipo de interface.

4.1.2 GPOS

O mundo não-seguro é apropriado à execução de um GPOS tradicional (i.e., Linux) pois a sua complexidade (milhões de linhas de código), confere-lhes um carácter inseguro. Em contrapartida, são ideais para aplicações dependentes da Internet e proporcionam uma evoluída interface, ideal para interações homem-máquina. A escolha recaiu no Linux, pois é *open source* e possui bastante suporte ao nível de documentação e *software* por parte do fabricante da plataforma de desenvolvimento, a Xilinx.

Em *user mode*, executam NSCApps que comunicarão com SSvcs no TEE, para a proporcionar ao Linux funcionalidades que este não tem permissão para aceder diretamente. As aplicações deverão por isso, ser implementadas de forma a estarem cientes da existência de uma API de comunicação, TEECAPI. Esta, na forma de uma biblioteca, executará em *userspace* e terá a responsabilidade de criar um canal de comunicação lógico entre as NSCApps do Linux e os SSvcs do T-RTOS. Para isso, a TEECAPI deverá tirar partido de um *device driver* alojado em modo *kernel* do Linux.

4.1.3 Comunicação entre Mundos

O REE é considerado inseguro por não oferecer as garantias de segurança exigidas, contudo este pode usufruir de funcionalidades que apenas o TEE pode realizar e oferecer. Uma determinada funcionalidade sobre um recurso seguro é realizada em virtude da NSCApp que requisitou essa ação. A interação entre os distintos *end-points* da comunicação é realizada num estilo *remote procedure call* (RPC): Interfaces, definidas e seguras, fornecem o contacto e forma de invocar uma funcionalidade específica ao serviço, isto é, a NSCApp tem de esperar pela resposta do SSvc, enquanto são realizadas as operações necessárias (estilo *blocking*).

A Figura 4.2, apresenta os componentes intervenientes na comunicação. Esta é sempre iniciada no mundo seguro por iniciativa da NSCApp, e resumidamente aloca memória não-segura, que será o *container* dos dados a transmitir. De seguida, a aplicação conecta-se ao *device* especifico da comunicação, que é responsável por realizar a manutenção do contexto da comunicação com o TEE, assim como, por invocar a exceção (SMC) quando todas as operações *low-level* estiverem concluídas. Neste ponto, o pedido é reencaminhado ao componente Monitor que por sua vez o transfere ao T-RTOS. O T-RTOS executa o SSvc desejado e após o seu término, é responsável por reenviar novamente o pedido de volta ao REE, realizando o processo inverso.

4.2 Gestão temporal e de interrupções

O LTZVisor implementa um escalonamento assimétrico ou *idle*, devido à dificuldade que é inerente à virtualização de um ambiente de tempo-real. A execução da *gest* não-segura é concedida durante períodos *idle* da *gest* segura e sempre que esta última deseje, tem o poder de recuperar o controlo de execução. O L-TEE não possui os problemas associados à virtualização e, como tal, não implementa a mesma política de escalonamento. O T-RTOS possui um comportamento passivo

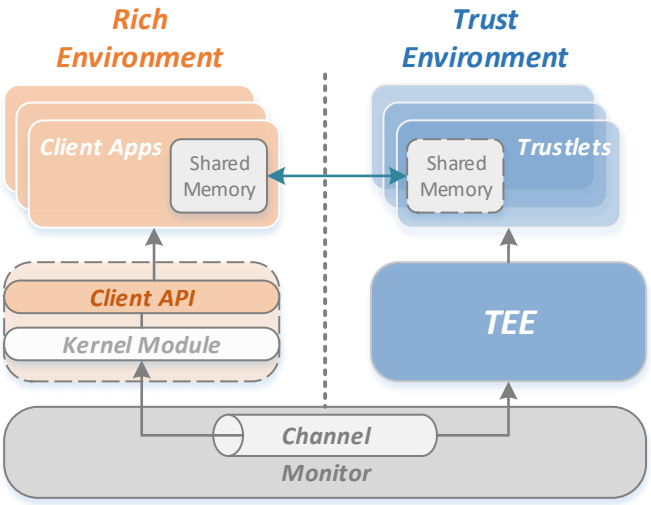


Figura 4.2: *Path* de comunicação

(pois não tem que assegurar execução determinística), sendo apenas executado quando um cliente no GPOS (REE) requer os seus serviços.

A política de escalonamento no LTZVisor levou à utilização de duas unidades de temporização, por forma a conceder isolamento temporal. No L-TEE, apenas é utilizado um *timer* (*Triple Timer Counter* - TTC1) pois o isolamento temporal não é crítico. O TTC1 é disponibilizado ao Linux, e é dedicado ao seu escalonador. O *timer* deve ser configurado no mundo seguro pelo T-RTOS, como acessível pelo mundo não-seguro, pois caso contrário, no primeiro acesso por parte do GPOS é gerada um exceção e o sistema para.

Nos SoCs habilitados com tecnologia TrustZone, os *Generic Interrupt Controllers* (GICs) suportam a coexistência de fontes de interrupções seguras e não-seguras. No L-TEE as interrupções FIQs são configuradas como seguras e as IRQs como não-seguras. A segurança de cada interrupção pode ser individualmente configurada através do conjunto de *Interrupt Security Registers* (ICDISRx). As FIQs podem ser usadas para interromper o L-TEE para realização de tarefas como *input* seguro ou para alerta de potenciais ameaças. Quando o T-RTOS tem o controlo de execução e é gerada uma exceção FIQ, esta é atendida pelo próprio SO sem intervenção do Monitor (o *bit* FIQ do SCR tem de estar inativo 3.1.2.1). Caso, ocorra uma IRQ, o T-RTOS não é interrompido, mas imediatamente após o GPOS retomar o controlo, a interrupção é atendida. Quando é o GPOS que se encontra na posse da execução e por sua vez ocorre uma FIQ, o fluxo é redirecionado para o Monitor que por sua vez o transfere ao T-RTOS. Se a exceção se tratar de uma IRQ então esta é atendida pelo próprio GPOS.

4.3 Proteção da memória

O TZASC permite particionar a memória em regiões seguras e não-seguras. Esta característica é explorada para conceder isolamento espacial entre o REE e o TEE. Ao GPOS é atribuída memória não-segura e ao T-RTOS e Monitor uma região segura. Assim sendo, um acesso por parte do GPOS a um endereço de memória segura desencadeia uma exceção.

Na plataforma Zynq, as regiões são definidas com uma granularidade de 64MB, através de um registo de controlo do sistema (TZ_DDR_RAM). Cada *bit* representa uma região incremental, cujo seu estado, depende do seu valor (0 - seguro, 1 - não-seguro). No L-TEE são configurados como não-seguros 256MB, e atribuídos ao GPOS e 64MB são configurados como seguros e atribuídos aos componentes do TEE. A plataforma disponibiliza 1GB de memória RAM, pelo que os restantes *bits* excedentes a este limite não possuem qualquer efeito. Os restantes 680MB ainda disponíveis não foram utilizados. Na Figura 4.3, observa-se a disposição da memória e componentes pertencentes à *framework*.

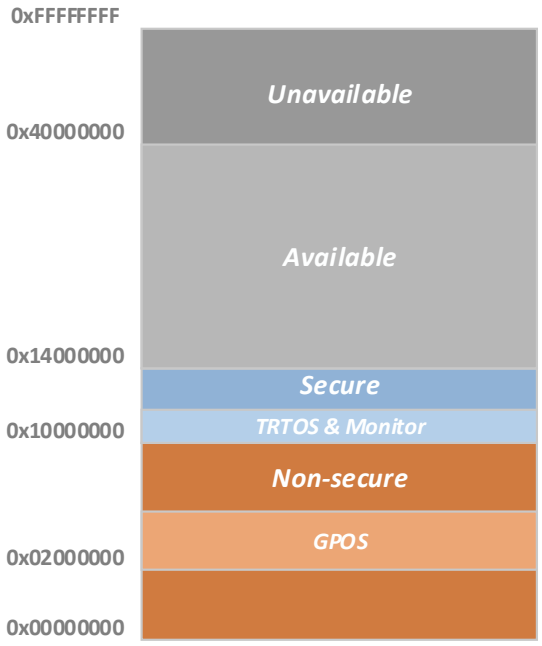


Figura 4.3: Configuração da memória no LTEE

4.4 Proteção dos dispositivos

A tecnologia TrustZone permite a configuração de um elevado número de periféricos existentes no SoC como seguros ou não-seguros, o que contribui também para o isolamento espacial.

No L-TEE os periféricos não-seguros são associados ao GPOS e os seguros ao T-RTOS, pelo que acesso ilícito a um periférico seguro gera automaticamente uma exceção. Na Zynq, o estado de segurança dos dispositivos é configurável através de um conjunto de registos apenas acessíveis no mundo seguro. Os registos security_apb e security_fssw_s0 pertencentes ao *Global Programmers View* (GPV) são responsáveis respetivamente por configurar o TTC1 como acessível ao mundo não-seguro e por garantir que apenas transações seguras são transmitidas até aos periféricos na PL que se encontrem conectados à interface *master* AXIGP0.

4.5 Comutação entre ambientes

Um CPU físico é dividido em dois virtuais na tecnologia TrustZone (secção 3.1.2.1), devido ao banqueamento de um vasto conjunto de registos (i.e., coprocessadores). Visto que cada SO executa num domínio distinto, a comutação entre ambos é realizada com mínimo de esforço devido ao suporte concedido pelo *hardware*. Outra característica que aumenta a performance é a existência de um NS-*bit* por cada entrada da *cache*. Esta última aumenta também a segurança do sistema perante ataques por *software* que exploram a cache.

No L-TEE, o contexto associado ao GPOS é composto por 30 registos: 13 de propósito geral, spsr, lr e sp dos modos *system*, *abort*, IRQ, *undef* e *user* e ainda o lr e o spsr do modo *monitor*. O contexto do T-RTOS está otimizado e requer apenas 16 registos: 13 de propósito geral, spsr, lr e sp do modo *system*.

4.6 Proteção da infraestrutura de *debug*

A tecnologia TrustZone permite definir o controlo de acesso ao *debug* do CPU, podendo ser desabilitado para o mundo seguro e habilitado ao mundo não-seguro, vice-versa, ou ainda em nenhum. A visibilidade é controlada por quatro sinais, DBGEN, NIDEN, SPIDEN e SPNIDEN, abordados na secção 3.1.2.1 cuja configurabilidade depende do próprio fabricante. Esta característica pode ser utilizada para proteger o sistema contra ataques físicos que se aproveitam da infraestrutura de depuração.

Na plataforma de desenvolvimento, os sinais são controlados através de um registo de controlo devcfg.ctr pertencente à *Device Configuration Interface* (DevC). No L-TEE, o *debug* encontra-se desativado para o mundo seguro (SPIDEN e SPNIDEN estão a 0), evitando qualquer tentativa de monitorização ou interferência na execução que ocorre no mundo seguro. No que toca ao mundo não-seguro, o

debug encontra-se ativado (DBGEN e NIDEN estão a 1) de forma a flexibilizar o processo de desenvolvimento de aplicações no Linux.

4.7 Comunicação

As funcionalidades de comunicação são uma característica própria na natureza de um TEE, por isso, foram desenvolvidas e integradas na *framework*. O suporte concedido no TEE é abordado na secção 4.7.1, seguidamente é explicada na secção 4.7.2 as funcionalidades que fornecem a comunicação no REE.

4.7.1 TEE

Um TEE disponibiliza aos seus SSvcs o meio para comunicarem com NSCApps do REE. O Monitor é o *gatekeeper* do mundo seguro e juntamente com a instrução SMC formam a base para a construção do canal de comunicação entre mundos. No Monitor, o suporte à comunicação é realizado através de duas rotinas *tz_api_call* e *ret_from_secure_api* (Lista 4.1), implementadas no SMC *handler* do modo monitor. Estas lidam respetivamente com os pedidos vindos do REE e com as respostas a esses mesmos pedidos vindos do TEE.

Lista 4.1: Suporte à comunicação no Monitor

```
1  .globl _mon_smc_handler
2  _mon_smc_handler:
3
4      ; Call TrustZone API
5      cmp    r0, #CALL_TRUSTZONE_API
6      beq    tz_api_call
7
8      ; Return TrustZone API
9      cmp    r0, #RET_SECURE_API
10     beq    ret_from_secure_api
11
12     ; (...)
```

No T-RTOS, as funcionalidades nativas para a criação de tarefas foram exploradas para a criação de SSvcs. São criados com uma prioridade superior à tarefa *idle* (*tskIDLE_PRIORITY + 1*) e recebem como parâmetro um apontador para uma estrutura que representa o pedido da NSCApp (*param*). Este parâmetro foi codificado no *kernel* do REE por questões de performance. Na Lista 4.2, é apresentado a criação do serviço seguro ECHO (ECHO_SS), que sucintamente recebe e envia de volta os dados da ECHO_NSCApp. Os parâmetros restantes à

criação do ECHO_SS são, o apontador para o *entry-point* da função (*prvEchoSS*) que implementa do serviço, um nome descritivo da função (*echo_ss*) usado para facilitar o processo de depuração e o tamanho da *stack* que devera ser alocado para uso do serviço (*configMINIMAL_STACK_SIZE+100*).

Lista 4.2: Criação de um serviço seguro

```
1  int ftz_ss_dispatcher(void *param)
2  {
3     /* {...} */
4     struct otz_smc_cmd *cmd = NULL;
5     cmd = (struct otz_smc_cmd*)param;
6
7     switch(cmd->context)
8     {
9        case ECHO_SS_TASK_ID:
10       {
11          xTaskCreate( prvEchoSS, (signed char *) "echo_ss",
                configMINIMAL_STACK_SIZE+100, (void *)param, (
                tskIDLE_PRIORITY + 1), NULL );
12          break;
13       }
14    /* {...} */
15    }
16    /* {...} */
17 }
```

No corpo da tarefa *idle* (*prvIdleTask*) encontra-se o suporte *back-end* à TEE *Client* API. Foram implementadas rotinas de abertura e fecho de sessão da comunicação (*open_session_from_ns()* e *close_session_from_ns()*), bem como uma responsável pela criação do SSvc associado à sessão (*ftz_ss_dispatcher()*). Por motivos de *time-constraint*, não foi desenvolvida uma especificação interna ao TEE dedicada à gestão dos serviços seguros (TEE *Internal* API).

4.7.2 REE

O suporte à comunicação no REE foi divido em duas secções, uma que aborda a biblioteca TEECAPI (secção 4.7.2.1) da GlobalPlatform, e uma secção 4.7.2.2 que abrange a implementação de um *kernel module*.

4.7.2.1 GlobalPlatform TEE *Client* API

A GlobalPlatform é responsável por desenvolver uma série de especificações para TEEs. A TEE Client API (secção 3.3.1), define um conjunto de funções que

uma NSCApp no REE pode utilizar para comunicar de forma segura e eficiente com um SSvc no TEE. No L-TEE a API foi incorporada ao GPOS através de uma biblioteca em *user space* que recorre às funcionalidades de baixo nível de um *kernel module*.

Tabela 4.1: Funções TEECAPI

Função	Descrição
TEEC_InitializeContext	Inicia um contexto para a comunicação com um TEE específico
TEEC_FinalizeContext	Fecha o contexto e limpa todos os recursos associados ao mesmo
TEEC_RegisterSharedMemory	Regista um bloco de memória da CA para comunicação
TEEC_AllocateSharedMemory	Aloca um bloco de memória em virtude da CA para a comunicação
TEEC_ReleaseSharedMemory	Liberta todos os recursos associados com a memória partilhada
TEEC_OpenSession	Abre uma sessão com uma TA identificado com UUID
TEEC_CloseSession	Fecha a sessão e liberta os recursos associados
TEEC_InvokeCommand	Responsável por invocar comandos remotos à TA

As funções implementadas encontram-se resumidas na Tabela 4.1. Uma qualquer NSCApp que deseje usufruir de uma funcionalidade associada a um SSvc, devera primeiramente estabelecer um contexto com o TEE. A TEEC_InitializeContext() é responsável por estabelecer o contexto, que recorrendo ao *kernel module* retorna, em caso de sucesso, um identificador do TEE. Após o contexto estar estabelecido, procede-se à abertura de uma sessão de comunicação entre a NSCApp e o SSvc. O UUID deve ser passado à função responsável por este processo, TEEC_OpenSession(). Em caso de sucesso é retornado à NSCApp um identificador da sessão, significando que a comunicação se encontra estabelecida. De seguida é necessário alocar um bloco de memória à comunicação para troca de mensagens. Para isto a NSCApp recorre à função TEEC_RegisterSharedMemory() ou TEEC_AllocateSharedMemory(), cuja diferença incide no facto de a memória registada para comunicação pertencer à própria NSCApp ou se a memória é alocada em virtude desta, mais uma vez recorrendo ao *kernel module*. Estando a memória entre os dois intervenientes preparada a NSCApp pode invocar comandos ao SSvc, recorrendo à função TEEC_InvokeCommand(). O SSvc responde ao pedido colocando as respostas na bloco partilhado. Quando não existirem mais pedidos para serem enviados, a NSCApp deve fechar a sessão, que consiste na utilização da função TEEC_ReleaseMemory() para libertar os recursos de memória usados e por fim, fechar a comunicação recorrendo à função TEEC_CloseSession().

4.7.2.2 *Device driver* TrustZone

O *device driver* ou modulo do *kernel* interpreta as *system calls* das NSCApps e estabelece a comunicação com o TEE. No GPOS foi integrado, devido a *time-constraint*, um um *device driver* desenvolvido e disponibilizado pela OpenVirtualization. Este possui uma complexidade considerável e é especifico para plataformas virtuais (VE). Por esta razão , foram realizadas modificações com respeito à manutenção da coerência da memória física na Zynq. Recorreu-se para esse fim, à função ___cpuc_flush_dcache_area() disponível no Linux.

O *driver* concede a ilusão que a comunicação apenas possui dois intervenientes, a NSCApp e o SSvc (comunicação lógica), abstraindo a verdadeira complexidade associada à comunicação que requer cuidado na interpretação de IOCTLs para: 1) abertura e fecho de sessões (i.e., FTZ_CLIENT_IOCTL_SES_OPEN_REQ); 2) alocação de memória (i.e., FTZ_CLIENT_IOCTL_ENC_MEM_REF); 3) codificação e descodificação de pedidos (i.e., FTZ_CLIENT_IOCTL_ENC_UINT32); e por fim, 4) estabelecimento da comunicação propriamente dita (envio de comandos), através da instrução SMC.

Metodologia e Implementação do COMON

A presente dissertação, necessitou de uma investigação que proporcionou a idealização e conceção da solução de segurança. No presente capítulo, é inicialmente definido, na secção 5.1, o espectro de segurança da tecnologia TrustZone, posteriormente serão apresentados os motivos que, na ótica do autor, permitem a existência de uma solução de segurança baseada em monitorização (secção 5.2). De seguida, na secção 5.3, a característica não intrusiva da solução é justificada como sendo um desafio arquitetural e, perto do termino do capítulo, é explicado a investigação propriamente dita (secção 5.4), necessária para conceber a solução de segurança. Por fim, na secção 5.5, é abordado a implementação do periférico de segurança.

5.1 Espectro de Segurança

Um dos principais objetivos da dissertação é estender o espectro da tecnologia alvo. É fundamental definir o espectro de segurança, de maneira a melhor identificar as vulnerabilidades presentes na tecnologia TrustZone e, desta forma, compreender como será possível complementar a camada de proteção concedida por esta.

Os ataques físicos, regra geral, não justificam o valor de retorno por serem excessivamente dispendiosos (secção 2.2.3.2). No entanto, os tipos de ataque que recorrem à infraestrutura de *debug* são comuns e fora do leque de vulnerabilidades da tecnologia. Os ataques *side-channels*, destacam-se pela pouca complexidade na sua realização, sendo a tecnologia vítima em trabalhos como [17, 18], que após análise de propriedades da *cache*, permitiu ao atacante prever o valor de uma chave criptográfica alojada no domínio seguro. Existe uma enorme variedade de ataques *side-channel* atualmente, mas na ótica do autor, a tecnologia não concede nenhum mecanismo específico de proteção. Contudo, as características inovadoras da plataforma Zynq e o seu suporte à tecnologia TrustZone em todo o sistema, possibilita o desenvolvimento de mecanismos capazes de minimizar o impacto deste tipo

de ataques. Relativamente aos ataques *Trojan*, tanto quanto é do conhecimento do autor não existe, igualmente aos ataques *side-channel*, qualquer mecanismo de proteção contra estes, por parte da tecnologia TrustZone.

A investigação debruçou-se mais na compreensão dos ataques baseados em *software*, visto existir um considerável número de trabalhos de diferentes analistas nesta área, que com sucesso derrubaram toda a proteção concedida pela tecnologia. Dan Rosenberg, com duas publicações em 2013 e 2014 [19, 20], foi o pioneiro na divulgação destes ataques. O seu primeiro ataque resumidamente, deveu-se à descoberta de um *bug* no *kernel* seguro da Motorola, comercializado em dispositivos como o Atrix HD, Razr HD e Razr M. O ataque resultou no desbloqueio do *bootloader* e, consequentemente, permitiu a instalação de *kernels* customizados nestes dispositivos. Recorrendo à engenharia reversa e ferramentas adequadas, o atacante concluiu que não necessitava de quebrar a robusta cadeia de *boot* para realizar o *unlocking* pois, a própria implementação do kernel seguro era a responsável por aceder diretamente aos QFuses (fusível Qualcom), componentes com a capacidade de desbloqueio. Posto isto, o atacante detetou um *bug* (*zero write-primitive*) na implementação do *kernel* seguro, sendo esta suficiente para, através de um SMC manipulado, desbloquear o dispositivo.

O segundo trabalho de Dan Rosenberg, incidiu sobre a implementação do TEE da Qualcom (QSEE). Os dispositivos afetados foram: LG Nexus 4, LG Nexus 5, LG G2, HTC ONE Series, Moto X, Samsung Galaxy S4 e Samsung Galaxy Note 3. A natureza do ataque foi um *integer overflow* que, em última instância permitiu a execução de código maligno no mundo seguro da tecnologia TrustZone. A construção do ataque deveu-se a vários *bugs* encontrados, todos pertencentes ao QSEE. Para além do *bug* acima mencionado, um outro, neste caso uma *write-primitive*, permitiu a invocação de SMCs manipuladas pelo atacante, capazes de modificar a memória segura de forma limitada. Por fim, a descoberta de outra vulnerabilidade, neste caso arquitetural, foi a chave para o sucesso de ataque: de maneira a identificar o serviço seguro referenciado no pedido SMC, o QSEE percorre uma lista onde cada elemento aponta especificamente para um serviço seguro. Entre outras informações, o tamanho do elemento na lista encontrava-se definido. Através da *write-primitive*, o atacante alterou o tamanho deste elemento para um valor exagerado (inseguro) e, aquando do varrimento da lista, o processador acabou por executar um código em memória não-segura, criado pelo atacante.

Concluindo, em comparação ao primeiro, este ataque foi mais complexo, contudo resultou na execução de código arbitrário. O atacante teve acesso a dados sensíveis como chaves criptográficas, conteúdo DRM, foi capaz de desligar

mecanismos de segurança existentes no dispositivo e em alguns modelos realizou o *unlocking* do *bootloader*.

Em 2014, o analista Nathan Keltner, partilhou o trabalho desenvolvido pela sua equipa - *Here be Dragons*, [21]. O ataque também explorou o QSEE, contudo modificado pela HTC (OEM), e também resultou na execução de código arbitrário. No entanto, neste ataque, as vulnerabilidades existiam especificamente nos serviços seguros HTC. Após dissecação do QSEE, o analista aborda várias deficiências, como o facto destes serviços serem muito frágeis na validação de *inputs* oriundos do domínio inseguro. Os serviços HTC aumentaram e expuseram em demasia a TCB, que por defeito se encontrava robusta, onde apenas existia *software* da Qualcom. Como consequência da individualidade da HTC e falta de *expertise* em segurança de *software*, no geral o sistema apresentava uma incoerência ao nível de segurança, considerada cómica. "*(...) write vulns pop up like mushrooms from this fertile ground, and write vulns can really ruin your day*" [21]. Uma *zero write primitive* (poderosa, uma vez que permitia gerar código executável) num serviço DRM da HTC, possibilitou ao atacante apagar zonas críticas de memória (responsáveis pela verificação de *inputs*) através de algumas SMCs com parâmetros manipulados. Como resultado, todas as funcionalidades e todos os recursos da área segura da TrustZone ficaram desprotegidos.

A *framework* SeCReT, referida na secção 2.2.7.1, enquadra-se nos ataques anteriormente descritos, pois aborda e tenta mitigar a vulnerabilidade arquitetural da própria tecnologia TrutZone que é o principio dos ataques. O *design* atual da TrustZone não autentica o acesso aos seus recursos sensíveis [16]. O canal de comunicação da tecnologia é inseguro, pois carece de qualquer tipo de mecanismos de proteção suportados diretamente pela mesma. Precisamente devido a isto, um atacante com privilégios ao nível do *kernel* (no REE) possui a capacidade de enviar pedidos (SMCs) com argumentos manipulados, com o objetivo de descobrir vulnerabilidades no *kernel* seguro.

Após a análise dos referidos trabalhos concluiu-se que a tecnologia TrustZone é vulnerável ao tipo de ataques que são o seu principal propósito de *design*, e o sucesso dos ataques se deve, de forma geral, a dois tipos de vulnerabilidades: 1) vulnerabilidades na codificação do *software* seguro presente no mundo seguro da tecnologia TrustZone - a eficácia da tecnologia está altamente dependente da segurança do *software* que executa; 2) vulnerabilidade arquitetural no canal de comunicação, pois a tecnologia não fornece suporte a este canal, como controlo de acesso.

A Tabela 5.1, condensa de forma simples as conclusões retiradas nesta secção, e representa o espectro de segurança da tecnologia.

Tabela 5.1: Espectro de segurança da tecnologia TrustZone

Ataques		**Proteção**	**Observação**
Físicos	Inf. *Debug*	Alta	Proteção eficaz contra estes ataques.
	Outros	Nenhuma	Não concede qualquer proteção.
Side-channel		Baixa	Vulnerável, contudo passível de ser utilizada para mitigar este tipo de ataques.
Trojans		Nenhuma	Não concede qualquer proteção.
Software		Variável	Possui um canal de comunicação frágil e é altamente dependente da segurança do *software* que executa.

O objetivo de *design* da tecnologia TrustZone é a proteção dos seus recursos especialmente contra ataques em *software*. Contudo, em alguns casos, estes quebram a proteção que a tecnologia concede, com mínimo esforço. O analista Gal Beniamini, atualmente em voga devido ao seu trabalho realizado neste âmbito [22], fez estremecer a comunidade cientifica, quando o divulgou em 2016, desvendando um conjunto de vulnerabilidades que permitiam que uma aplicação cliente, sem qualquer privilégio, pudesse executar de código arbitrário na "fortaleza TrustZone". O trabalho incidiu igualmente no QSEE e culminou com 60% dos SoCs Qualcom, atualmente comercializados, vulneráveis ao seu *exploit*, [23].

5.2 Monitorização do canal de comunicação

Sistemas baseados em TrustZone são alvos atrativos aos atacantes. Apesar de todo o esforço que se observa nos dias de hoje, os atacantes têm conseguido derrubar a segurança destes sistemas com sucesso. Como foi referido na secção 5.1, a instrução SMC é o meio que estes utilizam de forma recorrente pois esta instrução concede a maior superfície de ataque ao *software* que se encontra no mundo seguro da tecnologia TrustZone. A exceção SMC é o mecanismo base na comunicação entre os diferentes mundos ou ambientes de execução, é o "*entry-*" e o "*exit-point*" da comunicação que ocorre no mundo seguro. Por tal facto, a instrução, direta ou indiretamente, transmite informação e propriedades sobre o estado de comunicação no mundo seguro. Apesar de o canal de comunicação ser um aspeto crítico para todo o sistema, uma vez que permite o acesso a recursos sensíveis a aplicações clientes não confiáveis, este sofre de uma vulnerabilidade arquitetural igualmente crítica - falta de controlo de acesso (secção 5.1).

Os ataques resumidos anteriormente provam, de forma nítida, a presença e o impacto de uma deficiência na arquitetura TrustZone, mas também, enumeras

vulnerabilidades no *software* seguro desenvolvido por *third-parties*. A ideia da introdução de uma camada de monitorização começa a ganhar maior ascendente, visto em teoria ser possível, através desta, minimizar o impacto resultante das lacunas arquiteturais. Monitorizar uma aplicação consiste na observação de certas propriedades suas ou do sistema, enquanto esta se encontra em execução. O comportamento da aplicação pode ser descrito e identificado com base nessas propriedades. Tendo como termo de comparação uma aplicação, considerada benigna, o valor das propriedades consideradas para comparação que não garantam a concordância, são consideradas perigosas. Uma propriedade natural do canal de comunicação, que o caracteriza e pode ser obtida de forma lateral (efeito independente que resulta do processamento) é o tempo. Num contexto técnico, o tempo refere-se aos ciclos de relógio decorridos no mundo seguro, durante a comunicação com um serviço seguro. Se o acesso ao canal for realizado exclusivamente por uma aplicação de cada vez pode-se, teoricamente, caracterizar a comunicação em função desse serviço de forma trivial. O tempo é apenas um dos inúmeros efeitos naturais e laterais que podem caracterizar o sistema, neste caso, a execução no mundo seguro.

Outro aspeto possível de ser monitorizado é o número de ocorrências da instrução SMC. Um serviço seguro é desenvolvido a par com uma aplicação cliente pois este é desenvolvido para satisfazer as necessidades específicas da aplicação. As suas funcionalidades terão que satisfazer com sucesso e eficientemente as exigências da aplicação cliente, sempre que forem solicitadas. Desta forma, é garantido que o mundo seguro da tecnlogia é utilizado de forma otimizada, o que também contribui para o aumento da sua segurança (pois é contraproducente usufruir do mundo seguro da tecnologia TrustZone se a funcionalidade não for assegurada, assim como esta fica mais exposta). Desta forma, é aceitável considerar que o comportamento de determinado serviço seguro necessário para executar a pretendida funcionalidade é fixo, assim como, o número de ocorrências das exceções SMCs. É consciencioso aceitar que o número de ciclos de relógio executados no mundo seguro, assim como o número de SMCs executados para fornecer a funcionalidade ao cliente, são propriedades que podem ser usadas para caracterizar a comunicação em função de determinado serviço seguro. Contudo, existem alguns aspetos que necessitam de ser entendidos, pois estes implicam mudanças comportamentais na comunicação para com um serviço seguro específico: 1) um serviço seguro pode providenciar diferentes funcionalidades sobre o mesmo recurso; 2) pode fornecer a mesma funcionalidades através de mecanismos distintos; 3) os dados transmitidos entre o par influencia diretamente o comportamento da comunicação.

Podemos concluir que o canal de comunicação pode ser caracterizado em função do serviço seguro para o qual está a ser invocado. Tudo o que foi expresso anteriormente esteve na base da idealização da solução de segurança. Monitorizar permanentemente certas propriedades que caracterizam a comunicação podem proteger o próprio canal e, consequentemente, aumentar o espectro de segurança da tecnologia TrustZone. Por outras palavras, a introdução de uma camada monitorizadora, há já existente camada de *trust* da tecnologia, pode aprimorar a segurança geral de um sistema TrustZone-*Aware*.

5.3 Deteção não intrusiva da SMC

Um dos pontos principais da presente investigação centrou-se na deteção não intrusiva da instrução, que é a base de todo o processo de monitorização - SMC. O termo "não intrusivo" neste contexto, significa a ausência de código específico à solução de segurança, necessário para detetar esta instrução sempre que ela ocorra. Nesta secção é dado a perceber o porquê desta consideração em relação às suas implicações no sistema. Este "princípio" foi pensado tendo em consideração que a solução de segurança não deveria introduzir *overhead* à *framework* base da dissertação, resultante de código específico para a deteção da SMC. Este *software* tem implicações a nível da performance geral do sistema, nomeadamente, o *context-switch* entre os domínios (seguro e não-seguro) tornaria-se mais demorado em resultado do *overhead* introduzido.

Outra razão para a ausência de código, específico para a deteção da instrução, foi a segurança. Como se pretende monitorizar a componente temporal e tal como foi abordado acima na secção 5.2, a instrução SMC pode ser encarada como o perímetro que delimita o mundo seguro e a comunicação que ocorre neste. Após a execução desta exceção, o processador entra no modo Monitor, sendo que este modo é considerado como sendo a camada mais sensível ao nível de segurança num sistema TrustZone, por ser a interface entre os dois mundos. Assim sendo, o código para detetar a instrução aumentaria a TCB desta camada, o que em última instância é um atrativo aos atacantes. A deteção não intrusiva resulta numa maior segurança a nível geral por não introduzir possíveis vulnerabilidades à *framework*.

5.4 Investigação

Detetar a execução da instrução SMC (ou qualquer outra) por parte do processador, de forma não intrusiva e em tempo-real, não é uma tarefa trivial, pelo

que houve a necessidade de realizar uma investigação dedicada a ultrapassar esta dificuldade. A dificuldade na deteção ou identificação de uma instrução escolhida para execução é explicada por vários motivos. O mais plausível é o facto de um processador ARM tradicional (*hardcoded*), não conceder acesso ao *pipeline* ou ao barramento de instruções, uma vez que, essa capacidade requer *hardware* dedicado embebido no processador. A sua área em silício, preço e complexidade aumentariam sem justificação aceitável.

5.4.1 Barramento AXI e visibilidade à PL

Como foi explicado na secção 3.2, o barramento AXI está fortemente presente na arquitetura Zynq. O AXI está profundamente integrado no processador, desempenhando o papel de unir as *caches* L1 ao *core*, por outro lado, é responsável também, por interligar os periféricos instanciados na PL com os processadores na PS. Foram efetuados uma série de avaliações ao barramento AXI disponível na PL, com o objetivo de averiguar se através do barramento é possível detetar a execução da instrução SMC recorrendo a um periférico na PL. Resumidamente, a ideia consiste em averiguar se o barramento AXI utilizado para operações internas é partilhado com os periféricos existentes na PL e, caso isso se comprove, se o barramento concede visibilidade sobre as transações mais internas (i.e., *fetch* de uma instrução da memória). Esta hipótese poderia, em teoria, ser explorada para a deteção da instrução SMC de forma não intrusiva, ou seja, um periférico por meio de uma monitorização constante das transações do barramento poderia ser capaz de identificar a instrução SMC através do seu *opcode* ou endereço em memória. As bases para o desenvolvimento da solução de segurança estariam satisfeitas.

Os testes foram realizados na arquitetura Zynq. Na zona reconfigurável (PL - FPGA) instanciou-se um periférico (desenvolvido para o efeito), que estava interligado aos processadores da PS através de portos PS-PL de interface AXI. O periférico foi modificado para ter acesso a todos os *bits* do canal de endereços e possuía lógica adicional para detetar a ocorrência de um endereço específico. Com isto, pretendia-se detetar a ocorrência da instrução SMC, usando o seu endereço em memória previamente conhecido ou *opcode* único. Os resultados dos testes foram conclusivos e coerentes, no sentido em que, o barramento acessível ao periférico não concede a visibilidade pretendida. A arquitetura Zynq baseia-se na arquitetura Harvard (Figura 5.1), em que dois barramentos físicos e independentes existem para o processamento de instruções e de dados, respetivamente. Um periférico em zona reconfigurável não possui a capacidade de monitorizar transações

internas do processador, como aquelas necessárias para realizar o *fetch* de uma instrução à memória.

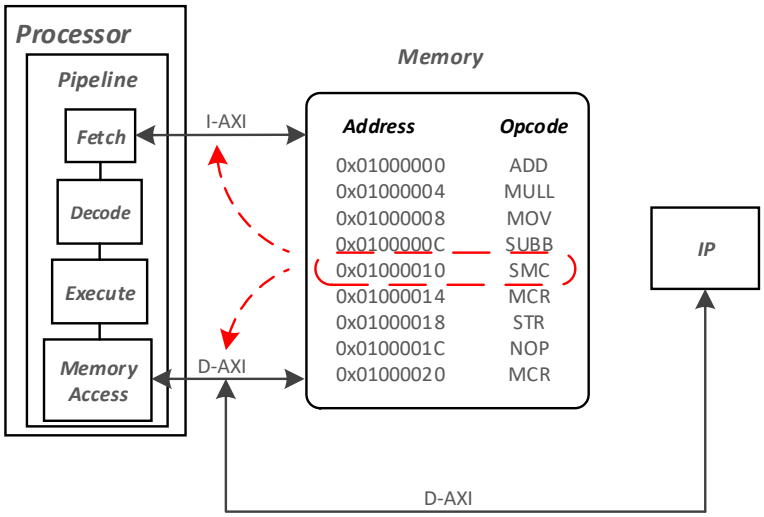


Figura 5.1: Arquitetura *Harvard* na Zynq

5.4.2 Componentes CoreSight na Zynq

Os componentes de *debug* e *trace* presentes na Zynq seguem a especificação CoreSight da ARM. A CoreSight é uma tecnologia muito difundida nos processadores ARM. A tecnologia abrange vários tipos de componentes em *hardware*, intimamente ligados ao processador. As suas responsabilidades consistem no auxílio do processo de depuração para que seja realizado em tempo-real com elevada precisão e eficiência.

A tecnologia divide-se em quatro grandes classes de componentes: Controlo e Acesso, *Trace Source*, *Trace Link* e *Trace Sink*. A classe Controlo e Acesso engloba os componentes de interface entre o utilizador e toda a infraestrutura de *debug*. Os componentes da classe *Trace Source* são os que produzem a informação de *trace* como é o caso dos endereços das instruções, endereços das transações e o empacotamento dessa informação [24]. Os pacotes de *trace* são enviados para os componentes da classe *Trace Link* onde são combinados ou replicados, e posteriormente são novamente enviados para a *Trace Sink*. Aqui, a classe de componentes é responsável por transmitir essa informação até ao exterior recorrendo a componentes como o *Embedded Trace Buffer* (ETB) ou *Trace Port Interface Unit* (TPIU). Todos os componentes estão interligados via três barramentos independentes: Programação, *Trigger* e *Trace*. O primeiro, programação, é o caminho utilizado pelos componentes da classe controlo e acesso para transmitir informação

a cada componente da tecnologia, acerca da sua configuração de funcionamento. Os sinais de *trigger* são igualmente utilizados por todos os componentes para enviarem *triggers* entre si, de maneira a coordenar a sua operação. Por último, o barramento de *trace* é o caminho principal onde os pacotes de *trace* circulam por todas as classes, interligando-as. Na Figura 5.2 encontra-se representado a arquitetura *Debug* e *Trace* presente na Zynq.

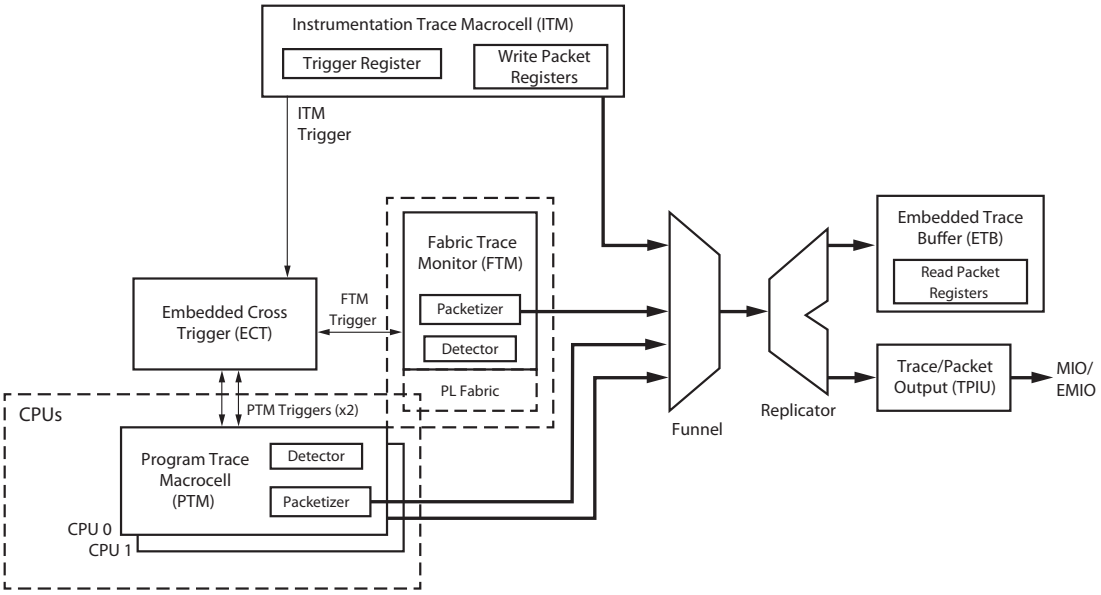


Figura 5.2: Arquitetura CoreSight na Zynq

O Program Trace Macrocell (PTM) é um componente da tecnologia que pertence à classe *Trace Source*. Desta forma, é um componente responsável pela captura de informação relativamente ao estado de execução do processador. Encontra-se profundamente integrado no processador e, deste modo, proporciona que a aquisição de informação de *trace* seja realizada em tempo-real e de forma precisa. Este componente, é especialmente relevante, uma vez que possui um conjunto de recursos internos capazes de detetarem uma instrução através do seu endereço em memória, quando esta é colocada em execução pelo processador. Teoricamente, esta capacidade pode ser utilizada para efetuar a deteção não intrusiva da instrução SMC.

Existiu a necessidade de efetuar alguns testes com o objetivo de validar a teoria, nomeadamente, era de interesse perceber: 1) como se deveria configurar o PTM para se detetar a instrução pretendida; e 2) como é que se poderia informar o periférico (na PL) da ocorrência desse fenómeno. 1) A deteção da instrução pretendida é realizada através da configuração de alguns recursos do PTM, em particular um *Address Comparator* (AC) é pré-definido com o valor do endereço

da instrução. De seguida, outro recurso, neste caso um contador, é configurado para decrementar quando ocorre um *match* entre o endereço pré-definido no AC e o endereço que circula no barramento de instruções do processador; 2) O valor do contador encontra-se mapeado em memória, ou seja, visível a todo o sistema através do endereço de um registo. A solução pode explorar esta particularidade e monitorizar o conteúdo deste registo e assim, obter a informação que a SMC ocorreu.

A Figura 5.3 ilustra, de forma geral, o processo de deteção de uma instrução através do PTM (esquerda) e a monitorização desse fenómeno levado a cabo por um periférico na PL (direita).

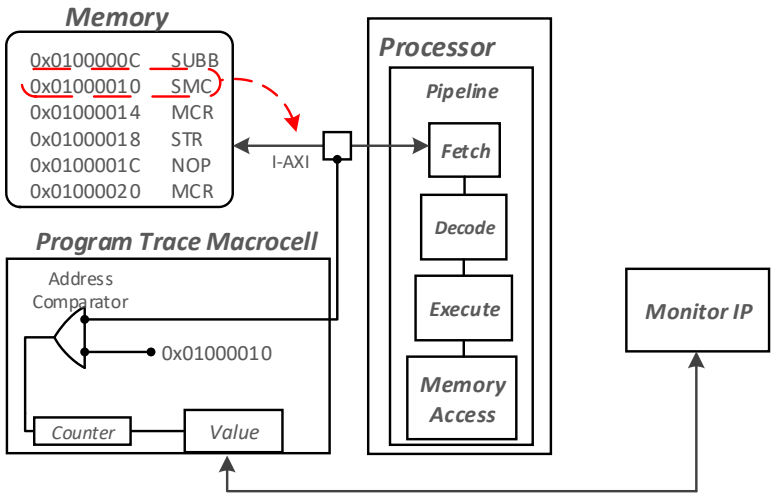


Figura 5.3: Deteção de uma intrução através do PTM

Os testes realizados validaram as suposições iniciais. Um periférico instanciado na PL conseguiu detetar a ocorrência de uma instrução predefinida pelo AC. As bases para a realização da monitorização foram alcançadas. Contudo, na infraestrutura de *Trace* e *Debug*, existe outro componente, a *Performance Monitoring Unit* (PMU), com utilidade para a investigação. Este possui a capacidade de contar os ciclos de relógios realizados pelo processador numa zona de *software* que seja de interesse ao utilizador. Ou seja, a PMU, pode ser utilizada para fornecer, com elevadíssima precisão e simplicidade, os ciclos de relógio decorridos da comunicação no mundo seguro. A Figura 5.4 representa a forma com a PMU pode ser utilizada por um periférico para fins de monitorização. O valor indicador do número de ciclos encontra-se mapeado em memória através de um registo e, por isso, visível ao periférico.

Os componentes descritos nesta secção revelaram-se absolutamente essenciais para traçar um perfil comportamental fidedigno da comunicação. Além de essenciais são únicos pela sua capacidade de aceder/conceder informação interna e precisa

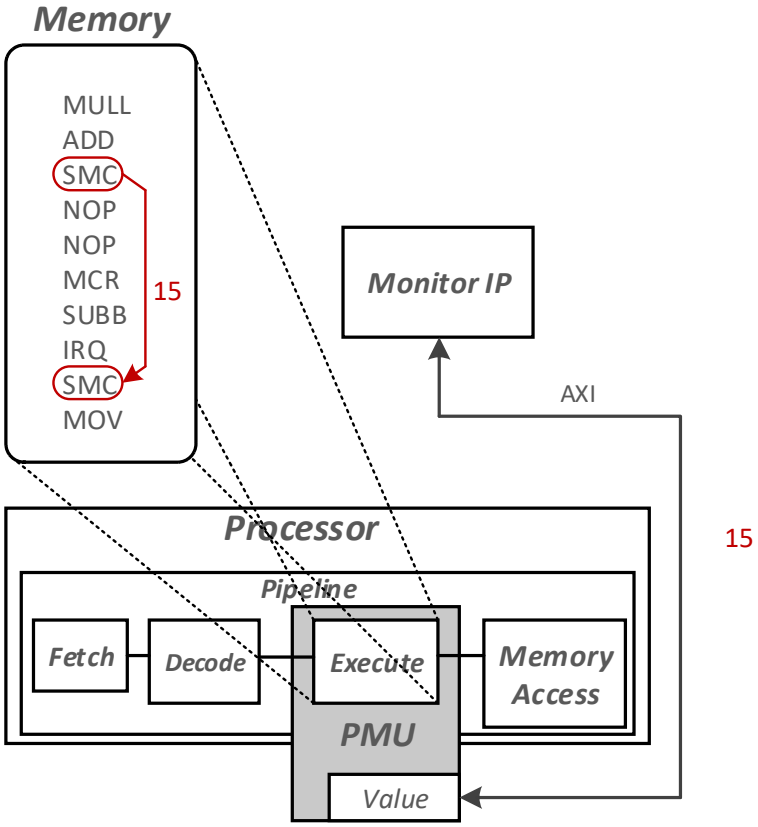


Figura 5.4: Contagem do número de eventos através da PMU

sobre a execução do processador. Como irá ser observado nas próximas secções, a simbiose entre a infraestrutura de *Trace* e o periférico monitorizador foram a chave para a solução de segurança como um todo.

5.5 *Communication Monitor* - COMON

O COMON é um componente monitorizador, desenvolvido em *hardware*, com o objetivo de aumentar a proteção em torno do canal de comunicação da tecnologia TrustZone. Como já foi referido nas secções 5.1 e 2.2.7.1, o canal não possui quaisquer mecanismo de proteção (controlo de acesso), o que leva a que este seja facilmente alvo de ataques. O canal emite certas propriedades possíveis de serem monitorizadas, recorrendo a componentes de *trace* disponíveis na própria plataforma. O COMON introduz uma fina camada de monitorização sobre a tecnologia TrustZone, criando um padrão comportamental da execução do canal de comunicação que é estabelecido no mundo seguro a pedido de determinada aplicação cliente. A comunicação do TEE é monitorizada e é interrompida quando o seu padrão comportamental se desvia do que o COMON aceita como benigno.

O periférico foi desenvolvido na PL da Zynq (FPGA) e encontra-se acoplado aos processadores da PS através dos portos de interface AXI de propósito geral (GPAXI), *slave* e *master* (Figura 5.5). O comportamento do COMON é descrito em duas fases: 1) um período de aprendizagem (*learning*), seguido 2) de um período de monitorização (*monitoring*). O primeiro é uma fase adaptativa, tipícamente de curta duração e realizada antes do *deployment* do sistema. Durante esta fase é caracterizado o canal associado ao par específico, aplicação cliente e serviço seguro. O período de *monitoring* é onde ocorre a proteção ao canal em tempo-real, que esta associado ao par. É comparado o padrão que foi gerado na primeira fase, com a informação que é recolhida no momento.

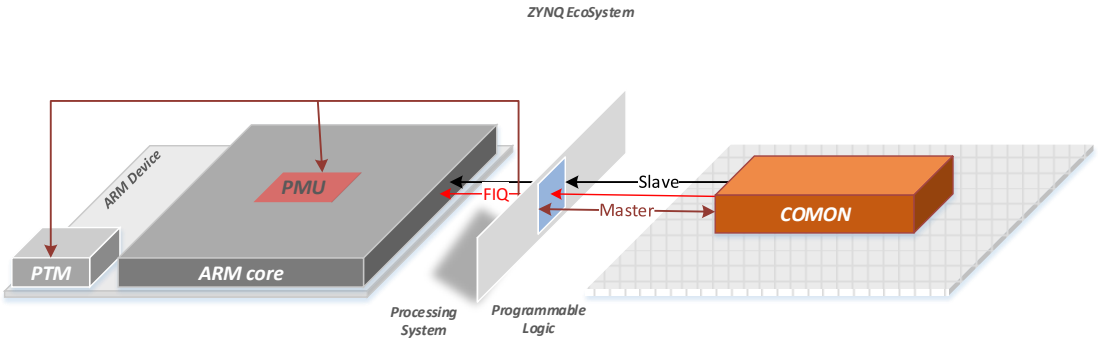


Figura 5.5: Acoplamento do COMON à Zynq

O componente é composto por três módulos, tal como se observa na Figura 5.6: 1) um módulo AXI (*wrapper* AXI) que envolve todos os sinais das interfaces *master* e *slave*; 2) um pequeno módulo que age como um *driver*, configurando e preparando os componentes de *trace* da CoreSight, para serem posteriormente utilizados pelo COMON para a monitorização não intrusiva e, por último: 3) o terceiro módulo pode ser encarado como sendo o motor da monitorização, pois é responsável por iniciar transações necessárias, interpretar as suas respostas e realizar simples operações estatísticas a fim de traçar o perfil da comunicação. A escolha de *design* no segundo ponto, é análoga à explicada na secção 5.3 e prendeu-se com a segurança geral do L-TEE, pois visa não aumentar escusadamente a TCB do mesmo.

5.5.1 Proteção do Canal

A falta de controlo de acesso é a característica da tecnologia TrustZone explorada pelo COMON, para a introdução de uma camada de monitorização. O L-TEE, através desta camada, torna-se consciente de intrusões que exploram *bugs*

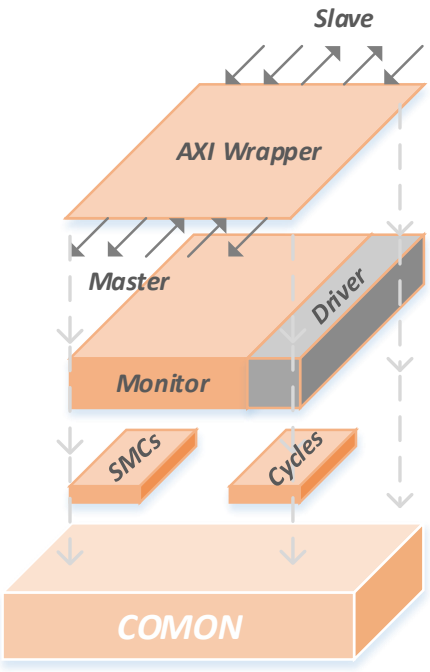


Figura 5.6: Sub-componentes do COMON

no código em qualquer um dos seus componentes seguros (T-RTOS e Monitor) e consequentemente modificam o funcionamento correto do canal.

A configuração do COMON deve ser realizada antes da utilização do canal ou durante a inicialização do sistema, englobando a programação de quatro registos. Antes de se proceder ao *enable*, último *bit* a ser ativo através do registo de controlo (cntr_reg), têm de ser configurados os endereços base do PTM e PMU (que dependem da plataforma) nos registos ptm_addr e pmu_addr, respetivamente. Estes serão usados pelo módulo *driver* que será explicado no próximo parágrafo. Seguidamente a configuração é feita programando o mvbar_reg com o endereço do monitor *vector table*. O COMON calcula o SMC *handler* (que é um *offset* fixo deste endereço base) e, tal como foi descrito na secção 5.2, é usado para descrever a comunicação segura que ocorre no mundo seguro.

As transações AXI abordadas de seguida serão referidas de forma abstrata, pois os acessos de leitura ou escrita subentendem a escolha do canal apropriado da interface AXI, assim como a escrita do endereço e dado (se for o caso) no barramento respetivo (tópico abordado na secção 3.2). O *driver* implementado é responsável por configurar o PTM e a PMU, componentes que vão auxiliar o processo de monitorização. A máquina de estados, na Figura 5.7, resume a sequência de operações que é realizada. A PMU, assim como todos os componentes CoreSight, não são totalmente acessíveis através da sua interface *memory mapped*. Para total uso das suas funcionalidades é necessário primeiramente proceder-se ao

seu desbloqueio (*unlock*) através da escrita de um valor (0xC5ACCE55) definido pela ARM num registo de desbloqueio PMLAR (estado PMU UNLOCK). Seguidamente, o contador deve ser parado por se encontrar por defeito ativo, através da escrita (0x80000000) no registo PMCNTENSET, o que acontece no estado COUNTER STOP. A configuração do PTM é consideravelmente mais extensa, pelo que não será totalmente detalhada, contudo, a nível geral esta consiste, para além do *unlock* (PTM UNLOCK), na programação do recurso *Address Comparator* com o SMC *handler* anteriormente calculado (AC VALUE e AC TYPE), assim como, a configuração do contador que concede a sinalização (registo ETMCNTVR1) da execução do SMC no processador (COUNTER VALUE, COUNTER EVENT, RELOAD VALUE e RELOAD EVENT). Após todas as configurações do PTM serem realizadas é necessário sinalizar através da escrita (0x00000000) no registo de controlo principal (ETMCR), o que é realizado no último estado CLEAR PROGBIT.

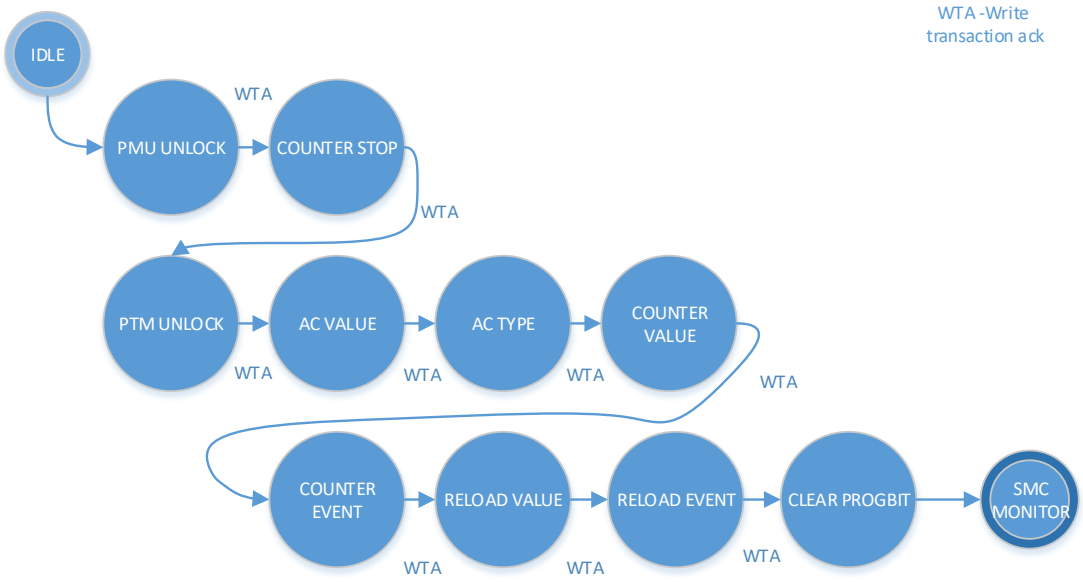


Figura 5.7: Máquina de estados do comportamento do *driver*

Após todas as configurações necessárias à monitorização estarem completas, o COMON dá início à monitorização propriamente dita. No primeiro estado, SMC MONITOR, é monitorizado a ocorrência de um SMC, através de uma leitura contínua ao registo ETMCNTVR1. Quando ocorre um SMC, ou seja, quando o valor do registo ETMCNTVR1 é decrementado, são efetuadas duas transações de escrita consecutivas. Uma realiza o *reset* do contador da PMU (PMU RESET) e a seguinte inicia a contagem desse mesmo contador (COUNTER ENABLE). O próximo estado é idêntico ao primeiro, é monitorizado o ETMCNTVR1 até ocorrer alteração do seu conteúdo. Caso isso se verifique, o número de ciclos decorridos

durante o estado inicial e o estado atual representam o tempo da comunicação no mundo seguro (de uma fase da comunicação). Consequentemente, a próxima transação, realizada no estado COUNTER STOP, é uma escrita em PMCNTENCLR para parar o contador. O contador da PMU é lido paralelamente caso não ocorra a SMC, para que o COMON esteja sempre ciente do número de ciclos decorridos (PMU READ). No último estado, SESSION CLOSE, o COMON certifica-se se o SMC de retorno corresponde ao fim de uma sessão de comunicação, para esse efeito, realiza uma transação de leitura a um endereço específico. Este estado, depende da implementação da comunicação do L-TEE. O diagrama da Figura 5.8 apresenta, de forma resumida, a sequência de estados anteriormente descrita, que por sua vez, se repete todas as vezes que o canal de comunicação é utilizado.

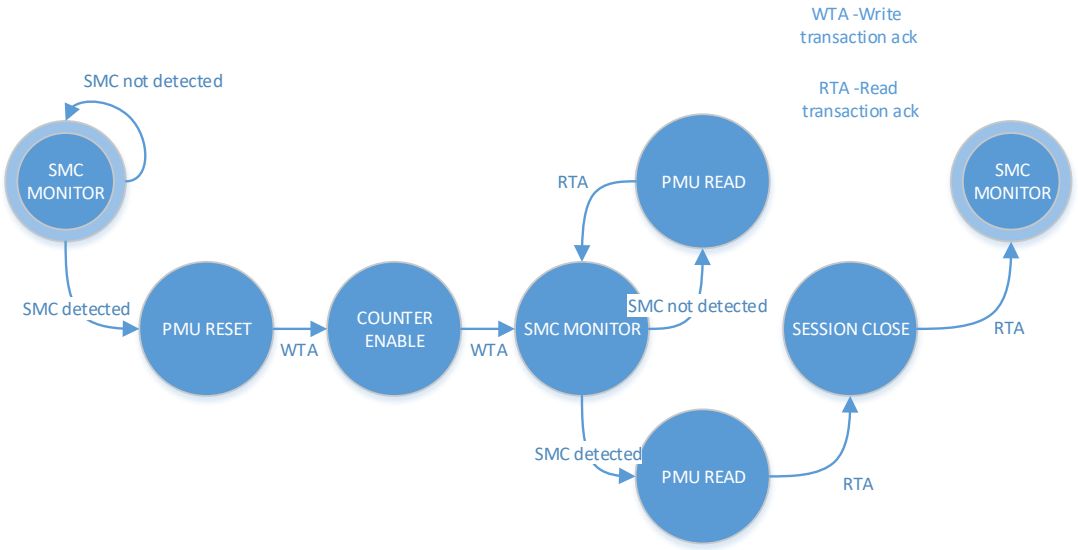


Figura 5.8: Máquina de estados representativa da monotorização

As operações estatísticas são calculadas com base no resultado de algumas transações. A ocorrência da SMC é detetada quando ocorre a transição do primeiro estado (SMC MONITOR) para o segundo (COUNTER ENABLE). A quantidade de ciclos é extraído do *income* da transação do quinto estado (PMU READ). O padrão que é construído durante o período de *learning*, recorre a uma memória (desenvolvida para o efeito), na qual é armazenada a informação temporal e um contador que é responsável por manter a par do número de SMCs. O ciclo de monitorização, descrito no parágrafo anterior, repete-se nSMC vezes (até a *flag* fim de sessão ser detetada), multiplicado por *nLearning* repetições, ou seja, o número pré-definido de execuções do par aplicação cliente-serviço seguro. O período de *learning* é mais preciso quanto maior for o número de análises comportamentais

(*nLearning*). Antes desta fase terminar é calculada a média dos ciclos de relógios representativos a cada fase da comunicação, bem como do número de SMCs igualmente característicos de toda a sessão de comunicação. A monitorização é realizada a duas dimensões (número de SMCs e ciclos de relógio): estas componentes "esboçam" o perfil da comunicação. Nenhuma execução de determinado código num processador possui um comportamento exato e estático. Um processador possui diversas variáveis que interferem com a execução de código e que variam de execução para execução. Devido a estas não-linearidades, é adicionado um *threshold* às médias das duas componentes, representativa desta elasticidade na execução. Assim, é assegurado que a atuação realizada pelo COMON é mais flexível.

O COMON possui também um sinal que se encontra interligado ao CPU e que usa para interromper a sua execução através de uma interrupção FIQ. A configuração da interrupção oriunda da PL, associada ao COMON, é realizada tal como ilustrado na Lista 5.1. Mais uma vez, recorreu-se às funcionalidades proporcionados pelo T-RTOS para habilitar a interrupção no *interrupt destributor* (linha 5), associar a rotina de atendimento à interrupção do COMON (Lista 5.2) à interrupção (linha 6), e definir a prioridade da interrupção (linha 7).

Lista 5.1: Configuração da interrupção do COMON

```
1  static void prvSetupTimerInterrupt( void )
2  {
3      /* (...) */
4
5      interrupt_enable(PL2PS_INTERRUPT,TRUE);
6      vFreeRTOS_handler_set(PL2PS_INTERRUPT,vCOMON_handler);
7      interrupt_priority_set(PL2PS_INTERRUPT,5);
8
9      /* (...) */
10 }
```

Na Lista 5.2 encontra-se a rotina dedica a realizar uma ação quando o COMON gera um interrupção FIQ. Nas linhas 5 e 6, a interrupção FIQ é limpa, para que novas intrusões possam ser detetadas. Devido a razões de *time-constraint* não foi implementado nenhuma ação, contudo essa tarefa pode ser realizada de forma trivial dependendo da criatividade do desenvolvedor.

Lista 5.2: Rotina de atendimento à interrupção do COMON

```
1  void vCOMON_handler( void ) {
2      uint32_t *comon_ptr;
3
```

```
4     printf("COMON FIQ interrupt\n\t");
5     comon_ptr = (uint32_t*)cntr_reg;
6     *comon_ptr = CLEAR_INTERRUPT;
7     while(1);//Security action
8 }
```

Em suma, o COMON através dos seus módulos implementa uma monitorização dinâmica da execução do código da comunicação no L-TEE, que o permite detetar alguns ataques de *software* que podem ser totalmente novos e nunca antes estudados, realizando proteção sem necessitar de recorrer a métodos anacrónicos, como a instrumentação estática do código (abordagens *signature based).*

Resultados

Este capítulo foca-se principalmente no ataque realizado à *framework* (secção 6.2), por forma a avaliar a proteção concedida pelo COMON. Contudo, é feito primeiramente na secção 6.1 uma validação da comunicação implementada e, no final do capítulo na secção 6.3, é feito uma análise de segurança concedida por toda a solução de segurança,

6.1 Validação da Comunicação

A presente secção pretende dar a conhecer ao leitor, em traços gerais, a comunicação implementada no L-TEE por forma a provar a sua efetividade, assim como, retratar o comportamento que existe entre o par de aplicações desenvolvido. O par é composto pela aplicação cliente ECHO_CLI (REE) e o serviço seguro ECHO_SS (TEE). A ideia geral do par de aplicações é simples, mas suficiente para provar o conceito: os dados criados e requisitados para a comunicação por parte do ECHO_CLI devem ser recebidos e novamente reenviados a este, por parte do ECHO_SS, como de um eco se tratasse.

A comunicação é realizada em três fases e subentende a utilização da TEECAPI (secção 3.3.1). Numa primeira instância, é necessário que se estabeleça uma sessão de comunicação com o ECHO_SS. A segunda fase é onde ocorre a troca de dados, sendo possível que a troca possa ser realizada de distintas formas, mediante o desejo do ECHO_CLI. A última fase tem como objetivo finalizar a sessão estabelecida.

Na primeira fase ocorre a abertura de sessão e para isso o ECHO_CLI fornece o UUID do serviço ao *kernel module*. Este é o responsável por iniciar e manter um contexto com o TEE e também, por facultar-lhe o pedido do ECHO_CLI, que neste momento se encontra codificado numa estrutura própria. O pedido é reencaminhado ao Monitor através da execução da instrução específica SMC, este por sua vez transfere-o ao T-RTOS. A sessão de comunicação é estabelecida se o contexto e o UUID forem válidos. Na Figura 6.1 está representado o diagrama de sequência respetivo à fase descrita.

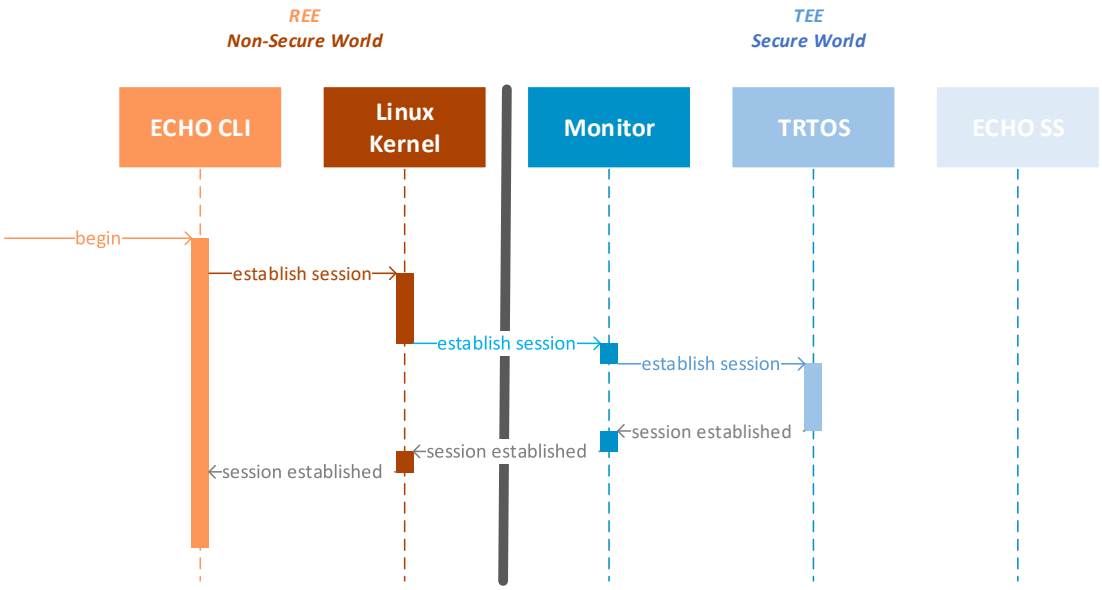


Figura 6.1: Abertura de sessão da comunicação

Na segunda fase, o ECHO_CLI aloca um bloco de memória para a troca de dados e para isso recorre a funcionalidades disponíveis no Linux *kernel module*. A zona de memória criada será utilizada como *container* de envio e de resposta, ou seja, os dados do utilizador serão escritos no *buffer* e a resposta por parte do ECHO_SS é colocada também no mesmo, contudo a partir de um *offset* especificado. Após tratamento dos dados, é invocado o pedido para que o ECHO_SS seja executado e se realize o eco. Neste ponto, é possível optar por diferentes mecanismos para a realização da funcionalidade, seja através de memória partilhada ou outro mecanismo mais simples. Essa distinção não altera o fluxo de interação, contudo é importante salientar que a escolha do mecanismo influencia o comportamento geral da comunicação. O pedido é enviado ao *kernel module*, este realiza as normais operações de coerência e reencaminha o pedido ao Monitor através da instrução SMC. O Monitor, por sua vez, transfere o pedido ao T-RTOS que irá criar o serviço associado à sessão que se encontra estabelecida. São utilizadas as funcionalidades de criação de tarefas nativas do T-RTOS e de seguida o controlo de execução é transferido ao ECHO_SS. O comportamento do ECHO_SS é simples: copia os dados que se encontram em memória partilhada (não-segura), para um espaço de endereçamento seguro. De seguida, é realizado o eco: os dados, agora seguros, são novamente copiados para o *buffer*, mas para uma posição definida pelo *offset*. Durante este processo, o ECHO_SS recorre a funcionalidades de gestão de memória do T-RTOS. Após conclusão do processo, o T-RTOS é responsável por terminar a tarefa e o pedido é enviado novamente para o REE, realizando o processo inverso. O ECHO_CLI pode agora fazer a leitura

dos dados que se encontram presentes no *buffer*, no previamente estipulado *offset*. A sequência referente à segunda fase está ilustrada no diagrama da Figura 6.2.

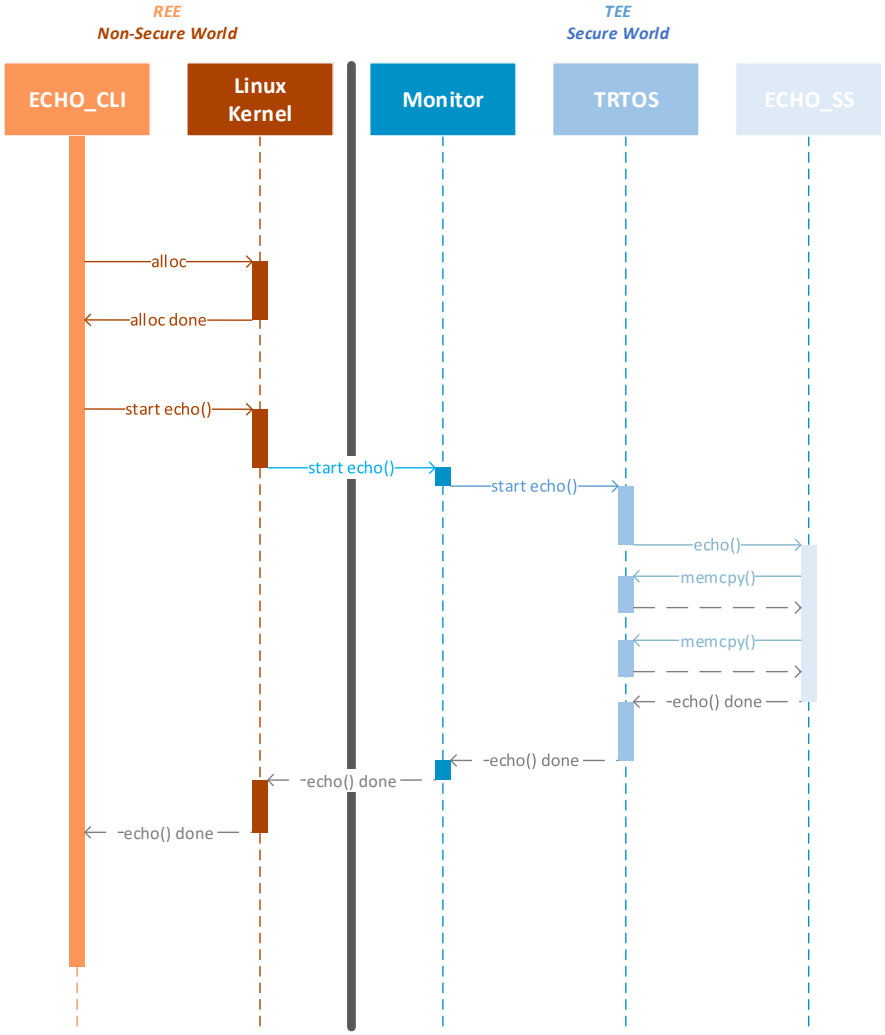


Figura 6.2: Realização da funcionalidade eco

O tamanho dos dados envolvido nesta fase tem interferência direta no tempo da comunicação. As funcionalidades de gestão de memória, disponíveis no T-RTOS, realizam o armazenamento em memória dos dados. O tempo dessa tarefa depende diretamente do número de dados no *buffer* transmitidos entre o par de aplicações.

A última fase é idêntica à primeira a nível de interação com ECHO_SS. Contudo o ECHO_CLI passa informação sobre a sessão (e não sobre o identificador do serviço UUID) ao *kernel module*, com o objetivo de terminar a sessão comunicativa que se encontra estabelecida. O *kernel module* trata de realizar as operações necessárias à coerência do contexto com o TEE. Após isto, reencaminha o pedido ao Monitor, que por sua vez o transfere para o T-RTOS. A sessão de comunicação é terminada se o contexto e a informação sobre a sessão estabelecida forem válidos.

6.2 Ataque ao LTEE

Implementou-se um ataque com o objetivo de demonstrar o conceito de proteção adjacente à monitorização do canal de comunicação. O ataque segue as mesmas linhas estratégicas dos ataques analisados na secção 5.1, na medida em que necessitaram de uma análise e compreensão da implementação do T-RTOS e de toda a comunicação, como o suporte *back-end* no T-RTOS e no GPOS, concretamente sobre a TEECAPI e *kernel module*, por forma a descobrir vulnerabilidades de segurança no *software*. Similarmente aos ataques analisados, este ataque requer que as aplicações malignas desenvolvidas possuam privilégios ao nível do *kernel*. Os ataques também utilizam a instrução SMC como sendo o mecanismo que explora as vulnerabilidades, pois a instrução é o meio que fornece a maior superfície de ataque ao mundo seguro da tecnologia TrustZone.

Recapitulando, o canal deveria apenas e só, ser utilizado para o estabelecimento de uma comunicação efetiva entre o par de aplicações, garantido que a tecnologia é utilizada de forma eficiente. Contudo, a falta de controlo de acesso ao canal, permite que este possa ser utilizado de forma insegura. Um atacante com privilégios *kernel* pode influenciar toda a comunicação por forma a explorar vulnerabilidades de segurança no TEE, com o objetivo principal de ganhar controlo do mundo seguro, ou também, de desencadear alguma operação sensível que lhe seja benéfica. O atacante, logo à partida, tem total controlo sobre o bloco de memória que será alocado e usado pelo TEE para a comunicação e, mais relevante ainda, pode influenciar completamente a estrutura principal da comunicação (apresentada na Lista 6.1). Esta, entre outros parâmetros, contém os endereços da memória partilhada (tal informação é expressa através de *req_buf_phys* e *resp_buf_phys*), o UUID (*id*) único ao serviço (que pode ser usado para executar um serviço detentor de vulnerabilidades), o elemento *meta_data_phys* que concede informações sobre os dados em memória (como o seu tipo e tamanho), entre outros. É apenas exigido ao atacante que compreenda a comunicação que existe nos dois mundos. Tal tarefa não demonstra ser um grande obstáculo, muito por causa de ferramentas de engenharia reversa do código como IDA Pro, que permitem que mesmo código confidencial possa ser meticulosamente analisado. O *software* no mundo seguro encontra-se muitas vezes disponível na Internet, ou então, a título de exemplo em sistemas Android, o *software* pode ser extraído recorrendo a funcionalidades próprias do SO.

A estrutura *otz_smc_cmd* é enviada juntamente com a instrução SMC no registo de propósito geral r1, sendo este um apontador para a estrutura que é preenchida no REE.

Lista 6.1: Estrutura principal da comunicação

```
1  struct otz_smc_cmd {
2     unsigned int     id;
3     unsigned int     context;
4     unsigned int     enc_id;
5     unsigned int     src_id;
6     unsigned int     src_context;
7     unsigned int     req_buf_len;
8     unsigned int     resp_buf_len;
9     unsigned int     ret_resp_buf_len;
10    unsigned int     cmd_status;
11    unsigned int     req_buf_phys;
12    unsigned int     resp_buf_phys;
13    unsigned int     meta_data_phys;
14    unsigned int     dev_file_id;
15 };
```

A manipulação do canal por parte do atacante irá interferir com o seu comportamento. A comunicação no mundo seguro engloba, como já foi visto na secção 6.1, um serviço seguro em particular mas também, o Monitor e o T-RTOS. Uma tentativa de exploração de debilidade em qualquer um destes intervenientes poderá alterar o comportamento normal do canal, pelo que, através da monitorização se pretende detetar esse efeito.

6.2.1 Modelo e Considerações

Do ponto de vista do ataque que foi desenvolvido, é assumida a existência de um TEE ciente da tecnologia TrustZone, ou seja, segundo a própria ARM [5], no mundo seguro encontram-se diversos recursos atrativos a atacantes (e.g., mecanismos para *unlock* do *bootloader*, chaves criptográficas, credenciais bancárias), isolados do mundo não-seguro onde executa um GPOS tradicional. Assume-se também, a existência do problema de segurança ortogonal da tecnologia, que está associado ao seu canal de comunicação que permite explorar vulnerabilidades no *kernel* seguro. Até à data não existe nenhuma solução certificada que o colmate. O problema pode ser explorado por atacantes através da invocação de SMCs manipuladas, é por isso assumido também, que o atacante possui privilégios ao nível do *kernel*. A elevação de privilégios de uma aplicação que não possua qualquer privilégio, é uma "ação" que pode ser alcançada com relativa facilidade, pois existem diversas ferramentas que simplificam e automatizam esse processo (e.g., *rootkits*). Pressupõe-se que o atacante possui total acesso às *stacks* de *software* associadas ao

REE (código fonte do GPOS), o que é aceitável pois tipícamente estes são *open-source*. Por último, assume-se que o atacante possa ter acesso ao código fonte no TEE, recorrendo a mecanismos como engenharia inversa apoiada por ferramentas adequadas.

Na perspetiva do periférico desenvolvido - o COMON - é igualmente assumida a existência de um TEE TrustZone-*Aware* e arquitetura de *trace* e *debug* CoreSight, detentora dos componentes PTM e PMU. Modificações introduzidas no *hardware* durante os estágios iniciais de fabrico, assim como, ataques físicos com ambição de observar a constituição *chip*, ou influenciar o seu comportamento através de sondas, não são considerados no ataque. *Software* não certificado, que possa potencialmente ser inicializado no sistema, não é de igual modo considerado pois é assumida a existência de mecanismos como o *secure boot* que garantem a integridade do *software* que executa no mundo seguro da tecnologia TrustZone. Intuitivamente, o *secure boot* apenas garante a integridade do *kernel* durante a inicialização do sistema. Depois desta fase, e quando o sistema começa a interagir com o atacante, o mecanismo não oferece quaisquer garantias. Ataques físicos que exploram a infraestrutura de *debug* encontram-se protegidos pela própria tecnologia. Acessos ilegítimos do GPOS à memória segura, com objetivo de comprometer a segurança, encontram-se bloqueados por mecanismos em *hardware* suportados pela tecnologia (TZASC). Para finalizar, ataques que restringem o escalonamento de funcionalidades seguras (estilo DoS) não são também considerados.

6.2.2 *Ataque Code Execution*

O ataque realizado permitiu a alteração da execução do código no mundo seguro da tecnologia TrustZone e por isso denomina-se de *code execution*. Como o nome indica, este tipo de ataque altera no fluxo normal de execução do programa, resultando na execução de código arbitrário. O ataque foi dividido em duas sub-secções (6.2.2.1 e 6.2.2.2) tendo em conta as fases envolvidas no seu desenvolvimento. Em primeiro lugar (secção 6.2.2.1), foi analisada toda a *stack* de *software* pertencente à comunicação no mundo seguro. Posteriormente, na secção 6.2.2.2, procedeu-se ao desenvolvimento e execução do ataque.

6.2.2.1 Análise da *stack* da comunicação

Pode parecer contraditório efetuar uma análise à comunicação, no entanto, esta análise consistiu na observação do código numa outra perspetiva, capaz de conceder percepção de vulnerabilidades de segurança, que pudessem permitir a execução de um ataque. Como pressuposto na secção 6.2.1, o atacante possui

privilégios *kernel*, podendo manipular todo o canal de comunicação de forma insegura (invocar a instrução SMC). Quando esta exceção é executada, uma rotina de atendimento à interrupção é invocada, _mon_smc_handler. A rotina encontra-se implementada no Monitor e verifica se a SMC se trata de um pedido para iniciar comunicação ou de um retorno de uma comunicação anterior. Nesta rotina, é também verificado se o pedido se trata de um escalonamento explícito do GPOS. Esta condição apenas se verifica uma vez, durante a inicialização do sistema.

Lista 6.2: Superfície de ataque no Monitor

```
1  .globl _mon_smc_handler
2  _mon_smc_handler:
3
4     ; Call TrustZone API
5     cmp    r0, #CALL_TRUSTZONE_API
6     beq    tz_api_call
7
8     ; Return TrustZone API
9     cmp    r0, #RET_SECURE_API
10    beq    ret_from_secure_api
11
12    ; Schedule non-secure
13    cmp    r0, #SCHEDULER_MACRO
14    beq    _mon_scheduler
```

Na Lista 6.2, observam-se as referidas condições. Estas foram mencionadas por pertencerem à superfície de ataque que primeiramente é do contacto do atacante. Intuitivamente, deve ser invocado a instrução SMC com conteúdo em r0 definido por CALL_TRUSTZONE_API. Ocorrerá a comutação entre mundos, durante a execução da função tz_api_call. Nesta, existem duas variáveis de interesse: params_stack e params_flag. A primeira corresponde a um vector onde são armazenados os registos de propósito geral (r0 a r3) que são enviados juntamente com a SMC. A params_flag é habilitada dentro da função e será utilizada à posteriori. Na Lista 6.3 observa-se a inicialização das referidas variáveis. A camada Monitor não é detentora de vulnerabilidades, o que se explica por esta ter sido desenvolvida cuidadosamente e de forma minimalista.

Lista 6.3: Variáveis relevantes da rotina tz_api_call

```
1  tz_api_call:
2     ldr       r4, =params_stack
3     stmia     r4, {r0-r3}
4
5     ldr       r1, =params_flag
```

```
6     mov       r2, #0x1
7     str       r2, [r1]
8
9     ; Switch to secure world
10    SCR_NS_CLEAR
11
12    ; Save GPOS context
13    ; (...)
14
15    ; Restore TRTOS context
16    ; (...)
17
18    movs  pc,lr
```

Após comutação de contexto, o T-RTOS retoma a execução da tarefa *idle*. Nesta, encontra-se a maior parte do suporte à comunicação o que enfatiza a sua importância. Todo o suporte é realizado sobre a forma condicional (*if...else*). A primeira condição apresenta-se através de um e-lógico entre a *flag* anterior e o primeiro elemento do vetor params_stack (ou seja r0). Esta condição lida com os pedidos associados à comunicação, e como pode ser observado através da Lista 6.4, não apresenta qualquer obstáculo pois os seus parâmetros já apresentam o valor desejável ao atacante.

Lista 6.4: Variáveis que permitem a execução da get_api_context()

```
1  /* (...) */
2  int svc_id, task_id, cmd_id, cmd_type;
3  /* (...) */
4  for( ;; )
5  {
6     /* (...) */
7     if(params_flag && params_stack[0] == CALL_TRUSTZONE_API){
8        params_flag = 0;
9        get_api_context(&svc_id,&task_id, &cmd_id, &cmd_type);
10       /* (...) */
11    }
12 /* (...) */
13 }
```

Observa-se também na referida listagem, que será invocada a função da linha 9 - get_api_context(). Esta função é responsável por manter o contexto da comunicação e, para isso, inicializa variáveis como svc_id, cmd_id e cmd_type que serão utilizadas posteriormente, por outras rotinas. As duas primeiras representam, respectivamente, o UUID do serviço e o comando que será efetuado pelo

serviço. O valor em ambas é extraido da estrutura otz_smc_cmd (Lista 6.1), configurável no REE. O cmd_type é preenchido com o conteúdo que é passado em r2 (params_stack[2]), portanto facilmente manipulável pelo atacante e representa a orientação do pedido. Esta função não apresenta deficiências que possam comprometer a segurança do mundo seguro.

Manipulando as três referidas variáveis, o atacante controla a execução da próxima função (open_session_from_ns()), também parte do suporte de comunicação presente na tarefa *idle*. Esta é responsável por abrir a sessão de comunicação, e verificando a Lista 6.5 é possível observar que esta função possui debilidades na sua implementação. Nas linhas 15, 18 e 19 é permitida a escrita em memória (inclusive segura) de três valores de forma controlada pelo atacante. Estas *write primitives* ocorrem devido à carência na verificação dos endereços de destino.

Lista 6.5: Rotina detentora de *write primitives*

```
1  int open_session_from_ns(void *param)
2  {
3     struct otz_smc_cmd *cmd = NULL;
4     cmd = (struct otz_smc_cmd*)param;
5     /* (...) */
6     ptr_svc_id = (uint32_t *)cmd->req_buf_phys;
7     ptr_task_id = (uint32_t *)cmd->resp_buf_phys;
8     /* (...) */
9     if(/* (...) */)
10    {
11       /* (...) */
12    }else{
13       if(*ptr_svc_id == OTZ_SVC_ECHO)
14       {
15          *ptr_task_id = ECHO_SS_TASK_ID;
16       }
17    }
18    cmd->ret_resp_buf_len = sizeof(int);
19    cmd->cmd_status = OTZ_STATUS_COMPLETE;
20    /* (...) */
21 }
```

O vocabulário é reduzido, pois apenas três valores diferentes podem ser escritos, tornando este conjunto de vulnerabilidades pouco flexível, mas que podem tornar-se úteis quando combinados com outras possíveis vulnerabilidades, que ainda não foram descobertas.

Outra função, cuja sua execução pode ser controlada uma vez mais através das três referidas variáveis (svc_id, cmd_id e cmd_type) é a close_session_from_ns().

A nível de segurança esta não acrescenta nenhum benefício ao atacante, apesar de possuir uma debilidade (*write primitive*). Contudo, o vocabulário não é estendido, pois o conteúdo que é escrito em memória já ocorre numa das três *write primitives* anteriormente descritas.

Por fim alcança-se a função ftz_ss_dispatcher(), cuja execução depende das variáveis svc_id, cmd_id. O seu papel já foi referido na secção 4.7.1 e, resumidamente, cria o serviço seguro com base no contexto estabelecido previamente. O atacante deve primeiramente estabelecer um contexto com o serviço que pretende atacar e só depois lhe é permitido criar o serviço seguro associado a esse contexto. A variável representativa do contexto pode ser descoberta facilmente pelo atacante manipulando o canal de comunicação. Este pode descobrir o valor da mesma, que é retornada durante a fase de abertura de sessão, através do elemento resp_buf_phys (ver Lista 6.5, linha 15). Assim sendo, é possível colocar em execução o serviço seguro desejável, neste caso o ECHO_SS. Durante a criação do ECHO_SS é enviado à função que o implementa (prvEchoSS) um parâmetro (pvParameters), nomeadamente um apontador para estrutura apresentada na Lista 6.1. Como se verifica pela Lista 6.6, são realizadas algumas inicializações às variáveis req_buf, res_buf, meta_data e svc_cmd_id (linhas 14 à 17), utilizando esse mesmo apontador. Isto significa que as variáveis são controláveis pelo atacante e além do mais, são passadas diretamente como argumento à função process_echo_svc_cmds() (linha 19).

Lista 6.6: Implementação do serviço seguro ECHO_SS

```
1  void prvEchoSS( void *pvParameters )
2  {
3     uint32_t *cmd_phy;
4     uint32_t svc_cmd_id;
5     struct otz_smc_cmd *cmd = NULL;
6     void *req_buf = NULL, *res_buf = NULL;
7     struct otzc_encode_meta *meta_data = NULL;
8
9     cmd_phy = (uint32_t *)pvParameters;
10    cmd = (struct otz_smc_cmd *)*cmd_phy;
11
12    for( ;; )
13    {
14       req_buf = (void*)cmd->req_buf_phys;
15       res_buf = (void*)cmd->resp_buf_phys;
16       meta_data = (void*)cmd->meta_data_phys;
17       svc_cmd_id = (cmd->id & 0x3ff);
18
```

```
19          process_echo_svc_cmds(svc_cmd_id, req_buf, cmd->
                req_buf_len, res_buf, cmd->resp_buf_len, meta_data, &
                cmd->ret_resp_buf_len);
20
21          vTaskDelete(NULL);
22      }
23  }
```

As variáveis req_buf e res_buf são apontadores para os *buffers* de envio e resposta dos dados. Estes são inicializadas através dos elementos req_buf_phys e resp_buf_phys nas linhas 14 e 15. A variável meta_data (linha 16), aponta para uma estrutura de dados na qual consta informação sobre a codificação do conteúdo dos *buffers*. A variável svc_cmd_id (linha 17) representa o comando que será executado pelo serviço seguro. Esta última é utilizada na função process_echo_svc_cmds() (Lista 6.7, linha 1) para executar uma rotina específica associada à funcionalidade. Caso o valor da variável svc_cmd_id seja o definido por OTZ_ECHO_CMD_SHARED_BUF, então será executada a função alvo do presente ataque, process_cmd_shared_buf().

Lista 6.7: Invocação da função vulnerável

```
1  int process_echo_svc_cmds(uint32_t svc_cmd_id, /* (...) */)
2  {
3      int ret_val;
4
5      switch (svc_cmd_id)
6      {
7          /* (...) */
8          case OTZ_ECHO_CMD_ID_SEND_CMD_ARRAY_SPACE:
9          {
10             ret_val= process_cmd_shared_buf(req_buf, req_buf_len,
                   resp_buf, res_buf_len, meta_data, ret_res_buf_len);
11             break;
12         }
13         /* (...) */
14         default:
15         /* (...) */
16     }
17 }
```

Pode ser visto na Lista 6.7 na linha 10 os parâmetros que são enviados à função process_cmd_shared_buf() (ilustrada na Lista 6.8). Esta função é responsável por descodificar os dados enviados pela aplicação cliente no Linux e por realizar o eco.

O eco é feito recorrendo duas vezes à função memcpy (linha 12 e 15), disponibilizada no T-RTOS. Na primeira invocação do memcpy (linha 12), são copiados os dados em req_buf para um *array* de caracteres (echo_data.data) armazenado em memória segura, que pode conter no máximo 1KB. Na segunda invocação do memcpy (linha 15), o conteúdo do *array* é novamente copiado para o *buffer* de resposta (resp_buf), totalmente controlável pelo atacante. Mais uma vez, é notória a falta de verificação do endereço resp_buf, pelo que não existe qualquer segurança no acesso à memória segura. O mecanismo eco, revela-se como sendo capaz de modificar até um 1KB de memória. Do ponto de vista do atacante, é possível usar o req_buf como *container* para enviar dados (i.e., instruções *assembly*) e através do resp_buf escrever este conteúdo em qualquer endereço de memória segura, pois não existe controlo de acesso.

Lista 6.8: Função vulnerável

```
1  int process_otz_echo_send_cmd_shared_buf(void *req_buf,void *
       res_buf, struct otzc_encode_meta *meta_data, /* (...) */)
2  {
3      echo_data_t echo_data;
4      char *out_buf;
5      /* (...) */
6          if(decode_data(req_buf, meta_data, (void**)&out_buf,/*
               (...) */) {
7              /* (...) */
8          }else{
9              echo_data.length = *((uint32_t*)out_buf);
10         }
11         if(decode_data(req_buf, meta_data, (void**)&out_buf,/*
               (...) */)) {/* (...) */}
12         memcpy(echo_data.data, out_buf, echo_data.length);
13         /* (...) */
14         if(decode_data(res_buf, meta_data,(void**)&out_buf,/*
               (...) */)) {/* (...) */}
15         memcpy(out_buf, echo_data.data, echo_data.length);
16         /* (...) */
17 }
```

Perante esta vulnerabilidade poderosa, o atacante pode realizar um ataque *code execution*, para ganhar controlo de execução no mundo seguro da tecnologia TrustZone.

6.2.2.2 Execução do ataque

A estratégia para ganhar controlo de execução (Figura 6.3) é simples: reescrevendo determinadas posições de memória segura com código executável é possível redirecionar o fluxo de execução para uma zona de memória não-segura onde exista uma aplicação maligna. A aplicação terá total liberdade para invocar qualquer funcionalidade disponível no mundo seguro da TrustZone, pois o estado do processador encontrar-se-á como seguro. Posto isto, o desenvolvimento do ataque foi dividido em duas fases. Uma prende-se com o desenvolvimento do *software*, tanto o que será escrito em memória segura para desencaminhar o fluxo de execução normal, como o que diz respeito à aplicação maligna. Na segunda fase, realiza-se a configuração dos registos e estrutura (Lista 6.1), através de um *driver* (desenvolvido para o efeito), sendo que estas configurações serão enviadas juntamente com a SMC, por forma a explorar a vulnerabilidade encontrada.

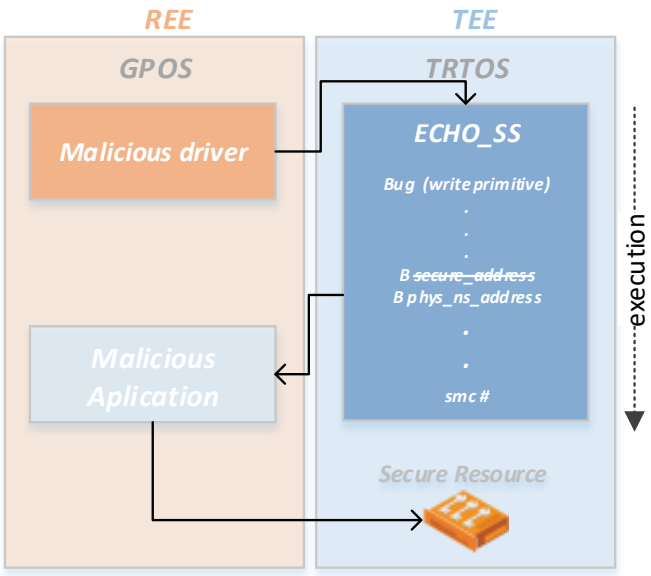


Figura 6.3: Visão geral do ataque *code execution*

Relativamente ao desenvolvimento do código que irá ganhar o controlo de execução, primeiramente será explicada a aplicação maligna. Foram utilizadas as capacidades do Linux para se codificar a aplicação diretamente na memória física. Como as aplicações no Linux executam sobre endereços lógicos (devido à presença da MMU), criar uma aplicação maligna que execute mediante esta topologia não seria uma boa abordagem, pois isso requereria que o contexto do GPOS fosse restaurado antes da invocação da instrução SMC. Essa tarefa aumentaria em demasia a complexidade do ataque. A partir do endereço 0x01000000, é escrito o código maligno, sendo que esta escolha deveu-se com o facto do endereço não estar a ser utilizado pelo GPOS (não irá comprometer o seu correto funcionamento)

e também por ser não-seguro. A aplicação é muito simples, mas suficiente para provar o conceito. Codificou-se no endereço anteriormente escolhido, a instrução mov r0, 0x10000000. O endereço colocado em r0, corresponde à função main() do T-RTOS. A nível de segurança não executa nenhuma funcionalidade crítica, mas como foi referido, servirá para provar o conceito. O *opcode* representativo desta instrução *assembly* foi escrito diretamente na memória. A ARM permite codificar determinados valores de 32-*bits* diretamente na instrução, apesar de por exemplo, o operando da instrução mov ser de 12-*bits*. Isto ocorre porque esses 12-*bits* não são encarados como um número, mas sim como um valor de 8-*bits* e a sua rotação descrita pelos restantes 4-*bits*. A próxima linha de código será equivalente a uma instrução de salto (*branch*). É necessária uma instrução que permita o salto para qualquer posição de memória endereçável por 32-bits. Por isso, utilizou-se a seguinte instrução bx, r0 (*branch and change instruction set*).

Tendo em consideração agora o código que será escrito em memória segura. Este será idêntico ao da aplicação maligna, com a pequena diferença do que endereço para onde será efetuado o salto ser 0x01000000, ou seja, o endereço onde se encontra a aplicação maligna. Resta abordar o endereço escolhido para a escrita deste pequeno código em memória segura, escolha essa que será fundamentada nos parágrafos que se seguem.

A segunda componente do ataque compreende a configuração dos registos, assim como, da estrutura principal da comunicação (Lista 6.1), enviados quando a exceção SMC é executada. Vão ser explicadas duas configurações: 1) uma para se obter o valor do contexto do serviço ECHO_SS; e 2) outra para que se explore a vulnerabilidade da função na Lista 6.8.

De maneira a que o atacante possa descobrir o valor da variável representativa do contexto, os registos de propósito geral (enviados aquando da execução da SMC) deverão possuir a seguinte configuração: 1) o registo r0 deverá possuir o valor definido por CALL_TZ_API, por forma a que o pedido seja equivocado por uma comunicação legítima, o que irá restaurar o contexto do T-RTOS e colocá-lo em execução; 2) o registo r1, contém o endereço onde a estrutura otz_smc_cmd se encontra, a configuração dos seus elementos é dada a conhecer adiante. O atacante, compreendendo toda a estrutura e a forma como esta é utilizada no TEE, manipula-a para alcançar os seus objetivos, neste caso, a execução da função responsável por abrir uma nova sessão de comunicação (Lista 6.5); por fim, 3) o registo r2 é configurado com o valor definido por OTZ_NS_TO_SEC, o qual será necessário para satisfazer a condição (na tarefa *idle*), que coloca a função desejável em execução, ainda que sejam precisas outras variáveis. Estas são respetivamente

o svc_id e o cmd_id, que são inicializadas pela rotina da Lista 6.4 através do elemento id (Lista 6.1). Este elemento, contém codificado o valor de duas variáveis distintas, uma é o UUID do serviço e outra é o comando que será executado pelo serviço. Estas foram configuradas respetivamente com o valor definido por OTZ_SVC_GLOBAL e OTZ_OPEN_SESSION. A obtenção do contexto é obtida configurando o elemento req_buf_phys (Lista 6.1) como o valor OTZ_SVC_ECHO. Isto é observável na Lista 6.5, através do teste presente na linha 13. O valor do contexto é acessível ao atacante através do elemento resp_buf_phys (Lista 6.1), podendo isso ser observado na linha 15.

Para se atacar o mundo seguro da tecnologia TrustZone a SMC é novamente invocada o r0 com o valor definido por CALL_TZ_API. A execução da tarefa *idle* é retomada. Foi manipulado o elemento id da estrutura otz_smc_cmd, para se colocar em execução a função ftz_ss_dispatcher(), através das variáveis svc_id e cmd_id. Codificou-se em id (Lista 6.1) o valor de OTZ_SVC_ECHO e OTZ_ECHO_CMD_SHARED_BUF. Por forma a criar e executar o serviço é necessário configurar o elemento context (Lista 6.1) com um valor específico (0x10), que foi extraído na primeira invocação do SMC. O serviço executa a função vulnerável (Lista 6.8) devido ao comando em cmd_id.

A primeira coisa de relevante que ocorre nesta função (Lista 6.8) é a invocação da rotina decode_data() (linha 6). Nunca havia sido feita referência à função, ainda que esta seja responsável por descodificar os dados quem vêm no *buffer*, neste caso de envio (req_buf, primeiro parâmetro), utilizando para isso a estrutura meta_data (segundo parâmetro), cujo apontador para esta é o endereço em meta_data_phys (elemento da estrutura da Lista 6.1). Da execução da função é retirado o tamanho do *buffer* de envio, que serve para inicializar a variável echo_data.length (linha 9). A estrutura meta_data foi manipulada para que o tamanho fosse duas *words* (correspondente às duas instruções). Posteriormente, a função decode_data() é novamente executada (linha 11), neste caso, é retirado o apontador para os dados propriamente ditos. O elemento meta_data foi configurado por forma a que se obtenha o endereço correto do *buffer* de dados. Na execução da função seguinte, no memcpy() (linha 12), é realizado a escrita das instruções responsáveis por alterar o fluxo de execução em memória segura (o segundo argumento, out_buf, aponta para os dados malignos). Na última invocação da função decode_data() (linha 14), é passado como argumento o *buffer* de resposta (res_buf). Importa salientar apenas, que o endereço deste *buffer* corresponde ao endereço base onde ocorrera a escrita do código maligno. A escolha do mesmo foi 0x10002420, que corresponde ao endereço do _mon_smc_handler, ou seja, endereço base do Monitor *vector table*

com um *offset* de doze *bytes*. O requisito para a escolha relativamente ao endereço do req_buf é precisamente uma posição em memória que fosse executada pelo processador. Como a rotina _mon_smc_handler é inevitavelmente executada sempre que haja um pedido de comunicação, optou-se pelo seu endereço.

Finalizando, na segunda execução da função memcpy() (linha 12) realiza-se a modificação de memória segura com o código que modificará o fluxo normal de execução do processador. Após o ECHO_SS terminar, será invocada uma exceção SMC de retorno. Quando isso ocorrer, ao invés de ser executada a rotina _mon_smc_handler, o processador irá começar a executar a aplicação maligna. Neste momento, o ataque realizado é considerado um sucesso e o atacante tem o poder de executar qualquer funcionalidade do TEE (neste caso é invocada a função main() do TRTOS).

6.2.3 Proteção do COMON

O ataque desenvolvido na secção anterior foi suficiente para provar o conceito de proteção baseada em monitorização proporcionado pelo COMON, por um motivo: o ataque altera o fluxo de execução associado à comunicação no TEE e, assim sendo, o comportamento do canal de comunicação que é inerente ao ataque não se enquadra no padrão comportamental realizado pelo COMON para o respetivo par de aplicações ECHO.

Como referido na secção 5.5.1, o COMON protege o sistema contra ataques que exploram *bugs* no código e consequentemente alteram o comportamento normal do canal. Antes do *deployment* do sistema, o COMON molda o comportamento da comunicação referente ao par de aplicações eco (ECHO_CLI e ECHO_SS). Este período de adaptação denomina-se *learning* e possui uma duração limitada, tipícamente curta. Isto é, a aplicação ECHO_CLI, responsável por invocar o ECHO_SS, é executada um número reduzido de vezes durante este período. A realização desta fase adaptativa requer, obviamente, que o COMON esteja devidamente configurado e habilitado, como foi explicado na secção 5.5.1. Assim sendo, a aplicação cliente foi executada um certo número (N) de vezes (especificado em *hardware*) de forma manual, ou seja, recorrendo à consola do Linux, N vezes. Este procedimento poderia ser facilmente realizado de forma automática através de uma aplicação em *software* de suporte ao COMON.

Durante a execução da aplicação ECHO_CLI, sempre que esta invoca o mundo seguro para comunicar, o COMON deteta e começa a elaborar o perfil dessa fase da comunicação. Quando o período de *learning* termina, o COMON realiza um padrão comportamental a duas dimensões.

Quando o ataque *code execution* é realizado, o primeiro pedido manipulado pelo atacante para a obtenção do valor do contexto associado ao serviço seguro ECHO_SS não é detetado pelo COMON. Apesar de ser um pedido maligno e provocar escritas arbitrárias em memória segura (se desejável pelo atacante), o periférico não identifica nenhuma irregularidade pois o perfil temporal que descreve este pedido não difere do que foi construído durante o período de *learning*. No entanto, o segundo pedido, que é responsável por tomar assalto o mundo seguro, é facilmente detetado, pois o perfil temporal deste pedido irá ser bastante diferente daquele que foi criado. O objetivo desta invocação é precisamente alterar o fluxo normal de execução. Quando o código maligno começa em execução, este é logo interrompido pelo COMON, através de uma interrupção FIQ que este gera e envia ao processador. A execução volta a ficar na posse do T-RTOS, que irá interpretar a mensagem e proceder em benefício de sua segurança.

O COMON, através de monitorização em tempo-real, proporciona ao sistema uma capacidade de reação mesmo quando este se encontra sobre alçada de uma intrusão. O perfil comportamental do canal de comunicação que é construído pelo COMON é preciso, muito devido à precisão do componente PMU mas também, por causa das informações exclusivas fornecidas pelo componente PTM. A nível de segurança esta característica provou que a proteção fornecida pelo COMON é fidedigna e fiável, e que pode ser usada como complemento de outros mecanismos de segurança. Aliás, a monitorização sempre foi encarada pelo autor como uma camada complementar da camada de *trust*.

6.3 Análise de Segurança

Nesta secção é feita uma análise das propriedades de segurança concedidas pela solução L-TEE e COMON. A análise é realizada através da enumeração das garantias de segurança fornecidas pelo sistema segundo o modelo CIA.

O L-TEE e o COMON contribuem parcialmente, para alcançar os três fundamentos CIA:

- **Confidencialidade** é a capacidade em restringir o acesso aos dados a apenas quem possua autorização. A confidencialidade é parcialmente garantida através dos mecanismos de isolamento espacial da tecnologia TrustZone. Qualquer acesso, não autorizado, por parte do GPOS a um segmento de memória alocado pelo T-RTOS será bloqueado pelo componente TZASC.

Além disso, o GPOS não consegue aceder a dispositivos atribuídos ao T-RTOS, pois tal acesso é igualmente bloqueado. Relativamente à comutação entre mundos, as características da MMU e da *cache* previnem parcialmente a fuga de informação pertencentes ao mundo seguro. Infelizmente, os ataques *side-channel* podem, de forma indireta, prever informação confidencial. O canal de comunicação é, sem dúvida, o maior problema de segurança da TrustZone. Este não fornece mecanismos de autenticação, o que permite a realização de ataques que manipulam e intersetam as mensagens transferidas no mesmo. O COMON introduz uma camada de proteção ao canal e restringe o acesso ao mesmo na presença de certos ataques que exploram esta vulnerabilidade, assegurando por sua vez, a confidencialidade da informação;

- **Integridade** assegura confiabilidade dos dados e do sistema durante o seu funcionamento. A integridade durante o processo de inicialização do sistema pode ser assegurada através de mecanismos tradicionais como o *secure boot*. Uma vez inicializado, a tecnologia TrusZone não fornece qualquer mecanismo em *hardware* ou *software* que assegure a integridade do sistema ao longo do tempo. Não obstante, o COMON monitoriza o canal de comunicação durante a vida útil do sistema, sendo capaz de intersetar intrusões que, inclusive, poderiam ganhar o controlo do mundo seguro da tecnologia TrustZone. Assim sendo, o COMON consegue assegurar a integridade parcial do sistema perante alguns ataques em *software*;

- **Disponibilidade** refere-se ao acesso à informação, sempre que intervenientes autorizados desejem. O T-RTOS executa a pedido de uma aplicação cliente que deseje usufruir das suas funcionalidades seguras. Existe um *overhead* para que seja disponibilizado o serviço, no entanto esse é perfeitamente adequado a sistemas de propósito geral. Ataques que tentam perturbar a execução do T-RTOS, estilo DoS, são condicionados devido à presença do COMON. Este componente, caso se encontre a monitorizar o canal de comunicação associado à respetiva aplicação, deteta e bloqueia este tipo ataques;

Conclusões e Perspetivas

No presente capítulo é realizada uma apreciação sobre o trabalho desenvolvido. Na secção 7.1 são referidas as principais conclusões e, seguidamente na secção 7.2, são abordadas algumas limitações, assim como, o trabalho que seria produtivo realizar no futuro por forma a aprimorar a solução de segurança.

7.1 Conclusões

A dissertação cumpriu os objetivos a que se propôs na secção 1.2 aquando do seu início. Fundamentalmente definiu-se com sucesso o espetro de segurança da tecnologia TrustZone e desenvolveram-se mecanismos de monitorização em *hardware* capazes de aumentar o espetro de segurança da mesma. A solução desenvolvida não é perfeita, esta apenas limita o raio de ação do atacante, nomeadamente quando este vislumbra oportunidades para realizar ataques que pretendam adquirir controlo de execução no mundo seguro. O COMON é portador de algumas limitações (enumeradas na próxima subsecção), muito devido a escolhas arquiteturais que condicionam as suas capacidades. Uma mais profunda investigação com foco em outras tecnologias ou componentes úteis, pode melhorar a solução que foi desenvolvida e torna-la num mecanismo realmente poderoso.

Para finalizar, com a cada vez maior sofisticação dos ataques aos sistemas embebidos, os mecanismos de autenticação relativos ao canal de comunicação não serão suficientes para proteger eficientemente os recursos do mundo seguro. É aceitável que, na altura do seu lançamento, a tecnologia TrustZone não fornecesse suporte direto para a proteção ao canal de comunicação. Contudo, os recentes avanços em torno do *machine learning*, justifica a presença de algum tipo de suporte para o fornecimento de segurança através da monitorização. Os problemas em torno do canal de comunicação são reconhecidos em torno da comunidade científica e atacantes, estes são sem margem de dúvida, o calcanhar de Aquiles da tecnologia TrustZone.

7.2 Limitações e Trabalho Futuro

No decorrer da presente dissertação, o autor deparou-se com várias limitações relativamente ao desenvolvimento da solução de segurança. No meio académico, o tema da monitorização em segurança carece ainda de consolidação e de informação fidedigna, o que implicou um rigoroso cuidado na escolha das ideologias adotadas pelo autor no projeto. Um dos conceitos bem estabelecidos na área de segurança é que nenhuma solução de segurança é totalmente segura, e como tal, a solução de segurança desenvolvida também possui deficiências. A mesma está vulnerável a ataques que consigam modificar o conteúdo dos registos de configuração do COMON. Como trabalho futuro, poderiam ser integrados mecanismos em *hardware* no próprio COMON ou até reutilizadas tecnologias disponíveis na própria plataforma, de modo a conceder proteção face a estes casos específicos.

Na ótica do autor, a maior limitação que restringiu o desenvolvimento foi o facto da solução adotar o principio não intrusivo, ainda que as razões para a sua escolha tenham sido explicadas na secção 5.3. A arquitetura ARM não concede acesso ao barramento de instruções, apenas o permite de forma intrusiva através de instruções próprias. Devido a esta característica, a solução apenas consegue recolher informações de forma indireta, o que condicionou muito o desenvolvimento de funcionalidades. Como consequência, a solução de segurança: 1) não é escalável ao nível da monitorização, isto é, apenas consegue proteger o canal de comunicação associado a um serviço seguro de cada vez; 2) é dependente da arquitetura da comunicação, no sentido em que o COMON possui a necessidade de verificar, através de um endereço específico, o fecho de uma sessão de comunicação. Esta necessidade pode ser explorada por atacantes, e ao mesmo tempo, retira flexibilidade à solução; 3) existem vários fatores que fazem variar o comportamento do canal (i.e., tamanho dos dados ou a utilização diferentes mecanismos para fornecer a mesma funcionalidade). Sem acesso ao barramento de instruções não é possível, de maneira alguma, transmitir essa informação ao COMON, sendo que esse fator restringiu o desenvolvimento de mecanismos de proteção mais eficientes. Por forma a contornar estas limitações, poderia ser realizada uma investigação que permitisse concluir com efetividade se existe alguma forma de aceder ao barramento de instruções de forma não intrusiva. Tal capacidade poderia ser utilizada pelo COMON para proteção de múltiplos serviços, visto que a própria instrução SMC possui um campo identificador de 4-*bits* codificado diretamente no seu *opcode*. Esta característica poderia ser usada para identificar diferentes serviços em execução (paralela ou não). Sobre a característica da SMC, o COMON poderia desenvolver diferentes mecanismos de proteção, mais eficazes, bem como deixar de

ser dependente da arquitetura da comunicação.

A segurança através da monitorização abre novas portas relativamente á forma de encarar a segurança. A monitorização permite a adaptação de um sistema ao meio envolvente, através da acumulação contínua de informação. Desta forma, o sistema torna-se dotado da capacidade, de tomar decisões de forma autónoma e em *run-time*. O *machine learning* é uma àrea vasta e adequa-se a soluções baseadas em monitorização. Na ótica do autor, a segurança em sistemas embebidos carece em muito de abordagens baseadas em *machine learning*, sendo que o tópico que resulta da intercepção de ambas as áreas, possui enorme potencial de investigação. Um estudo aprofundado deste tópico poderia permitir a adaptação de novos mecanismos de segurança que complementam a solução desenvolvida - o COMON.

Referências Bibliográficas

[1] J. Naisbitt and J. Cracknell, *Megatrends: Ten new directions transforming our lives*. Warner Books New York, 1984.

[2] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security and Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[3] D. Papp, Z. Ma, and L. Buttyan, "Embedded Systems Security : Threats , Vulnerabilities , and Attack Taxonomy," pp. 145–152, 2015.

[4] M. Janke, "Attacks on Embedded Devices,"

[5] ARM, "ARM Security Technology. Building a Secure System using TrustZone Technology ARM," *ARM white Pap.*, p. 108, 2009.

[6] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, "Hardware trojan attacks: threat analysis and countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.

[7] A. Gargantini, E. Riccobene, and P. Scandurra, "Model-driven design and asm-based validation of embedded systems," *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, pp. 24–54, 2009.

[8] P. Koopman, A. Smailagic, P. Steenkiste, D. E. Thomas, C. Wang, H. Choset, R. Gandhi, B. Krogh, D. Marculescu, P. Narasimhan, J. M. Paul, R. Rajkumar, and D. Siewiorek, "Undergraduate embedded system education at carnegie mellon," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 3, pp. 500–528, 2005.

[9] W. E. Guide, "An Executive Guide to Cyber Security for Operational Technology," pp. 1–30.

[10] S. Bhunia, M. Abramovici, D. Agrawal, P. Bradley, M. S. Hsiao, J. Plusquellic, and M. Tehranipoor, "Protection against hardware trojan attacks: Towards a comprehensive solution," *IEEE Design & Test*, vol. 30, no. 3, pp. 6–17, 2013.

[11] P. J. Sousa and M. P. Correia, *Segurança no Software*. Lisboa, Portugal: FCA, 2010.

[12] S. Sharma, "Embedded systems–a security paradigm for pervasive computing," in *Communication Systems and Network Technologies (CSNT), 2013 International Conference on*, pp. 472–477, IEEE, 2013.

[13] S. Parameswaran and T. Wolf, "Embedded systems security—an overview," *Design Automation for Embedded Systems*, vol. 12, no. 3, pp. 173–183, 2008.

[14] S. Pinto, D. Oliveira, J. Pereira, J. Cabral, and A. Tavares, "Freetee: When real-time and security meet," in *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*, pp. 1–4, IEEE, 2015.

[15] "Sierratee for arm® trustzone®." `https://www.sierraware.com/open-source-ARM-TrustZone.html`. Accessed: 2003-07-20.

[16] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment.," in *NDSS*, 2015.

[17] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *Proceedings of the 25th USENIX Security Symposium*, pp. 549–564, 2016.

[18] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Truspy: Cache side-channel information leakage from the secure world on arm devices,"

[19] D. Rosenberg, "Unlocking the motorola bootloader," *Azimuth Security Blog*, 2013.

[20] D. Rosenberg, "Qsee trustzone kernel integer over flow vulnerability," in *Black Hat conference*, 2014.

[21] "Here be dragons: Vulnerabilities in trustzone." `http://atredispartners.blogspot.pt/2014/08/here-be-dragons-vulnerabilities-in.html`. Accessed: 2017-03-01.

[22] "Bits, please!." `http://bits-please.blogspot.pt/2016/06/extracting-qualcomms-keymaster-keys.html`. Accessed: 2017-02-11.

[23] K. Lady, "Sixty Percent of Enterprise Android Phones Affected by Critical QSEE Vulnerability."

[24] Xilinx, *Zynq-7000 All Programmable SoC*, vol. 1025. 2013.