Hardware support for the security analysis of embedded softwares : applications on information flow control and malware analysis

Muhammad Abdul Wahab

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

CentraleSupélec

THESE DE DOCTORAT DE

CENTRALESUPELEC
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Electronique*

Par

Muhammad Abdul WAHAB

Support matériel pour l'analyse de sécurité du comportement des applications

Application au contrôle de flux d'information et à l'analyse des logiciels malveillants

Thèse présentée et soutenue à Rennes, le 10 décembre 2018
Unité de recherche : IETR, UMR 6164
Thèse N° : 2018-08-TH

Rapporteurs avant soutenance :

Pascal Benoit	Maître de Conférences Université de Montpellier 2
Jean-Luc Danger	Directeur des études Télécom ParisTech

Composition du Jury :

Laurence Pierre Université de Grenoble	Professeur Présidente
Pascal Benoit Université de Montpellier 2	Maître de Conférences
Jean-Luc Danger Télécom ParisTech	Directeur des études
Guillaume Hiet CentraleSupélec, campus de Rennes	Maître de Conférences
Pascal Cotret	Ingénieur R&D

Directeur de thèse

Christophe Moy Université de Rennes 1	Professeur

Invité(s)

Guy Gogniat Université de Bretagne-Sud	Professeur

I would like to dedicate this thesis to my loving parents and all my family members ...

Acknowledgments

In the name of Allah the most Merciful and Beneficient.

First and Foremost praise is to Allah, the Almighty, the greatest of all, on whom ultimately we depend on sustenance and guidance. I would like to thank Allah azza wa jal for giving me the opportunity, determination, and strength to do my research. His continuous grace and mercy was with me throughout my life and even more during the tenure of my research.

Prophet Muhammad, peace and blessings of Allah be upon him, said, "He who does not thank people, does not thank Allah" (Ahmad, Tirmidhi). I would like to express my sincere and deep gratitude to Dr.-Ing. Pascal Cotret for his continuous support, guidance, and encouragement. In addition to being an excellent supervisor, he is an extremely hardworking researcher. I tried my best to follow his example but I could not meet half of his dedication. I appreciate all his contributions of time, support, and ideas.

I would also like to express my gratitude to Prof. Guillaume Hiet, who helped me a lot throughout this work especially the last year. His methodology and way of guidance helped me carry on this work in the right direction during tough times. I also appreciate the support of Christophe MOY who played an important part throughout this thesis and a special thanks for his instructive reviews and guidance on this manuscript. I am extremely thankful to Prof. Guy Gogniat and Prof. Vianney Lapotre for their valuable guidance throughout this work. Their experience and feedback helped me a lot and I cannot thank them enough for their support.

I would like to thank my friend Mounir Nasr Allah for his help, support, and important feedback especially during our tea-break discussions. Our short talks were very valuable to me. I would like to thank my current and previous teammates Rami Othman, Vincent Gouldieff, Quentin Bodinier, Rémi Bonnefoi, Malek Naoues, Lilian Besson, Ali Cheaito, Hussein Chour, Majed Saad, Julio Cesar Manco Vasquez, Salah Eddine Kabbour, Xiguang Wu, Yifan Zhou, Abir Amri, Mouna Ben Mabrouk, Marwa Chafii, and all fellows in CentraleSupélec-Rennes for their help and kindness. A special thanks to Haifa Fares for her kind advice that was very helpful.

I would also like to extend my gratitude to the SCEE members: Amor Nafkha, Jacques Palicot, Carlos Bader, Yves Louet and all professors of CentraleSupélec, Rennes. I extend

my sincere thanks to Karine Bernard and Jeannine Hardy for their administrative support. I am very grateful to Professor Alberto Ferrante for his guidance during the mobility. I am very thankful to Olivier Deforges and Wassim Hamidouche from VAADER team to allow me to finish my manuscript in great conditions. I am also very thankful to the jury members of my thesis committee for reviewing this work.

Finally, I would like to thank my family for their unconditional support and sacrifices. A special thanks to my mother (Nusrat Parveen) and father (Muhammad Anwar) who have always encouraged me to engage in higher and better education. All my sisters played an important part in my education and my brother, my role model, inspired me with his dedication and hard work. I thank all my relatives and friends who contributed to my well-being and this work.

Résumé

La sécurité dans les systèmes embarqués devient un sujet de plus en plus important, notamment en raison de la multiplication des appareils portables capables de gérer des données sensibles (smartphones, tablettes, etc.). L'utilisation d'un support matériel pour implémenter des mécanismes de sécurité paraît incontournable afin de protéger efficacement ces systèmes contre les attaques logicielles, tout en minimisant l'impact sur le temps d'exécution des applications. Le contrôle de flux d'informations, *Dynamic Information Flow Tracking (DIFT)*, permet de détecter différents types d'attaques logicielles tels que les dépassements de tampon ou les injections SQL. Pour suivre et contrôler les flux d'informations, les travaux existants modifient l'architecture interne du processeur ou utilisent des processeurs de type *softcore*, c'est-à-dire des processeurs synthétisés sur un FPGA, tels que les Leon3. De plus, la plupart des approches existantes ne sont pas portables sur des processeurs de type *hardcore* (câblés dans une puce). Dans cette thèse, une solution ciblant les processeurs *hardcore* ARM Cortex-A9 est proposée. Notre approche s'appuie sur des composants ARM CoreSight, qui permettent de tracer l'exécution des programmes exécutés par le processeur, afin de réaliser le contrôle de flux d'informations. Le co-processeur DIFT que nous proposons est réalisé dans la partie FPGA Artix-7 du système sur puce (Soc) Zynq, afin de bénéficier de la présence sur la même puce de cœurs ARM Cortex-A9 dans la partie PS (*Processing System*) et d'une partie reconfigurable FPGA dans la partie PL (*Programmable Logic*). Il est montré que l'utilisation des composants ARM CoreSight n'ajoute pas de surcoût en terme de temps d'exécution et permet une amélioration du temps de communication entre le processeur ARM et le coprocesseur DIFT. De plus, les flux d'informations induits par les appels système ne sont pas pris en compte dans les travaux existants, ce qui les rend vulnérables aux attaques provenant de fichiers tels que la fuite d'information contenue dans un fichier. La solution proposée dans ces travaux utilise RFBlare, un noyau Linux modifié, qui se charge du stockage et de la récupération des étiquettes associées aux fichiers afin de protéger contre ces attaques.

Abstract

Embedded security is more and more crucial with the huge increase in the number of mobile equipment dealing with sensible data (smartphones, tablets, etc.). Runtime efficient protection from software attacks requires architectural support. DIFT (*Dynamic Information Flow Tracking*) is a technique that allows detecting software attacks such as buffer overflows or SQL injection by monitoring information flows. In order to track information flows, existing works modify internal CPU architecture or target softcore CPUs. Most of the existing approaches are not portable to hardcore CPUs. In this thesis, a solution that targets ARM Cortex-A9 hardcore CPUs and takes advantage of ARM CoreSight components, that allow tracing the CPU, is proposed in order to efficiently track information flows. DIFT co-processor is implemented in Artix-7 FPGA included in Zynq SoC in this work, taking advantage of both PS (*Processing System* containing ARM Cortex-A9 cores) and PL (*Programmable Logic* providing an FPGA) on the same chip. It is shown that ARM CoreSight components do not add execution time overhead which results in an improvement of communication time overhead between the ARM CPU and the DIFT coprocessor. In addition, information flows induced by system calls are not considered in existing works making them unsafe to attacks originating from files such as data leaks. The solution proposed in this thesis takes advantage of RFBlare, a modified Linux kernel, that provides support to store and retrieve tags associated to files in order to protect against these attacks.

Résumé étendu

Depuis plus d'une décennie, le nombre de vulnérabilités logicielles ne cesse d'augmenter. De nombreuses attaques logicielles exploitent les mécanismes architecturaux de bas niveau. Dans le contexte de la sécurité informatique, le contrôle de flux d'informations (*Dynamic Information Flow Tracking* en anglais) est un mécanisme de sécurité qui permet de lutter efficacement contre un large spectre d'attaques logicielles, par exemple celles exploitant les vulnérabilités de type *buffer overflow*, les *cross-site scripting*, les injections SQL ou les fuites de données en générale. Le contrôle de flux d'informations consiste à rajouter une étiquette (tag ou *taint* en anglais) sur chaque conteneur d'information. Le conteneur d'information varie suivant le niveau auquel on effectue le contrôle: par exemple, il peut s'agir d'un registre ou d'une adresse mémoire si on s'intéresse au niveau architectural, d'une variable si on s'intéresse au niveau d'une application ou d'un fichier au niveau d'un système d'exploitation. A chaque fois qu'une instruction est exécutée par le processeur, l'étiquette est également propagée entre les conteneurs d'information. Une politique de sécurité permet de préciser à quel moment l'étiquette doit être vérifiée afin de détecter une attaque. Par exemple, si on considère une attaque qui consiste à lire une zone/adresse mémoire non initialisée, la politique de sécurité consiste à détecter qu'il y a une écriture sur une adresse avant toute lecture de la même adresse. La détection de l'écriture peut se faire en vérifiant la valeur de l'étiquette. On peut considérer que lors de chaque écriture d'une adresse mémoire, son étiquette est mise à 1. Lorsque l'adresse mémoire est adressée pour une lecture, son étiquette est vérifiée pour savoir si elle a été initialisée (étiquette 1) ou non (étiquette 0).

Le contrôle de flux d'information peut être réalisé de plusieurs façons : de manière logicielle, matérielle ou en utilisant une solution hybride logicielle/matérielle. L'approche logicielle est la plus flexible, c'est-à-dire qu'elle permet de protéger contre de multiples attaques, mais en contrepartie elle ralentit l'application de 3 à 37 fois [1]. L'approche matérielle permet de réduire l'impact sur le temps d'exécution de l'application mais le spectre des attaques détectées est plus restreint et surtout il est figé dans le temps. L'approche hybride ([2, 3, 4, 5]) permet d'obtenir le meilleur compromis entre performance du système et flexibilité. Nos travaux se basent sur cette approche hybride et permettent de répondre aux questions suivantes qui sont non-traitées dans les travaux existants :

- Comment contrôler les flux d'informations sur un processeur ARM *hardcore* (c'est-à-dire un processeur dont l'architecture ne peut être modifiée)
- Comment s'assurer de la sécurité du coprocesseur ?
- Comment mettre en œuvre plusieurs politiques de sécurité de différentes granularités ?
- Comment protéger une application contenant plusieurs fils d'exécution (*threads*) ?

Pour pouvoir contrôler les flux d'informations, il faut déterminer tous les flux qui sont effectués lors de chaque opération exécutée par le processeur: chargement, mémorisation, affectation, opérations arithmétiques, etc. Dans les travaux existants [1, 2, 3, 4, 5], les processeurs qui exécutent l'application sont des processeurs de type *softcore* (c'est-à-dire des processeurs réalisés dans un FPGA) qui permettent de récupérer l'information nécessaire pour suivre et contrôler les flux d'information en utilisant des signaux existants. Sur un processeur de type *hardcore* (câblé dans une puce donc non modifiable), il n'est pas possible d'utiliser cette approche. Dans nos travaux, nous proposons une approche utilisant à la fois le mécanisme de trace offert par les composants ARM CoreSight, une analyse statique, l'instrumentation du code de l'application et RFBlare [6] (un noyau Linux modifié) afin de récupérer l'information et de la transférer à un co-processeur implémenté sur la partie reconfigurable d'un SoC (système sur puce) Zynq. Les composants CoreSight du cœur ARM sont utilisés pour déterminer le chemin d'exécution pris par l'application. En effet, à l'aide des traces générées par ces composants, il est possible de déterminer quel bloc de base le processeur est en train d'exécuter. En revanche, ces traces ne permettent pas d'inférer directement les flux d'informations générés à l'intérieur d'un bloc de base. C'est pourquoi une analyse statique est réalisée en complément lors de la compilation de l'application, pour déterminer tous les flux d'information à l'intérieur de chaque bloc de base. Le résultat de cette analyse est stocké dans le fichier exécutable de l'application et fourni au co-processeur lors du chargement de l'application. Toutefois, l'analyse statique ne peut déterminer toutes les adresses mémoires des instructions mémoires (par exemple, `ldr` et `str`). Aussi, lors de la compilation, il est également nécessaire d'instrumenter le code afin de récupérer les adresses mémoires des instructions `ldr` et `str` à l'exécution de l'application. RFBlare est quant à lui nécessaire pour pouvoir récupérer et mettre à jour les étiquettes associées à des fichiers. Il a été vu d'une part qu'en utilisant l'approche proposée, qu'il est possible de déterminer tous les flux d'information. D'autre part, l'approche proposé dans ces travaux est meilleure en termes du temps d'exécution que l'approche proposée dans l'état de l'art [4].

Le composant CoreSight PTM, disponible sur le SoC Zynq, génère la trace (autrement dit, des paquets qui permettent d'avoir des informations sur le code exécuté par le processeur). Cette trace peut être envoyée vers la partie reconfigurable du SoC (*système sur une puce*) en utilisant l'interface EMIO (*Extended Multiplexed Input/Output* en anglais). Le support pour les composants CoreSight n'existait pas pour le Zynq SoC dans le noyau Linux. Nous avons donc modifié le *device tree* pour pouvoir utiliser les pilotes associés à ces composants.

Par ailleurs, le pilote Linux existant ne mettait pas en œuvre toutes les caractéristiques offertes par ces composants: par exemple, il n'était pas possible de récupérer l'adresse des branchements directs. Nous avons donc également modifié le pilote des composants CoreSight pour rajouter ce support. Nous avons soumis un patch au mainteneur. Ce patch a été accepté et il est maintenant intégré dans le noyau Linux. Le pilote pour le TPIU (*Trace Port Input/Output Unit*) n'était pas complet et nous l'avons également modifié pour pouvoir utiliser ce composant afin de récupérer la trace sur la partie reconfigurable (FPGA) du SoC.

Pour effectuer l'analyse statique, un algorithme qui permet d'analyser chaque bloc de base d'un graphe de flot de contrôle a été proposé. La sémantique de chaque instruction exécutée par le processeur est analysée pour déterminer les flux d'informations générés par cette instruction. Par exemple, lorsque le processeur exécute une instruction `add r0, r1, r2`, il y a un flux d'information depuis les registres `r1` et `r2` vers le registre `r0`. L'analyse statique permet de récupérer cette information pour l'ensemble des instructions d'un bloc de base. Il faut également arriver à déterminer les adresses mémoires utilisées dans le programme. Or, cela n'est pas possible pour certaines instructions notamment pour les instructions mémoire (`ldr`, `str`, ...) qui sont relatives aux registres. Pour ces instructions, il faut instrumenter le code c'est-à-dire rajouter des instructions dans le code pour récupérer l'information sur l'adresse mémoire. L'analyse statique est réalisée en utilisant le désassembleur Capstone [7] qui fournit une API (*Application Programming Interface*) pour différentes architectures. Le chapitre 3 détaille l'utilisation des composants CoreSight, l'analyse statique et l'instrumentation pour pouvoir récupérer l'information nécessaire pour le contrôle de flux d'informations.

Un premier prototype a été réalisé et a permis d'évaluer la faisabilité de l'approche proposée. Le prototype permet de montrer comment le contrôle de flux d'information peut être réalisé pour protéger le système contre la fuite de données. Dans un premier temps, un processeur générique est utilisé pour effectuer les opérations nécessaires pour le contrôle de flux d'information. Le chapitre 4 détaille l'architecture réalisée.

Dans un deuxième temps, un coprocesseur dédié a été développé afin d'accélérer le contrôle de flux d'information. Les coprocesseurs dédiés de l'état de l'art considèrent que l'application n'exécute qu'un seul thread. Or, les applications utilisent de plus en plus toutes les ressources disponibles sur un processeur afin d'accélérer l'exécution. C'est pourquoi, il est important de s'assurer que les applications qui utilisent plusieurs threads soient également protégées des attaques logicielles. L'approche proposée a été améliorée pour pouvoir récupérer les informations nécessaires pour suivre les flux d'informations d'une application contenant plusieurs threads. Une configuration différente des composants CoreSight a été utilisée pour obtenir le *context ID* d'une application dans les traces. Le *context ID* inclut le ASID (*Application Specific ID*) et le TID (*Thread ID*). A l'aide de ces informations, le coprocesseur peut identifier le thread actuellement exécuté et donc réaliser le contrôle de flux d'information pour chaque tread. Par ailleurs, l'architecture du coprocesseur dédié que nous avons proposé

permet de découpler les deux opérations qu'il doit effectuer : reconstruire le chemin d'exécution de l'application et suivre les flux d'informations à l'aide des instructions générées par l'analyse statique. Le chapitre 5 détaille l'architecture du coprocesseur et comment l'utiliser pour protéger plusieurs threads ou suivre plusieurs politiques de sécurité en parallèle.

Par ailleurs, une mobilité a été effectuée entre décembre 2017 et février 2018 au sein du laboratoire ALARI à l'Université de la Suisse Italienne (à Lugano). Au cours de cette mobilité, la détection dynamique de logiciel malveillants (*malware*) pour les appareils portables à l'aide des accélérateurs matérielles a été étudiée. Avec l'augmentation du volume d'appareils intelligents, l'intérêt d'attaquer ces appareils est fortement en croissance, faisant de leur sécurité un défis majeur. Le vecteur d'attaque couramment utilisé pour détourner ces appareils de leur usage est l'utilisation d'un logiciel malveillant. Les deux moyens de protection contre ces logiciels malveillants sont l'analyse statique et l'analyse dynamique. Alors que les analyses statiques couramment utilisées s'appuient sur des signatures et ne peuvent détecter que les attaques connues, l'analyse dynamique nécessite des ressources de calcul qui ne conviennent pas aux appareils portables fonctionnant sur batterie. Une solution qui utilise les FPGAs embarqués dans les SoCs a été proposée afin de réaliser un calcul économe en énergie pour l'analyse dynamique, ce qui permet de l'adopter pour les appareils portables. La solution proposée peut également accélérer l'analyse dynamique grâce aux accélérateurs matériels générés à l'aide d'un outil de synthèse de haut niveau. Plus précisément, trois algorithmes de classification (la régression logistique, la classification naïve bayésienne et les machines à vecteur de support) ont été réalisés en matériel à l'aide de Vivado HLS. L'efficacité en termes du temps d'exécution et de la consommation énergétique ont été évalués. Le chapitre 6 détaille ces travaux.

Le chapitre 7 résume les travaux présentés dans ce manuscrit et décrit les perspectives futures. Il a notamment été montré dans ce manuscrit comment le contrôle de flux d'informations peut être réalisé sur une architecture ARM en utilisant les composants CoreSight, l'analyse statique, l'instrumentation et RFBlare (un noyau Linux modifié). L'architecture réalisée permet de protéger des applications contenant plusieurs threads et permet également de mettre en œuvre plusieurs politiques de sécurité en utilisant l'architecture modulaire du coprocesseur proposé. Par ailleurs, une version économe en consommation énergétique a été développée pour trois algorithmes de classification afin de détecter des applications malveillantes et d'augmenter la durée d'utilisation de la batterie des smartphones. En ce qui concerne le contrôle de flux d'information, les perspectives futures consistent en l'utilisation de l'architecture afin de détecter différentes attaques logicielles (tels que XSS (*cross-Site Scripting*) et attaque par traversée de chemin) et l'utilisation de l'approche proposée sur des processeurs Intel qui possèdent également un composant de debug similaire au composant ARM CoreSight PTM notamment Intel PT (*Processor Trace*).

Contents

Nomenclature

Acronymes et Abréviations

ABI	Application Binary Interface
ALU	Arithmetic and Logical Unit
ASIC	Application Specific Integrated Circuit
ASLR	Address Space Layout Randomization
AXI	Advanced eXtensible Interface
BSP	Board Support Package
CFG	Control Flow Graph
CLB	Configurable Logic Block
CPU	Central Processing Unit
CS	CoreSight
CTI	Cross Trigger Interface
CTM	Cross Trigger Matrix
DAP	Debug Access Port
DEP	Data Execution Prevention
DIFC	Dynamic Information Flow Control
DIFT	Dynamic Information Flow Tracking
DTB	Device Tree Blob
DTS	Device Tree Script
ECT	Embedded Cross Trigger
ELF	Executable and Linkable Format
EMIO	Extended Multiplexed Input Output
ETB	Embedded Trace Buffer

ETM	Event Trace Macrocell
FF	Flip Flop
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
FTM	Fabric Trace Macrocell
HDL	Hardware Description Language
HLS	High Level Synthesis
IP	Intellectual Property
ISA	Instruction Set Architecture
ITM	Instruction Trace Macrocell
LLVM	Low-Level Virtual Machine
LR	Logistic Regression
LUT	LookUp Table
MIO	Multiplexed Input Output
MMU	Memory Management Unit
NB	Naive Bayes
OCM	On-Chip Memory
OS	Operating System
PFT	Program Flow Trace
PL	Programmable Logic
PS	Processing System
PTE	Page Table Entry
PTM	Program Trace Macrocell
ROP	Return-Oriented Programming
SDK	Software Development Kit
SoC	System on Chip
SVM	Support Vector Machine

syscall	System Call
TCR	Tag Check Register
TMC	Tag Management Core
TMMU	Tag Memory Management Unit
TPIU	Trace Port Interface unit
TPR	Tag Propagation Register
VHDL	Very High Speed Integrated Circuit Hardware Description Language

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In this chapter, background context and motivations for this work are described. Then, the important notion of security policy is explained with the help of an uninitialized memory attack. A non-exhaustive list of software attacks is presented in order to understand the type of attacks that can be detected using information flow tracking.

Contents

1.1 Background and Motivations

Software security is still one of the main concerns in today's systems even though an important amount of research has been done on the subject. There is no universal solution for software security because the security has been overlooked for performance from the dawn of the computer age. From the invention of transistor-based computers, the focus of research and development has been put on the improvement of speed rather than the security of computers. This leads to devices considering security as an optional feature rather than a compulsory one. In the 1980s, the first impactful software attacks showed the need for making research efforts in the domain of software security.

Since the beginning of this field, most of the solutions proposed in research are not being widely adopted by hardware vendors and software developers and there are a few reasons for that. One of the main reason is the scope of existing works. Most related works provide

an answer to a particular type of software attacks. There are a huge variety of attacks that target different aspects: some target a particular aspect of the hardware while others take advantage of a software misconception. In average, 40 new software vulnerabilities have been detected every single day of 2017 [8]. In addition to the different level of attacks, there has been a diversification of targeted devices. The attacks are no longer limited to the personal computer but can affect all the connected devices. For all these major concerns, there is an important need of providing architectures that provide security features.

This paragraph briefly discusses important software security notions. First of all, there are three important software security principles that are briefly discussed: confidentiality, integrity, and availability. Confidentiality consists of making sure that only the authorized users has access to the data while integrity consists of making sure that only the authorized users can modify the data. Availability refers to the accessibility of data or the system to authorized users at all time. The security policy details which information containers (files, variables, registers, ...) are being considered as well as the security properties that are studied. Security mechanisms are used to detect, prevent or recover from a security attack. There are two important types of security mechanisms: preventive and reactive mechanisms. The preventive mechanisms are used in order to reduce the probability of an attack (e.g. cryptographic mechanisms, isolation, and formal proof). However, the attackers end up finding security loopholes in the existing code and bypass these mechanisms. Therefore, reactive mechanisms have been proposed in order to monitor the system and detect a security policy violation in order to recover. One of the most generic and universal solution to detect software attacks is the dynamic information flow tracking.

DIFT approaches target confidentiality and integrity principles. Existing software-based DIFT approaches are flexible (multiple security policies can be applied in order to detect multiple types of software attacks) while requiring an important amount of computation time. Therefore hardware-assisted approaches have been proposed in order to speed up the execution time of DIFT operations. However, almost all of these hardware-assisted DIFT approaches target open architectures such as SPARC V8 [9] whereas ARM, Intel, and AMD architectures are widely used in the industry are left unstudied. Furthermore, some related works target a system with a standalone OS (*Operating System*) which is not the case in most systems used in the industry. All these reasons showed the need to bridge this gap by providing a solution that can protect from most types of software attacks and is portable to existing systems.

There are two main types of approaches in order to detect software attacks: signature-based and behavioral-based. The signature-based approach consists in comparing the action or code sequence of the application with a known set of "signatures" (i.e. known pieces of code that indicate an attack or a malware). The behavioral-based approach consists in monitoring program behavior and comparing it with a model of the legitimate behavior: it works well in

order to detect new unknown attacks while a signature-based approach only detects known attacks.

The goal of the work proposed in this thesis is to propose a solution for software security on an ARM-based CPU running Linux OS in order to bridge the gap between research and industry. The ARM architecture is one of the most used architectures in embedded devices (smartphones, tablets and other electronic devices). In 2013, ARM-based chips were found in 60 % of the world's mobile devices [10]. The Linux kernel is also one of the most used OS in the world. The choice of the architecture and the OS is crucial in order to make sure that the end user would be able to take advantage of the approach proposed in this work.

This work studies behavioral-based solutions in order to detect software attacks. More precisely, we rely on taint-based and anomaly-based approaches. Taint-based detection scheme consists in adding tags to the information container, propagating the tag each time the information container is accessed or modified and checking the value of the tag when it is copied or written to another information container. The information container differs depending on which level the detection is made. For instance, if the low-level scheme is used, then the information container is a register or a memory address. The anomaly-based detection scheme consists in detecting programs that are likely to present a threat and block them. Chapters 2, 3, 4 and 5 studies taint-based detection in embedded systems running the Linux kernel while chapter 6 studies anomaly-based detection in embedded systems running Android.

HardBlare project

This work is done in the frame of the HardBlare project [11]. This project aims at exporting compute-intensive operations of the Blare project [12, 13, 14] to a dedicated hardware coprocessor. Blare, an OS-level taint-based approach, consists in providing security features in the Linux kernel using LSM (*Linux Security Modules*) hooks. It allows detecting security policy violations such as intrusions and malware.

Blare is an OS-level information flow monitor. The information container at the OS level is a file: it means that tags are associated with files. Each time a program accesses a file, then the tag of the file is associated with the program. Conversely, each time the program writes to a file, the tag of this file is updated with the tag of the program and the security policy verifies whether this information flow is authorized or not. The propagation and check required by the security policy are all done by the Linux kernel. Furthermore, the memory used by the process is considered to be tagged by all the files read. For instance, if the program reads two files, then the whole memory area used by the process is tagged with both tags instead of tagging only the region where bytes read from the file are located. It results in an important number of false positives and false negatives.

HardBlare uses Blare in order to properly initialize tags of the file in the kernel. When a read operation is made by the process, the kernel sends some information to the coprocessor in order to propagate tag. When the kernel writes to a file, the Blare part recovers the tag of the file being written and requests the tag of the file from the hardware accelerator. It allows having a more accurate vision of the memory as only the area that contains the file read by the process are tagged using the tag of the file (the behavior is similar for write operations).

1.2 Security policies

The security policy states how to initialize, propagate, and check tags. Defining proper security policies in order to detect software attacks with few false positives and false negatives require formal verification and some deep knowledge on the attack that needs to be detected. This problem correspond to a dedicated research field and is out of the scope of this thesis. This work aims at providing users a framework allowing them to implement any security policy.

Some works such as [15] try to analyze and formally verify that the security policy succeeds in detecting software attacks. For instance, if an uninitialized memory corruption attack is considered, then the security policy states the propagation and check rules shown in equations 1.1 and 1.2. For the purpose of simplicity, only two tag values are considered: 0 and 1 where the value 0 means that the tag is uninitialized, the value 1 means that the memory is initialized and t_m represents the tag of the memory address being accessed or modified by the instruction.

$$\textbf{Store} : \{t_m = 1\} \tag{1.1}$$

$$\textbf{Load} : (t_m = 0) \rightarrow \textit{interrupt} \tag{1.2}$$

The bold text shows that the operation must happen if the instruction being executed on the main CPU has the mentioned opcode, then the propagation or the check operation should take place. If the instruction is a `store` instruction (equation 1.1: it could be either `str`, `stm`, `vstr` or `vstm` instruction), then the tag of the memory address should be set to 1. Otherwise, if the instruction is a `load` instruction (equation 1.1: it could be either `ldr`, `ldm`, `vldr` or `vldm` instruction), then the tag of the memory address should be checked. If it is equal to 0, then an uninitialized memory address is being accessed allowing to detect the memory corruption. For all other instructions, there is no propagation or check operation. This simple example shows how the security policy is specified and how it can be used in order to protect against uninitialized memory corruption attack. Similarly, there is a security policy that

specifies how to propagate and check tags in order to implement information flow control. DIFT (*Dynamic Information Flow Tracking*) requires that for each instruction executed by the CPU, another operation must be done on the tag of operands used in the instruction. Depending on the instruction type and the software attack protected, propagation and check rules of the security policy may differ.

1.3 Non-exhaustive list of software attacks

This section presents the buffer overflow attack that can be protected using DIFT. The attack itself and the security policy used in order to detect this attack are explained. There are other types of attacks (Format string, SQL injection, Cross-Site Scripting, Directory traversal...) that can be detected using DIFT and specific security policies as described in [15].

1.3.1 Buffer Overflow

Buffer overflow is one of the first software attacks that resulted in the emerge of the software security field. It consists of overwriting some buffer in memory to modify program control flow and execute some malicious code. The goal of the security policy is to enforce the integrity of return address in order to prevent from modification of program control flow. Listing 1.1 shows an example code that can result in a buffer overflow. The user types an input that is stored in `idx` variable. It is then used to modify an element of the buffer. If the user types an index value that is greater than the buffer size, then this could result in modifying the elements outside the buffer which could corrupt the program data. Historically, the attacker overflows the buffer in such a way that the input data contains malicious code which could be executed after overflow by modifying the return address stored in memory (as shown in Figure 1.1). When the return address is used, it points towards the specially crafted malicious injected code by the attacker.

```
1 int idx = tainted_input; //stdin (> BUFFER SIZE)
2 buffer[idx] = x; // buffer overflow
```

Listing 1.1 – Buffer overflow example code.

There is an important amount of techniques that allow preventing from buffer overflows: stack canaries, DEP (*Data Execution Prevention*) and ASLR (*Address Space Layout Randomization*) are the most common ones. Compilers can insert stack canaries after each buffer in order to detect a change of this value after execution allowing to successfully detect the attack. DEP consist in adding a mark using a special NX (*Non eXecutable*) bit available in the MMU (*Memory Management Unit*). It allows preventing the attacker to execute injected code from the stack. So, even if the attacker succeeds in modifying return value and pointing

Registers	
Tag	**Data**
	r1:&input
	r2:idx=input
	r3:&buffer
	r4:&buffer+idx
	r5:x

Memory	
Tag	**Data**
	Return Address
	int buffer[Size]

Figure 1.1 – Initial state of the system before the program starts execution.

it towards the injected code, the attacker will not be able to run that code residing in the stack. To bypass the DEP, attackers come up with another attack named ROP (*Return-Oriented Programming*). Instead of overwriting the return address to point towards the injected code on the stack, the attacker manipulates the return address to point towards the instructions already available in the program also known as "gadgets" (i.e. existing code in the `.text` section). ASLR was designed in order to protect against ROP attacks. It works by randomizing the value of `.text` and `data` segments on each run. Therefore, the attacker cannot precisely determine the address of "gadgets" and cannot take advantage of the ROP attack.

1.3.2 DIFT security policy

DIFT consists in adding a tag to a register or a memory address. Each time an instruction is executed in the CPU, it makes an operation from one or multiple source operands (register or memory address) towards a destination operand (a register or memory address). DIFT consists in propagating the tag alongside the instruction in order to monitor program behavior. The goal of this paragraph is to show how the DIFT can be used to prevent from buffer overflows. Listing 1.2 shows the simplified assembly code of the program shown in Listing 1.1.

```
1 set r1 ← &tainted_input
2 load r2 ← M[r1]
3 add r4 ← r2 + r3
4 store M[r4] ← r5
```

Listing 1.2 – Simplified assembly code for the buffer overflow example code (Listing 1.1).

Figure 1.1 shows the initial state of the system before program execution. The main CPU registers are extended to contain an additional information about the tag. Green tags represent private tags while red tags represent public tags. Private tags mean that the information is coming from a secure source whereas public tags mean that the information is originating from a non-secure (or public) interface such as the standard input `stdin`. Each time an instruction is executed, the corresponding operation is realized on the tag. For

Registers	
Tag	**Data**
	`r1:&input`
	`r2:idx=input`
	`r3:&buffer`
	`r4:&buffer+idx`
	`r5:x`

Memory	
Tag	**Data**
	Return Address
	int buffer[Size]

Figure 1.2 – Final state of the system after the execution of program code.

instance, if the first instruction in Listing 1.2 (consisting in moving address of `tainted_input` in register `r1`) is executed on the CPU, the tag of `tainted_input` is also copied into register `r1`. When the second instruction is executed, the tag of `r1` is also copied into `r2`. `r4` is tainted after executing the third instruction because `r2` is tainted. When the store operation on the line four takes place, the tag of the `r4` register is copied to the `r5` register. Figure 1.2 shows the final state of the system after the program has finished execution. When the return address is used, the tag is checked which will result in the generation of an exception due to the fact that the return address is tagged as non-secure.

1.4 Open Research questions

This section presents some important research questions that are not dealt in existing works.

1.4.1 Question 1: How to recover required information for DIFT on hardcore CPUs ?

Most of existing works target softcore CPUs for DIFT implementations. It results in important design considerations that cannot be fulfilled on a hardcore CPU. One example is the interface between the CPU and the DIFT accelerator often implemented in FPGA. Decoupling tag operations on FPGA requires some information from the CPU. Existing works consider that this information can be taken from existing CPU signals. If the target CPU is a softcore, then all internal signals can be easily extracted and used in order to recover information required for DIFT. However, this is not the case on a hardcore CPU as internal signals cannot be extracted. Therefore, it is important to propose an approach allowing to recover information required for DIFT on a hardcore CPU.

1.4.2 Question 2: DIFT on ARM architecture (hardcore CPU)

Most devices (smartphones, tablets and electronic systems) include ARM-based CPUs. Therefore, it is important that the proposed approach provides a solution designed for the ARM architecture. It provides general purpose registers and an FPU (*Floating Point Unit*) in order to speed up computation on floating point numbers. The FPU requires special instructions that are not considered in existing works. Furthermore, software considerations are different on such an architecture. For instance, the ABI (*Application Binary Interface*) is different from the SPARC architecture considered in related works. Therefore, it is important to target the ARM architecture in order to provide a solution that is easily usable by researchers, developers and experienced users.

DIFT coprocessors proposed in the literature lack some important features such as the tag initialization support from the OS. Furthermore, the tag management scheme consists of a simple bitmap which is not adapted on a hardcore. It is important to provide these missing features in order to improve flexibility and reusability.

1.4.3 Question 3: How to implement multiple security policies of different tag size ?

A program may need require a protection based on multiple security policies. Existing works provide limited support for this feature. For instance, most hardware-assisted DIFT approaches can implement multiple security policies of 1-bit tag. If the tag size needs to be bigger than 4 bits, then the security policy cannot be implemented. Therefore, it is important to provide an architecture able to run multiple security policies with different tag sizes in order to detect multiple attacks.

1.4.4 Question 4: Multi-threaded system / Multicore system

Most applications take advantage of multicore systems in order to speed up execution by using multiple threads. However, no existing hardware-assisted DIFT approach provides a way to manage multiple threads. Therefore, it is important to propose an approach that can be used with a multi-threaded program.

1.5 Manuscript organization and contributions

1.5.1 Manuscript organization

This section presents the overall organization of this thesis. Chapter 2 describes important existing works on the topic of information flow tracking and provides a comparison of features proposed in this work. Then, chapter 3 explains the first contribution of this work that details how to recover information required for DIFT on an ARM-architecture based hardcore CPU. Chapter 4 presents a first proof-of-concept implemented using a MicroBlaze softcore CPU in order to experiment that the proposed approach can allow implementing DIFT on a hardcore CPU. Later, chapter 5 shows a detailed view of the custom DIFT coprocessor that is implemented in order to improve the execution time of tag operations. Chapter 6 describes a different approach for malware detection on Android smartphones. The main goal is to propose an energy-efficient implementation of a malware detection approach proposed in [16]. Finally, chapter 7 concludes this work by summarizing proposed contributions and providing future perspectives. In addition to the main content, this manuscript contains four appendices.

- Appendix **A** details some characteristics of the development board used in this work.
- Appendix **B** details the design and evaluation method followed in order to test the developed IPs in this work.
- Appendix **C** details the ISA (*Instruction-Set Architecture*) of the DIFT coprocessor proposed in chapter 6.
- Appendix **D** lists the publications that the author has contributed to during the Ph.D.

1.5.2 Manuscript contributions

The contributions of this thesis are the following:

- It provides a method to recover information required for DIFT on ARM CPUs. It proposes to use CoreSight PTM, static analysis, and binary instrumentation to recover information required for DIFT. It is shown that the PTM does not add any execution time overhead. This contribution is detailed in chapter 3 and chapter 4.
- Two patches have been added in the Linux kernel in order to add support for missing features [17].
- It proposes a method to protect the DIFT coprocessor using ARM TrustZone. This contribution is detailed in chapter 4.
- It provides flexible security policies implementation in hardware: previous off-core hardware-assisted approaches lack ways of specifying security policies (compile-time or

runtime only) and do not offer support for multiple security policies of different tag granularities (page, word, etc.). This contribution is explained in chapter 5.

- This work is compatible with multi-threaded applications mainly thanks to the context ID information obtained from CoreSight PTM. It allows filtering trace using Context ID and to determine the exact order of operations on the CPU for each thread. This contribution is explained in chapter 5.
- This work tracks all information flows unlike most existing works. It allows detecting a wider range of attacks. This contribution is explained in chapter 5.
- This work provides a power-efficient and runtime-efficient implementation of three classification algorithms: LR, NB, and SVM. This contribution is explained in chapter 6.

1.6 Summary

This chapter sums up the gap between research and industry in the domain of embedded software security. One of the main reasons for this gap is that solutions in the literature do not consider architectures widely used in the industry especially ARM. This work focus on minimizing this gap by targeting the ARM architecture. There are some other important research questions that are not handled in existing works such as multiple security policies of different tag sizes or tracking multiple threads in an hardware-assisted approach. This work focuses on providing an architecture able to track multiple security policies with different tag sizes and providing a method to recover information required for DIFT for multiple threads or multiple processes by taking advantage of a unique feature of CoreSight debug components.

Chapter 2

Related works

This chapter presents existing works on information flow tracking with an emphasis on hardware-assisted DIFT architectures. It also details specifications for this work and compares them with existing works.

Contents

2.1 Introduction

This chapter introduces existing works in order to fully understand the contributions reported in this thesis. Section 2.2 describes different types of IFT (*Information Flow Tracking*): static IFT (SIFT) and dynamic IFT (DIFT) both explained in section 2.2. SIFT is mainly used to test an application binary while DIFT is used to protect against software attacks [1]. Section 2.3 explains different levels of IFT (from application to gate level): IFT can be used at different levels in order to monitor a program behavior, to detect software security attacks and even to detect hardware trojans using gate-level information flow tracking [18]. Section 2.4 describes different hardware assisted DIFT solutions. Section 2.5 explains the benefits of this work compared to existing solutions. Finally, section 2.6 summarizes this chapter.

2.2 Different types of IFT

There are two types of IFT approaches: static and dynamic IFT, each one has its own goals.

2.2.1 Static IFT (SIFT)

SIFT is an offline analysis of the application aiming to check that all branches of the control flow graph are trustworthy. SIFT is mainly used for the purposes of general program understanding, detection of attacks and vulnerabilities. Figure 2.1 shows a simplified CFG (*Control Flow Graph*) of an application compiled for the ARM-v7 architecture. The application opens a file depending on the user type determined using the `geteuid` function: if the user is root, the secret file `passwd` is opened. Otherwise, a public file `welcome` is opened. Then, contents of the file are copied into a buffer and printed on the `stdout`. The content of the `passwd` file should not be printed on the `stdout`. The CFG shows two possible paths for the application depending on the output of the test made in basic block #1. The path #1 is made up of execution of basic blocks #1, #2 and #4 while path #2 consists of the execution of basic blocks #1, #3 and #4. SIFT will analyze both these paths in order to find security vulnerabilities. As the analysis is done offline before execution, SIFT does not add runtime overhead. However, it is possible that the analyzed path may never be used by the application during execution. As a consequence, SIFT tends to produce an important number of false positives.

2.2.2 Dynamic IFT (DIFT)

DIFT is performed at runtime: it monitors data flow of the application binary in order to check if the execution is safe. DIFT is used to protect against software attacks (such as buffer overflows, format-string attacks, SQL injection, cross-site scripting, etc.) [1] or used for data leakage prevention [4]. DIFT only tracks the execution path taken by the application instead of analyzing all possible execution paths which results in an approach more precise than SIFT. Considering the example shown in Figure 2.1, DIFT would track only path #1 (basic blocks #1, #2 and #4) or path #2 (basic blocks #1, #3 and #4). However, DIFT requires more information in order to determine the execution path taken by the application.

2.3 Different levels of IFT

Figure 2.2 shows the four different levels of an embedded software architecture: application layer, system service layer, OS layer and hardware abstraction layer.

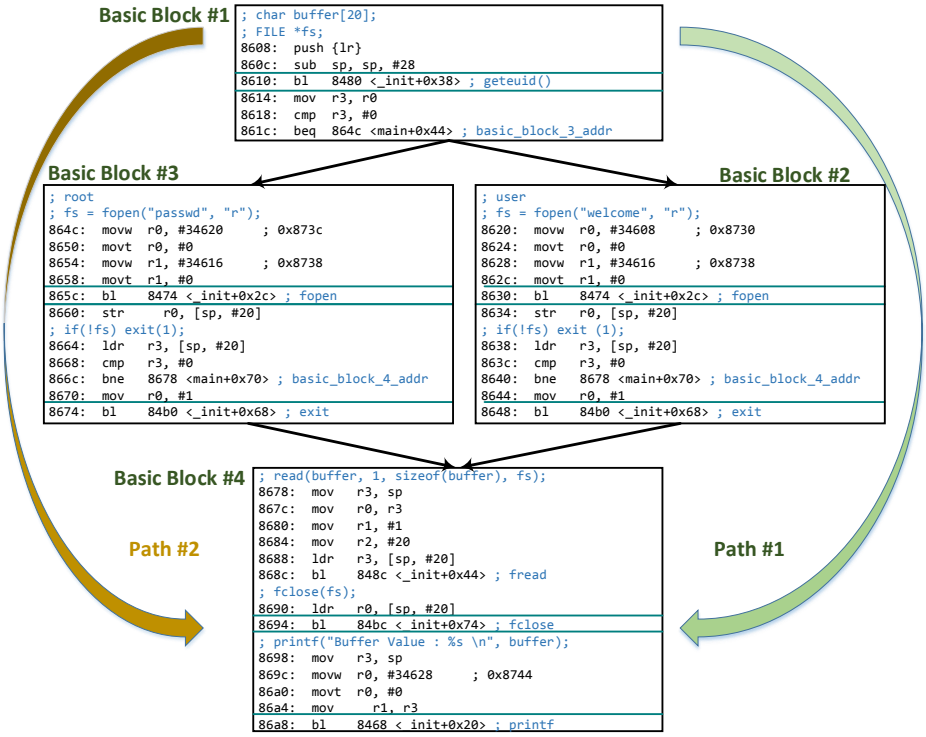


Figure 2.1 – A simplified sample Control Flow Graph.

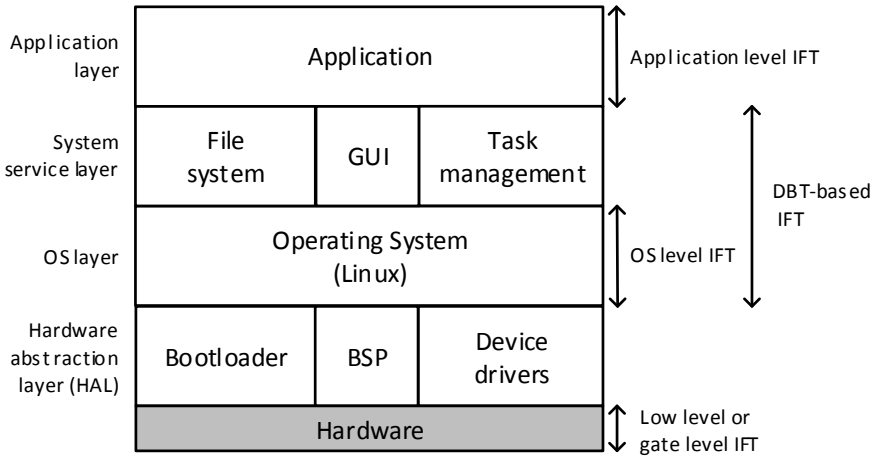


Figure 2.2 – Software architecture of an embedded system.

The IFT can be done at different levels: starting from high-level (OS-level) to hardware level (or low-level). Each level has its own advantages and drawbacks that are explained below.

2.3.1 Application level IFT

Application level IFT tracks information flows between application variables. Considering the C code shown in Listing 2.1, the corresponding information flow tracking program code is shown in Listing 2.2. The goal of information flow tracking in this example is to prevent from data leakage, which consists in leaking confidential information on a public interface, in three main steps:

1. **Tag initialization**. Each variable in the C code (for instance `a`, Listing 2.1) has its corresponding tag variable (`tag_a`, Listing 2.2).
2. **Tag propagation**. The tracking program propagates the tag of a variable each time it is used in the code (for instance variable `c`).
3. **Tag check**. Before writing the value to a channel, the tracking program makes sure that the variable being written is public. If the variable is public, then the write operation succeeds, , otherwise the tracking program raises an alert to inform the user about a data leakage.

These three steps are common to all levels of IFT. The tracking program, running alongside the original application, is generated using a modified compiler or by analyzing the program.

```
1 char buffer;
2 int a, b, c;
3 c = a + b;
4 print(c);
5
```

Listing 2.1 – Example C code.

```
1 char tag_a = 1, tag_b = 0, tag_c = 0;
2 tag_c = tag_a | tag_b;
3 if (tag_c == TAG_PRIVATE)
4   print(Secure information being sent
      to public channel);
```

Listing 2.2 – Tracking program code for application level IFT.

2.3.1.1 Interpreter-based

Some languages provide a specific mode allowing to specify whether a variable is public or private. A private variable cannot be sent towards public channels.

```
#!/usr/opt/perl
my $arg=shift;  # get parameter from command line
system($arg);   # and execute it as a system command
```

Listing 2.3 – Application level IFT.

In Listing 2.3, the Perl script has a clear security problem: a malicious user can use any system command as an argument and the script would execute it. Therefore, it is necessary to taint (in other words, add a tag to) the user argument so that when it is used, the Perl interpreter can propagate this taint to other variables. This is done by using taint mode [19] i.e. executing the script with `-T` option like `#!/usr/opt/perl -T`.

2.3.1.2 Language-based

There are several security extensions that have been proposed for existing languages. This subsection cites some important approaches and provides a summary of these works.

Java One of the first works that proposed a programming language extension is JFlow [20]. JFlow adds statically-checked information flow annotations to Java language. Because this approach is static, its main advantages are little code space, data space and low runtime overhead in the implementation. JIF [21] is another extension that adds support for information flow control and access control that are enforced at both compile-time and run-time. JBlare [22] is an hybrid approach associating a tag to each variable used by the program in order to track all information flows. The Java bytecode is analyzed statically in order to generate annotations that are executed dynamically by a modified JVM interpreter.

Android TaintDroid [23] is an extension of the Android platform that tracks data information flows in applications. The main goal of their proposed approach is to detect when sensitive data leaves the system in order to analyze specific Android applications or services. AndroBlare [24] is an approach that tracks information flows on the Android operating system. It modifies an existing information flow monitor, called Blare, that tracks information flows in Linux in order to trace information flows on an Android platform.

Other languages There exist other solutions specific to each language. For instance, a solution is proposed in [25] for Javascript and FlowCaml [26] for OCaml.

2.3.2 OS level IFT

The application level IFT tracks information flows for each variable of the application which results in an important runtime overhead. Therefore, another coarse-grained approach was proposed to track information flows at the OS level. In this approach, instead of tagging variables used by the application, the files used by the application (read or written) are tagged. On the one hand, the advantage of this approach is that it reduces the number of information flows which results into an improvement of the runtime overhead. On the other hand, the main drawback of this approach is that it results in more false positives than the application level approach. There are two types of OS-level IFT approaches: approaches based on a dedicated OS and approaches that modify an existing OS.

HiStar [27] is a new OS that has been designed in order to provide precise data specific security policies. Asbestos [28] and Flume [29] are some other solutions that tracks information flows on specific OSs. On the other hand, Blare [30] is an information flow monitor that tracks the information flow of Linux kernel OS. Figure 2.3 describes how Blare works. Each time an application reads a file, the OS tags the corresponding process with the tag of the file read. Similarly, when the application writes to a file, the process associates the tag of the process to the file being written. Security policies are enforced by checking the tag of the process that writes to a file. The work proposed in this thesis uses Blare [30] and a modified Linux kernel OS in order to provide support for retrieving and storing tags of files.

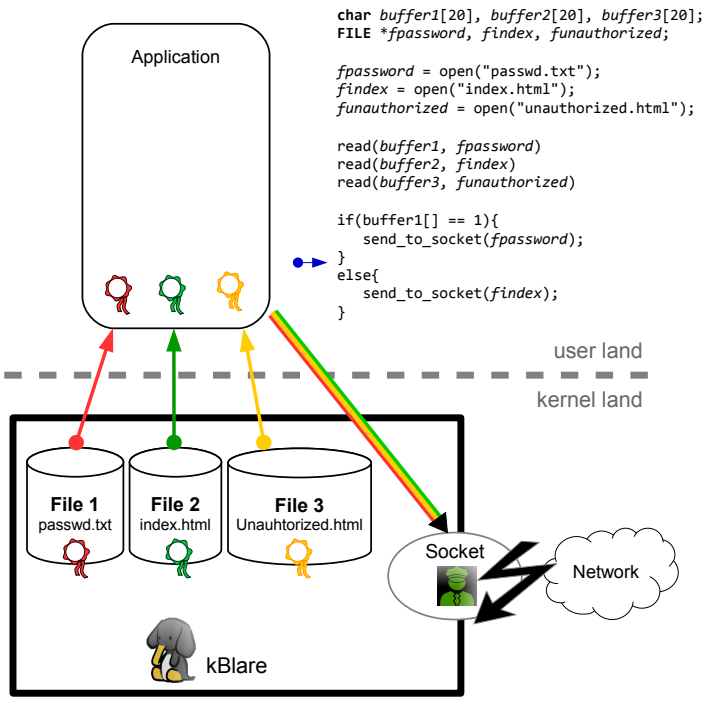


Figure 2.3 – Blare approach.

2.3.3 Low level IFT

Low level IFT tracks information flows at a lower level than application level. Instead of tagging variables or files, this approach tags registers and memory addresses. The idea of tagging registers and memory addresses is to design an hardware accelerator in order to improve execution time overhead of IFT. This approach can be implemented using different types of solutions. The different low-level hardware-assisted solutions are described in section 2.4 as it will be the approach followed in this thesis work.

2.3.4 DBT-based IFT

There are many existing dynamic binary instrumentation frameworks: Pin for Intel [31], StarDBT for multiple platforms [32], DynamoRIO for multiple platforms especially for ARM [33] and Padrone [34]. This non-exhaustive list shows few important works on DBT frameworks. These DBT frameworks can be used in order to implement DIFT. In practical, they are not used because they add huge runtime overhead. However, they are used to implement memory checking tools and some specific fixed security policy but lack generic security policy solutions.

QEMU-based

The instrumentation-based approach is architecture dependent. Whelan et al. [35] describes PIRATE, an architecture-independent DIFT approach. It is based on the QEMU (*Quick EMUlator*) [36] dynamic binary translator. It translates code of the guest architecture into its own custom IR (*intermediate representation*) and then converts it into different host architectures. The idea is to use the QEMU IR to analyze information flows.

2.3.5 Gate level IFT

Gate level IFT [37] is used to create "secure" circuits. The goal is to protect against hardware trojans and unauthorized behaviors. This is done, during the creation of the hardware circuit, by adding additional logic for each gate used in the design.

2.4 Hardware-assisted

In [1], Dalton et al. reports that software DIFT solutions add huge runtime overheads: the application can be slowed up to 37 times if DIFT is performed in software. Therefore, an interest arose in developing hardware accelerators in order to improve execution time

overhead at the expense of flexibility. This section details different types of hardware-assisted approaches: new CPU architecture designed for DIFT, in-core, off-loading and off-core DIFT approaches.

2.4.1 New CPU architecture

Instead of securing existing platforms, the SAFE project [38] proposes a new CPU architecture that takes into account security features. The architecture includes additional features designed for security. The originality of this work is to have a different look at security. Security is not an option and is included by default in the system. The main drawback is that the whole software stack needs to be adapted to the architecture. Another similar approach is proposed in CHERI [39].

2.4.2 In-core

The in-core approach relies on a deeply revised processor pipeline. Figure 2.4 shows the architecture proposed in [1]. All memory elements (DRAM, instruction and data caches) are extended with tag bits. Furthermore, the processor architecture is modified in order to compute tags in another level. When the processor fetches an instruction, the corresponding tag is fetched as well. During decode stage of the processor pipeline, the first level decodes the instruction while the second level decodes the security policy in order to determine how tags should be propagated and checked. When the first level executes the instruction, the second level propagates the tag. When the data is written into memory, the tag is taken from tag ALU and written to memory at the same time. To sum up, each stage of the pipeline is duplicated with a specific hardware module in order to realize tag-related operations all along the program execution. This approach can be adapted to add support for multi-threaded software but invasively modifies the processor architecture making it not portable on hardcore CPUs.

In [40], similar to Raksha [1], PUMP modifies the CPU architecture in order to make DIFT computations in the processor pipeline. In addition, they proposed a rule cache that avoids the computation of a new tag if the rule cache is hit. This in-core approach is not feasible with a hardcore CPU, not modifiable, such as the ARM Cortex-A9 considered in this work.

2.4.3 Off-loading

The off-loading approach shown in Figure 2.5 [41, 42] suggests to separate program execution from DIFT computation by using two general purpose processors. The first core executes the application while the second core executes the tag-related operations.

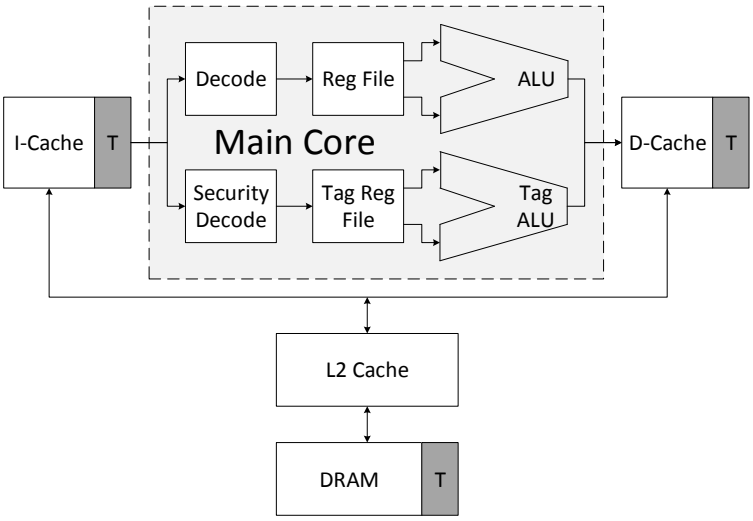


Figure 2.4 – In-core DIFT (adapted from [2]).

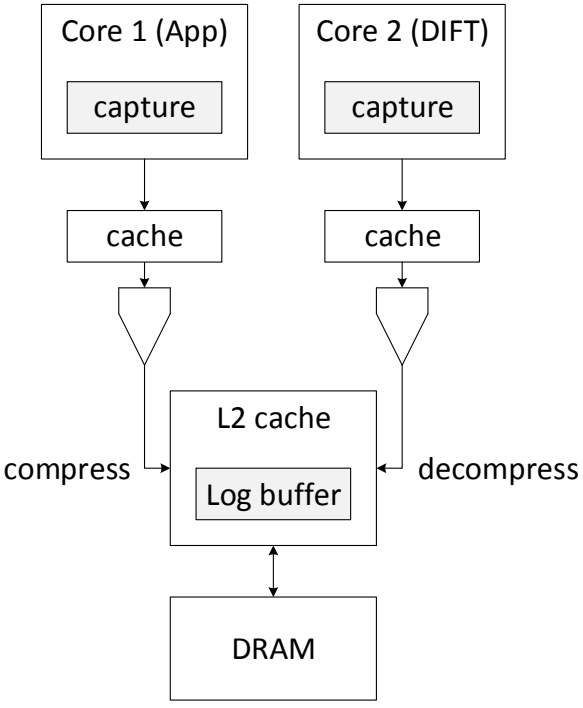


Figure 2.5 – Off-loading DIFT: Log-based approach (adapted from [2]).

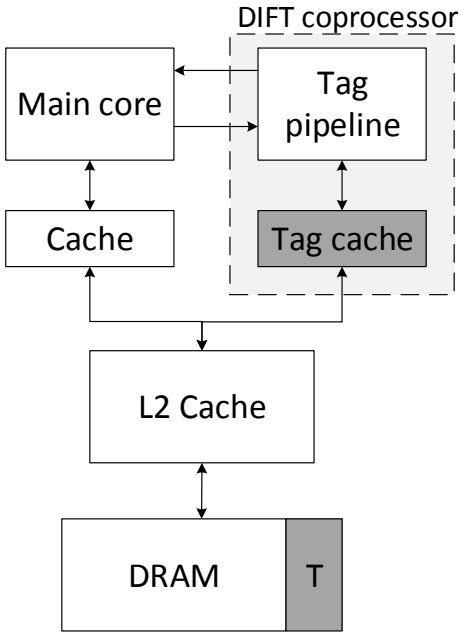


Figure 2.6 – Off-core DIFT (adapted from [2]).

In addition to program execution, the first core also compresses and stores the information required for DIFT inside a shared buffer located in memory or cache. The second core decompresses this information and realizes tag computation in order to check whether an illegal information flow has taken place or not. Illegal information flows are specified using security policies.

2.4.4 Off-core

This approach seems similar to the off-loading one. However, DIFT is performed on a dedicated coprocessor instead of a general purpose processor. Furthermore, this approach requires support from the OS in order to synchronize the main core and the DIFT coprocessor. The synchronization part stalls the main core in order to wait for the end of tag propagation. Another important thing to notice is that the OS is responsible for initializing tags of memory regions. However, no existing approach proposed an architecture that offers this possibility.

2.4.4.1 RAKSHA v2

Kannan et al. [2] described one of the first works that proposed the idea of using a dedicated DIFT coprocessor in order to improve tag computation runtime overhead. The tag pipeline unit (Figure 2.6) is responsible for propagating tags for registers and memory addresses. The tag cache is used to fetch tags from memory. The memory is extended with some additional tag bits. The most important architectural decision is the interface between the main core

and the tag pipeline unit. Figure 2.6 shows this interface. As the targeted main core is a softcore CPU, the existing CPU signals are exported towards the DIFT coprocessor. However, sending this information on a hardcore CPU typically requires instrumentation and static analysis [4] and adds over 90% of total DIFT time overhead. Therefore, it is important to reduce the amount of information that needs to be sent in order to compute tag operations on the FPGA side. In addition, this approach considers a tag size of only 1 bit per memory word making it not suitable for more complicated security policies such as heap protection security policy presented in [15] which requires a tag size of multiple bits (up to 32). The security policy can only be specified at run-time using dedicated configuration registers: TPR (*Tag Propagation Register*) and TCR (*Tag Check Register*).

2.4.4.2 FlexCore

Deng et al. [3, 43] proposed an hardware accelerator that can work up to 1GHz. The idea of increasing frequency is to propose an architecture that can match the main CPU frequency running the monitored application. However, the hardware accelerator requires a lot of information from the main core. Therefore, this solution is not suitable for hardcore CPUs as the amount of information that needs to be sent would result in a huge runtime overhead. Another original aspect of their work is to propose a generic hardware accelerator that can be configured to run different security policies [43]. Furthermore, this approach is lacking support for modern OS (such as Linux kernel) making it not suitable for hardcore CPUs.

2.4.4.3 Program Analysis Unit (PAU)

Heo et al. [4] proposed a monitor named PAU that can run different security policies. The original aspect of their work is to propose an instrumentation-based solution that is adaptable on hardcore CPUs. This approach instruments code (i.e. adds instructions) in order to track information flows for each CPU instruction executed by the application. However, adding an instruction for each CPU instruction results in a huge runtime overhead. Therefore, Heo et al. [4] proposed an approach that instruments only memory addresses and branch instructions. Listing 2.4 shows an example ARM assembly code and added assembly instructions as suggested by [4]. The instructions at line 1, 2 are required in order to reconstruct the CFG. While lines 4 and 8 have been added in order to recover memory addresses. This approach requires some static analysis to get information flows inside basic blocks of the application.

```
1  mov 0x5, r8        // Basic Block ID: 5
2  str r8, [r9]       // Trace #0: Basic Block ID
3  ldr r3, [fp, #-20]
4  str fp, [r9]       // Trace #1: Load address is pushed.
5  and r3, r3, #1
6  cmp r3, #0
7  str r2, [r1]
8  str r1, [r9]       // Trace #2: Store address is pushed.
```

Listing 2.4 – Example ARM assembly code (instrumented).

This solution was implemented on a softcore CPU (Leon 3, SPARC-v8 architecture) inside an FPGA instead of a hardcore CPU. In addition, the code is analyzed statically in order to generate the tag propagation rules that runs on the FPGA part. The security policy is fixed during compilation and the program needs to be re-compiled in order to run a different security policy.

Lee et al. [5] proposed the idea of using the CDI *Core Debug interface* in order to recover information required for DIFT. The debug component taken into account sends information for each CPU instruction being executed. As for [4], [5] targets a softcore CPU. On a hardcore CPU, it is possible to recover the same information with ARM CoreSight ETM (*Event Trace Macrocell*) trace component. However, this trace component has been replaced with the more recent CoreSight PTM (*Program Trace Macrocell*) as in the Zynq SoC, embedding ARM Cortex-A9 CPU, considered in this work. The PTM sends information only for instructions that change program flow such as branches and interrupts. Therefore the approach proposed in [5] is not compatible with Zynq SoCs.

2.4.4.4 Others

There exist many other important works regarding hardware architectures for DIFT. The other approaches, non-exhaustive list, are : Minos [44], RIFLE [45], LIFT [46], WHISK [47], PIFT [48], V-DIFT [49]. Recent works, such as [50], [51], takes care of including support for implicit control flows in existing DIFT mechanisms. Micropolicies [15] and Stack protection [52] are other important works that consist in defining a valid security policy for DIFT. These works are important in order to understand different types of attack that can be detected using existing DIFT architectures.

2.5 Proposed solution for DIFT on hardcore CPUs

2.5.1 Specifications

No existing work has considered and deployed their work on the ARM architecture which results in the fact that existing solutions are not used by hardware vendors and research community. It is very important to protect ARM cores because most of the portable devices (smartphones, tablets and laptops) use ARM CPUs. It has been reported in [53] that 17.7 billion ARM-based chips, representing a market share of 34%, have been shipped in 2016. This number is expected to grow up to 55 billion devices only for market share of application processors (smartphones, tablets and laptops). Therefore, this work targets the ARM architecture in order to protect it against software attacks in embedded systems.

Another important aspect that must be taken into account is regarding security policy. The architecture should have the maximum flexibility regarding security policies. The security policy specifies how to propagate and check tags in order to detect a particular kind of attack [40]. There are two missing aspects in existing off-core solutions regarding security policies: flexibility to specify security policies and tag granularities.

Security policies can be specified at compile-time or at runtime. The compile-time solution [4] consists of hard-coding the propagation and checking operations using dedicated opcodes, during the compilation of the application. For instance, consider that the security policy states that all arithmetic and logic instructions on ARM core result in the logical `OR` operation on their corresponding tags. Then, during static analysis, for all arithmetic and logical instructions of the program code, an `OR` operation is used to compute the tags of their operands. The runtime solution [1] requires special register called TPR (*Tag Propagation Register*) to specify the operation that has to be done on tag values. This time, the static analysis gives the operands for each instruction and the class of ARM instruction: arithmetic/logical, load/store, branch, floating point load/store. Thanks to this class information, the DIFT coprocessor can determine with the help of the TPR register, the operation that needs to be done to propagate the tags corresponding to the instruction executed on the main CPU. The main advantage is that the TPR value can be modified at runtime to modify the policy without recompiling the application. No existing works provide the flexibility to specify security policies using both methods. The proposed architecture should support both methods of specifying security policy.

2.5.2 Comparison

Tables 2.1 and 2.2 show different aspects of related works and compare them with this work. Table 2.1 shows that no existing work has proposed an implementation on a hardcore CPU

as the experimental target in existing works is a softcore CPU. Furthermore, no existing approach has targeted an ARM architecture. In terms of communication interface used, no approach has used the recent debug components sending information on control flow alone. In addition, most existing works except [1, 2], cannot protect against software attacks from non-tracked programs. The only hardcore-portable solutions are [4, 5] but their implementation has been done using a softcore CPUs.

Table 2.1 – Features comparison with related works.

Related work	Type of approach	CPU type	Experimental Target	Hardcore portability	Communication interface	Interface simulated	Coprocessor isolated
[1] [40]	In-core	Softcore	Leon3	No	Signals	N/A	No
[42] [41]	Off-loading	Softcore	Leon3	No	Log buffer in cache	No	No
[2] [3],[43]	Off-core	Softcore	Leon3	No	Signals	No	No
[4] [54]	Off-core	Softcore	Leon3	Yes	System bus	No	No
[5]	Off-core	Softcore	Leon3	Yes	CDI	Yes	No
This work	Off-core	Hardcore	ARM Cortex-A9	Yes	EMIO and System bus	No	Yes

Table 2.2 shows that off-core approaches do not provide the same features as existing in-core and off-loading approaches. It can be seen that most off-core approaches lack support for multi-threaded and floating-point software. Furthermore, most existing mechanisms are limited in terms of support for different security policies either because of tag granularity or tag scheme. In addition, most existing solutions lack support for modern OS such as Linux kernel.

Table 2.2 – Features comparison with related works.

Related work	Type of approach	Experimental Target	Hardcore portability	Communication interface	Interface simulated	Coprocessor isolated
[1] [40]	Yes	Physical	Partial	4	Extended memory	No
[42, 41]	Yes	Virtual	Partial	4-8	Tag TLB	No
[2]	No	Physical	Partial	4	Extended memory	No
[3], [43]	No	Physical	No	1-32	Tag TLB	No
[4]	No	Physical	No	1	Packed array (bitmap)	No
[5]	No	Physical	Partial	1	Packed array(bitmap)	No
This work	Yes	Virtual	Yes	1-32	TMMU	Yes

2.6 Summary

This section presented an overview of different DIFT approaches with an emphasis on hardware-assisted approaches. The existing hardware DIFT mechanisms have not targeted the ARM architecture and hardcore CPUs. Some important missing features in existing works are: specifying security policies using compile-time or run-time solution, support for floating-point, multi-threaded software and kernel support for initializing and checking tags. These missing features are the main reasons why most existing DIFT mechanisms have not been adapted by hardware vendors or by research community. This thesis is based on existing off-core DIFT approach but differs in its implementation: the application runs on a hardcore CPU rather than a softcore CPU as in previous works. The goal is to implement DIFT on hardcore CPUs while minimizing runtime overhead so that security does not slow an application, at the expense of a relatively small hardware increase. The above-mentioned missing features in related works have been addressed in this work. One of the main challenges is to find a way to recover required information for DIFT on a hardcore CPU. In existing works, the information is easily recovered via existing softcore CPU signals. However, no such signals exist on a hardcore CPU. Another important achievement is to make sure that all information required for DIFT can be recovered on the considered architecture.

Chapter 3

Recovering information required for DIFT on hardcore CPUs

This chapter explains how the information required for DIFT on ARM hardcore CPUs can be recovered using ARM CoreSight components, static analysis, instrumentation and RFBlare [6], a Linux kernel information flow monitor.

Contents

3.1 Introduction

This work aims at providing a method to recover information required for DIFT on a hardcore CPU. One of the main challenges faced in order to track information flows on a hardcore CPU is the visibility issue. The FPGA part does not know which instructions are being executed by the ARM CPU core. Therefore, the information required to compute DIFT operations needs to be determined and sent to the FPGA part. A simple approach would be to send an information for each CPU instruction executed by the CPU on the FPGA part. However, sending an information for each CPU instruction executed would increase the total runtime overhead. Existing works mainly send signals to the DIFT coprocessor in order to compute DIFT operations. As all existing works have their main core in the FPGA itself, sending existing softcore CPU signals do not add any runtime overhead. However,

these signals cannot be extracted from a hardcore CPU. Therefore, information recovery for DIFT adds important runtime overhead on a hardcore CPU. Heo et al. [4] show that the communication time, i.e. the time required to send information to the FPGA part, represents 90% of the total runtime overhead required for DIFT operations. As a consequence, it is important to lower the amount of information to be sent to the FPGA coprocessor in order to minimize this communication time. First, the information required for DIFT on a hardcore CPU is detailed in section 3.2. In section 3.3, CoreSight components are described as they play an important role in the proposed approach in section 3.4. Section 3.5 evaluates the proposed approach by providing time, area, and power overheads. Section 3.6 sums up this chapter.

3.2 Context of the proposed approach

All hardware-assisted existing works protect applications running on a softcore CPU. As the program runs on a softcore CPU, the information required for DIFT is recovered using existing softcore CPU signals. Table 3.1 shows different signals sent from the main core (softcore CPU) executing the application towards the DIFT monitor for existing off-core solutions. On a hardcore CPU, this information cannot be recovered using signals.

Table 3.1 – Information required for DIFT in existing works.

Related work	Information sent to DIFT monitor on the FPGA
Kannan et al. [2]	PC, instruction, memory address, valid
Deng et al. [3, 43]	PC, instruction, memory address, result of an instruction, source operands, branch, opcode, decoded instruction configuration signals
Heo et al. [4]	Instructions, execution trace (instrumentation), static analysis
Lee et al. [5]	Core Debug Interface (CDI)

Some signals, shown in Table 3.1, are not mandatory to compute tag operations and implement DIFT: for instance, the result of an instruction, sent by [3, 43], is not required in order to compute tags. This section describes which signals are needed for tag operations and, most importantly, how these signals can be recovered on the FPGA part without adding an important runtime overhead.

Information required on the FPGA part

Listing 3.1 is a C code used as an example, throughout this chapter, to illustrate which pieces of information are required for DIFT. In this program, two files (one public and one secret) are read and one file is written. The content of the file being written can come from either

the public or the secret file. The goal is to explain the required information for DIFT using this simple example code.

```
int main() {
  int file_public, file_secret, file_output;
  char public_buffer[1024];
  char secret_buffer[1024];
  char *temporary_buffer;
  file_public = open("files/public.txt",O_RDONLY);
  file_secret = open("files/secret.txt",O_RDONLY);
  file_output = open("files/output.txt",O_WRONLY);
  read(file_public, public_buffer, 1024);
  read(file_secret, secret_buffer, 1024);
  srand(time(NULL));
  if( (rand() % 2) == 0){
       temporary_buffer = public_buffer;
  }
  else{
       temporary_buffer = secret_buffer;
  }
  write(file_output, temporary_buffer, 1024);
  return 0;
}
```

Listing 3.1 – Example C code for DIFT

This C code is compiled using the LLVM (*Low-Level Virtual Machine*) compiler in order to obtain an ELF (*Executable and Linkable Format*) binary running on the target platform (based on the ARM-v7 architecture). A simplified control flow graph composed of five basic blocks is shown in Figure 3.1. For this code, trace (i.e. addresses generated by CoreSight components) of the main function is shown in Listing 3.2.

```
10618 10494 10634 10494 10648 10494 1065c 10464 10678 10464 10690
1047c 10698 10470 106a4 10458 106a8 106c8 10440 106e4 00000 00000
```

Listing 3.2 – Decoded trace of main function of the binary compiled from Listing 3.1.

In Listing 3.2, the first decoded trace value (`10618`) corresponds to the starting address of basic block #1 (Figure 3.1). The following value is the value of the `open` function call (`0x10494`). Then, the return address of the function call (`0x10630+4` or `0x10634`) is obtained in the decoded trace. This example shows that the decoded trace allows to determine which basic block is being executed on the main CPU core. As the trace is generated only for instructions that change the program flow (such as branches), any other instruction is not traced. For instance, instructions between addresses `0x10618` and `0x10630` generate no trace.

So, what happens inside a basic block remains unknown for the FPGA part. Therefore, a static analysis (detailed in 3.4.2) is done offline in order to determine information flows inside each basic block of the application. However, all missing information cannot be recovered through static analysis: typically, memory addresses inside `ldr/str` instructions cannot be resolved statically.

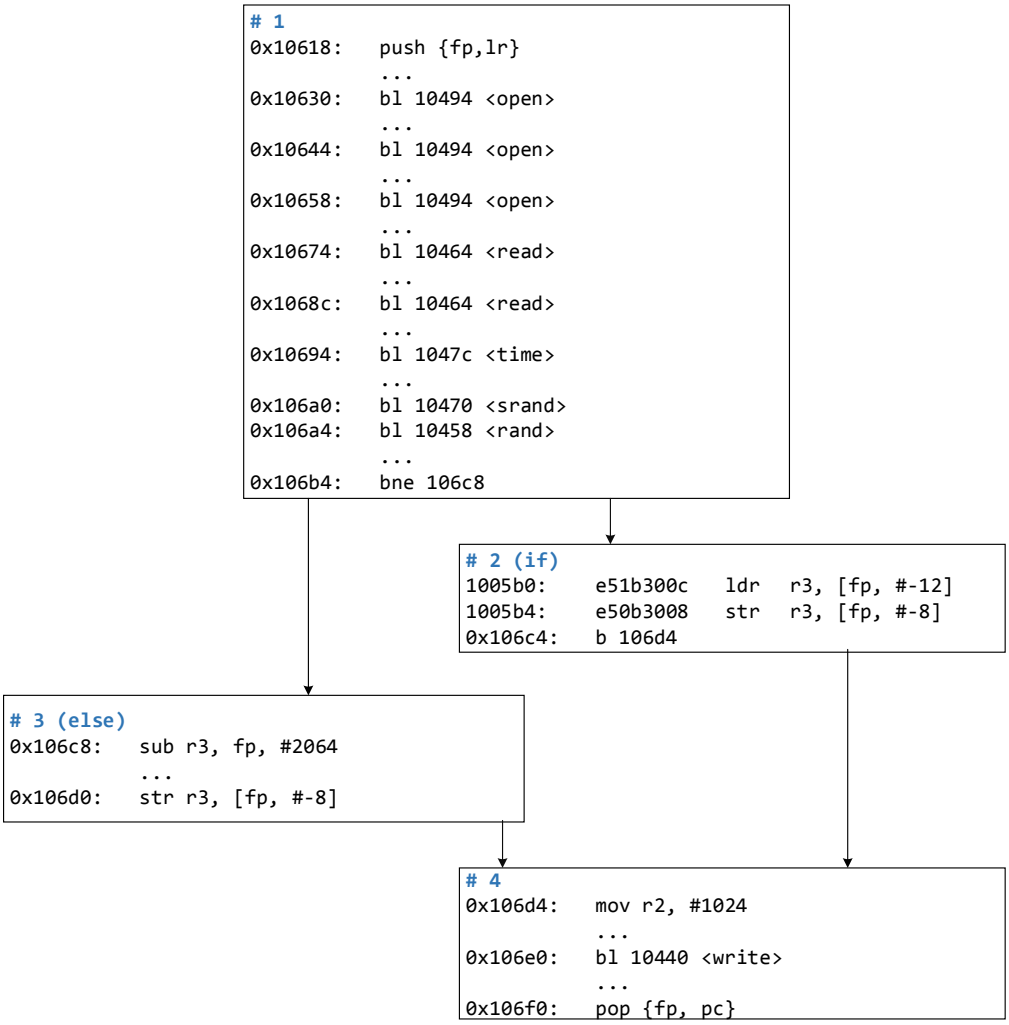


Figure 3.1 – Simplified CFG of the C code of Listing 3.1.

In order to send these memory addresses, the application binary is instrumented during compilation through an LLVM pass. Furthermore, an application relies on some external library code to take advantage of system calls and perform specific operations. Thus, the information flows of library code used by the application are also required in order to compute tags. A thorough search of the relevant literature for information flows tracking of library code yielded no article. This work proposes to use two FIFOs in order to send information

from the OS to the DIFT coprocessor and vice versa. Four pieces of information allow to overcome this visibility problem:

1. Trace obtained via CS (*CoreSight*) components.
2. Static analysis.
3. Instrumentation.
4. Dedicated FIFOs to communicate between the OS and the DIFT coprocessor.

In order to obtain trace on the FPGA, CoreSight components need to be configured. The configuration of CoreSight components is explained in section 3.3. Obtained trace needs to be decoded in order to obtain addresses of the program. It is explained in subsection 3.4.1. Static analysis is detailed in subsection 3.4.2 while the instrumentation is described in subsection 3.4.3. The dedicated FIFOs to communicate between the OS and the DIFT coprocessor are explained in subsection 3.4.4.

3.3 CoreSight components

CS components provide a set of hardware components to debug or profile the program. There are four main classes of CS components:

- **Control**. Control and access to CS components (for instance, CTI, CTM, DAP and ECT). These elements are not detailed in a subsection because they either do not require a special configuration (DAP) or are not used in this thesis (CTI, CTM and ECT).
- **Source**. Trace generation for CPU instructions (PTM , FTM and ITM).
- **Link**. Link between CS sources and CS sinks (funnel and replicator).
- **Sink**. Trace store or export (ETB and TPIU). Trace can be transmitted to an on-chip memory, the FPGA coprocessor or even dedicated pins.

The most important classes in this work are source and sink components. Figure 3.2 shows different ARM CS components available on Zynq SoC. All these components are connected to each other, each one having its important features as detailed in the following subsections.

3.3.1 Trace Source

There are three types of trace source components: ETM, ITM, and PTM. The ETM, not available on Zynq SoC, generates trace for each CPU instruction executed while the ITM is a `printf`-style like trace generator i.e. each time trace is required, the program source code should write a value to the ITM which is then recovered through trace. The PTM generates execution trace for each waypoint (i.e. instructions that modify program flow). A waypoint can be any of the following instructions: any indirect branches, conditional and

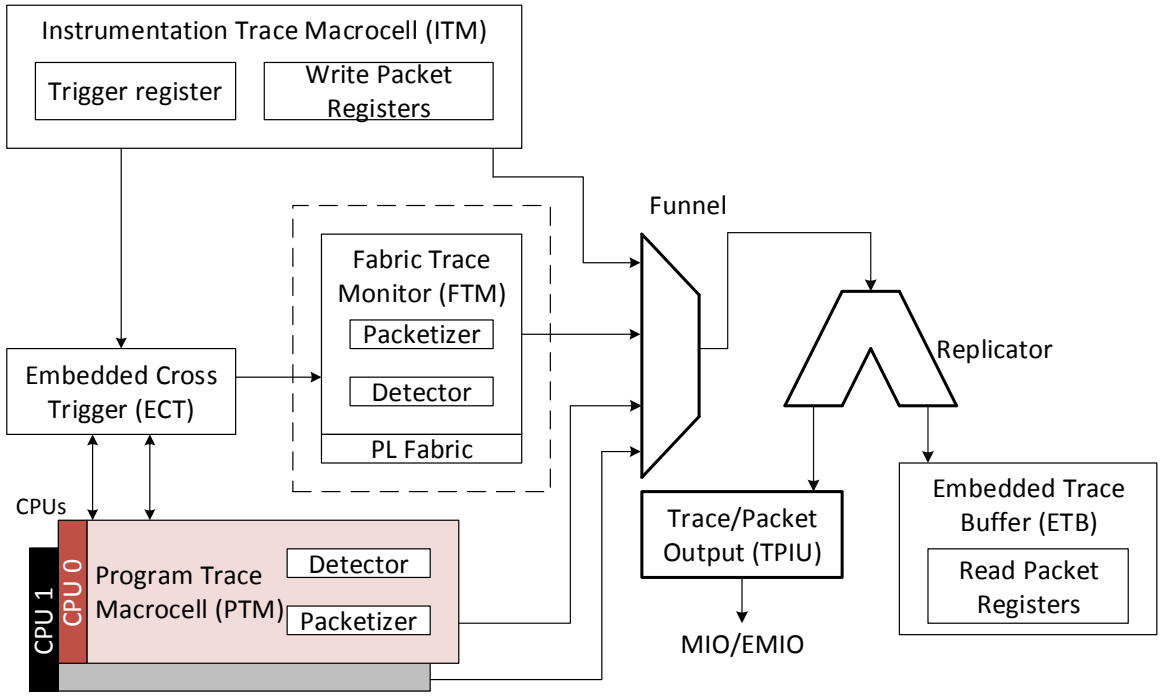


Figure 3.2 – CS components in Xilinx Zynq [55].

unconditional direct branches, all exceptions, any instruction that changes the instruction set state or security state of the processor. Each time a waypoint occurs, the PTM generates a trace describing this waypoint. Table 3.2 shows an example of trace output. Trace always starts with synchronization packets: `A-Sync` and `I-Sync` (Table 3.7). Then, each time a waypoint instruction is executed, trace is generated. For instance, `bl` and `beq` instructions generate trace packets as shown in Table 3.2. If a `BAP` packet is generated, it means that the branch was taken. Otherwise, an `Atom` packet is generated. The PTM generates 11 different types of trace packets which are described in PFT (*Program Flow Trace*) specifications [56]. All PFT packets are summed up in Table 3.7.

Table 3.2 – Example code and corresponding trace.

Address	Assembly code	Trace packets type
		`A-Sync`, `I-Sync`
860c	`sub r0, r1, r2`	-
8610	`bl 8480`	`BAP`
8614	`mov r3, r0`	-
8618	`ldr r3, [sp, #-8]`	-
861c	`cmp r3, #0`	-
8620	`beq 864c`	`BAP` or `Atom`
8624	`str r1, [r3, r2]`	-

The most important PTM feature used is branch broadcasting which allows to obtain all branch addresses in trace. However, the corresponding Linux driver did not implement this feature. A patch, shown in Listing 3.3, was developed in this work that enables the use of branch broadcasting in Linux driver. The patch has been integrated into Linux kernel v4.11 [17].

```
diff -uprN -X linux-4.7-vanilla/Documentation/dontdiff linux-4.7-vanilla/
    drivers/hwtracing/coresight/coresight-etm3x-sysfs.c linux-4.7/drivers/
    hwtracing/coresight/coresight-etm3x-sysfs.c
--- linux-4.7-vanilla/drivers/hwtracing/coresight/coresight-etm3x-sysfs.c
    2016-07-24 21:23:50.000000000 +0200
+++ linux-4.7/drivers/hwtracing/coresight/coresight-etm3x-sysfs.c 2016-09-28
    15:36:39.886542702 +0200
@@ -145,7 +145,7 @@ static ssize_t mode_store(struct device
       goto err_unlock;
     }
     config->ctrl |= ETMCR_STALL_MODE;
-  } else
+ } else
     config->ctrl &= ~ETMCR_STALL_MODE;

   if (config->mode & ETM_MODE_TIMESTAMP) {
@@ -163,6 +163,20 @@ static ssize_t mode_store(struct device
   else
     config->ctrl &= ~ETMCR_CTXID_SIZE;

+ if (config->mode & ETM_MODE_BBROAD)
+    config->ctrl |= ETMCR_BRANCH_BROADCAST;
+ else
+    config->ctrl &= ~ETMCR_BRANCH_BROADCAST;
+
+ if (drvdata->arch == (PFT_ARCH_V1_0 | PFT_ARCH_V1_1)) {
+    if (config->mode & ETM_MODE_RET_STACK) {
+      if (config->mode & ETM_MODE_BBROAD)
+        dev_warn(drvdata->dev, "behavior is unpredictable\n");
+      config->ctrl |= ETMCR_RETURN_STACK_EN;
+    } else
+      config->ctrl &= ~ETMCR_RETURN_STACK_EN;
+ }
```

Listing 3.3 – Patch developed for adding support for branch broadcasting feature of CS components in a Zynq SoC (Excerpt from patch).

Retrieving trace

In order to retrieve trace, the first thing is to make sure that these components have a corresponding driver in the Linux kernel, which is the case, and corresponding device tree entries in the device tree file. The device tree file is a file that contains information about hardware devices in a tree structure format. As there were no device tree entries for CS components available in Zynq SoCs, a patch has been proposed in this work in order to add support for these components [17]. Listing 3.4 shows the patch with different device entries corresponding to CS components available in Zynq SoCs. The patch was produced using the `diff` command on the Linux kernel 4.7 which was the current version when generating patches. Each device tree entry has several fields. The `compatible` field is the most important one because it allows the kernel to find the right driver for the entry. Other fields are used by the kernel to initialize a specific value for the driver. For instance, the `reg` field allows specifying the starting address of the hardware device and a range of programmable addresses. Using this information, the kernel can associate a virtual address for this physical address in device tree entry and this virtual address is used afterwards to program the corresponding device. After patching the device tree, CS components were accessible from the `sysfs` file system after booting the kernel.

```
--- linux-4.7/arch/arm/boot/dts/zynq-7000.dtsi.orig 2016-07-24
    21:23:50.000000000 +0200
+++ linux-4.7/arch/arm/boot/dts/zynq-7000.dtsi  2016-10-03 15:54:35.228460164
     +0200
@@ -96,6 +96,51 @@
       rx-fifo-depth = <0x40>;
     };

+    etb@f8801000 {
+      compatible = "arm,coresight-etb10", "arm,primecell";
+      reg = <0xf8801000 0x1000>;
+      coresight-default-sink;
+      clocks = <&clkc 47>;
+      clock-names = "apb_pclk";
+      port {
+        etb_in_port: endpoint@0 {
+          slave-mode;
+          remote-endpoint = <&replicator_out_port0>;
+        };
+      };
+    };
+
+    ptm0@f889c000 {
+      compatible = "arm,coresight-etm3x", "arm,primecell";
+      reg = <0xf889c000 0x1000>;
+      cpu = <&cpu0>;
+      clocks = <&clkc 47>;
+      clock-names = "apb_pclk";
+      port {
+        ptm0_out_port: endpoint {
+          remote-endpoint = <&funnel_in_port0>;
+        };
+      };
+    };
```

Listing 3.4 – Patch developed for adding support for CS components in a Zynq SoC (Excerpt from patch).

PTM PTM registers need to be programmed in a specific way as shown in Figure 3.3. Registers are unlocked by writing `0xC5ACCE55` to ETMLAR (*ETM Lock access register*) register as shown in Table 3.3.

Table 3.3 shows the main registers needed to enable the PTM. The PTM can be configured in three different modes: trace all instructions, trace a range of addresses and trace everything except a given region (modes #1, #2 and #3 respectively). This feature is essential if the

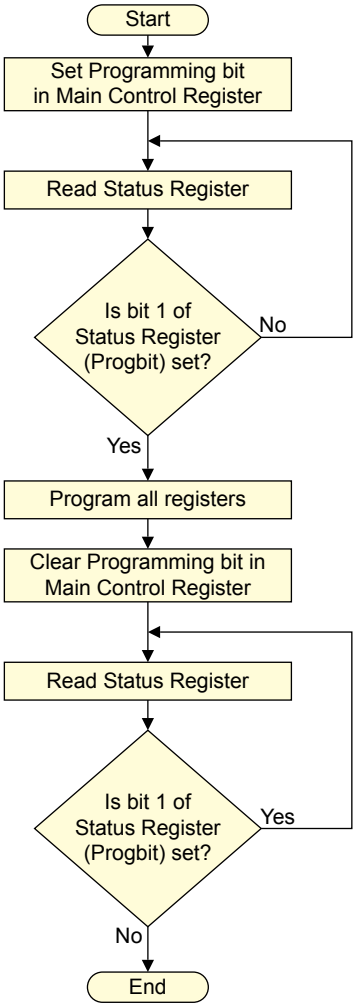


Figure 3.3 – PTM registers programming order (taken from [56]).

Table 3.3 – PTM registers, configuration values and purpose.

Register name	Value	Purpose
ETMLAR	0xC5ACCE55	Unlock PTM registers
ETMCR	1|(1«8)|(1«10)	Enable PTM features
ETMTRIGGER	0x6F	Events that capture trace
ETMTECR1	1«24	Trace all code
ETMTEEVR	0x6F	TraceEnable Event
ETMTRACEID	0x0F	Trace ID
ETMLAR	0	lock PTM registers

trace of a specific region such as .text is required. Table 3.4 presents the detailed configuration of PTM registers in order to enable each of these three modes.

Table 3.4 – PTM Configuration registers.

Register	Mode #1	Mode #2	Mode #3
ETMCR	1 ≪ 10 (Change programming bit alone)		
ETMCR	(1 ≪ 8)|(1 ≪ 12) (Activate other features)		
ETMTECR1	1 ≪ 24	0 ≪ 24	1 ≪ 24
ETMTEEVR	0x6F (Event ALWAYS TRUE)		
ETMACVR(n)	-	Start/stop address	
ETMACTR(n)	1		

In order to make sure that each change in the program flow is contained the in trace sent by the PTM, conditional execution must be disabled for instructions other than branches. Almost all instructions in ARM instruction set can be executed conditionally and can alter the program flow. However, these instructions are not traced by the PTM. As a result, an LLVM pass has been created to allow conditional execution only for branch instructions. All other conditional instructions (for instance, `addeq`) are converted to a conditional branch (`bne`) followed by a normal instruction (`add`). This way, each change in the program flow is contained in the execution trace generated by the PTM: in other words, it can be determined which basic block is currently being executed.

3.3.2 Trace Link

The funnel and the replicator (Figure 3.4) transport trace between source and sink components. The funnel takes care of merging trace received from multiple sources into a single one: if two Cortex-A9 cores need to be traced, both PTMs must be enabled. Furthermore, a source ID is assigned by a trace source component in order to identify each core from the funnel point of view: it allows multi-core debug tracing [57]. The replicator duplicates trace sent by the funnel and forwards it to sink components.

Table 3.5 – Funnel registers, configuration values and purpose.

Register	Value	Purpose
CSTFLAR	0xC5ACCE55	Unlock the funnel registers
CSTF Control	1	Enable input for trace 0
CSTFLAR	0	lock the funnel registers

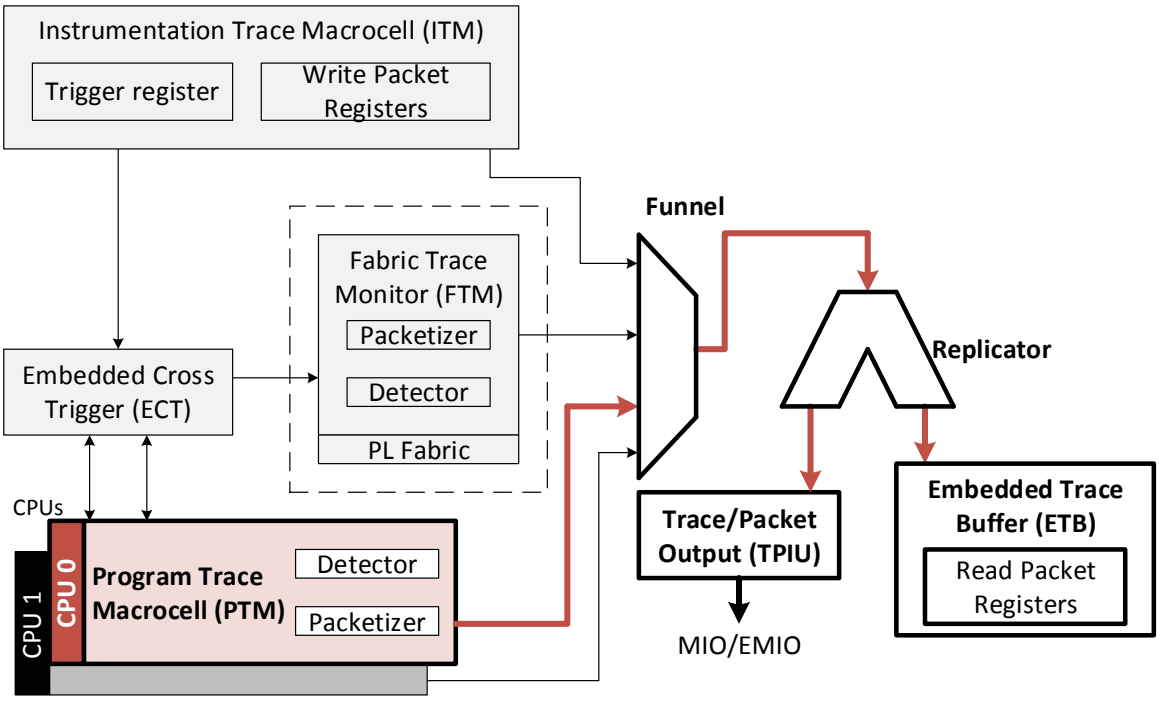


Figure 3.4 – CS components with trace trajectory shown by red bold lines.

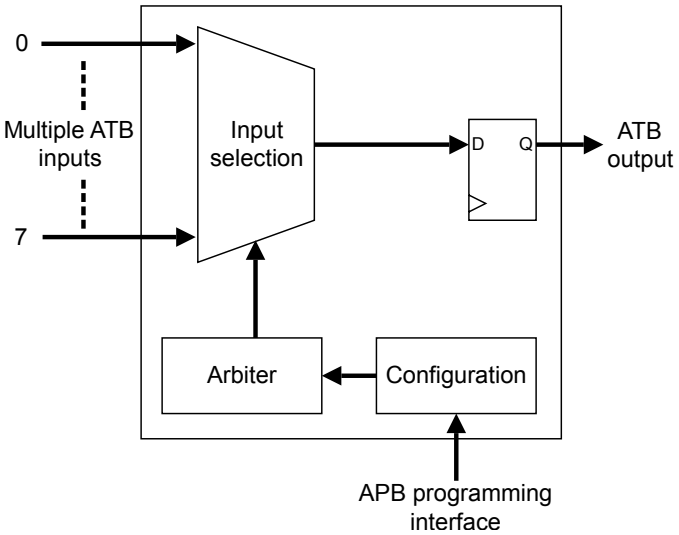


Figure 3.5 – CS Funnel (Link class component).

3.3.3 Trace Sink

Trace can be transmitted to two components: the ETB is a small 4KB memory where trace can be stored, while the TPIU is a module able to export trace towards the reconfigurable logic or towards a trace analyzer unit.

Figure 3.6 shows the program flow to follow in order to use the ETB or the TPIU to trace a program. The ETB and the TPIU have a similar set of registers. Table 3.6 shows the ETB configuration registers to be programmed with corresponding values. Once the ETB registers have been programmed, the PTM can be configured and enabled. Once these components are enabled, the program to be traced can be executed. When the program is terminated, tracing can be disabled by resetting the PTM. Then, a test is performed in order to make sure that trace is available in sink component before reading it.

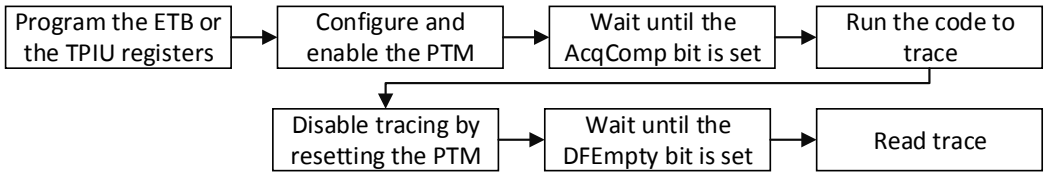


Figure 3.6 – Program flow for programming CS components and retrieving trace.

Table 3.6 – ETB registers, configuration values and purpose.

Register	Configuration values	Purpose
ETBLAR	0xC5ACCE55	Unlock the ETB registers
ETBFFCR	(1≪8|1≪9|1≪10)	Enable the ETB features
ETBCONTROL	1	Enable the ETB Trace Capture
ETBLAR	0	Lock the ETB registers

The TPIU needs to be enabled by software and hardware configurations. Figure 3.7 shows how the hardware configuration of the TPIU is done using Xilinx Vivado tools. The EMIO (*Extended Multiplexed Input Output*) interface is used to export trace to the FPGA part. The other option is to use the MIO (*Multiplexed Input Output*) interface in order to send trace towards external pins. The TPIU hardware configuration requires a clock. ARM CS Component user guide [57] propose two frequencies : 125 MHz or 250 MHz. If the frequency of 125 MHz is selected, trace is sent at both rising and falling edges of the clock. Figure 3.8 shows how to configure the clock speed for the TPIU using Xilinx Vivado tools.

The Linux kernel driver for the TPIU was incomplete and could not be used to program the TPIU. Therefore, a patch has been developed in this work for using the TPIU component (shown in Listing 3.5) [17].

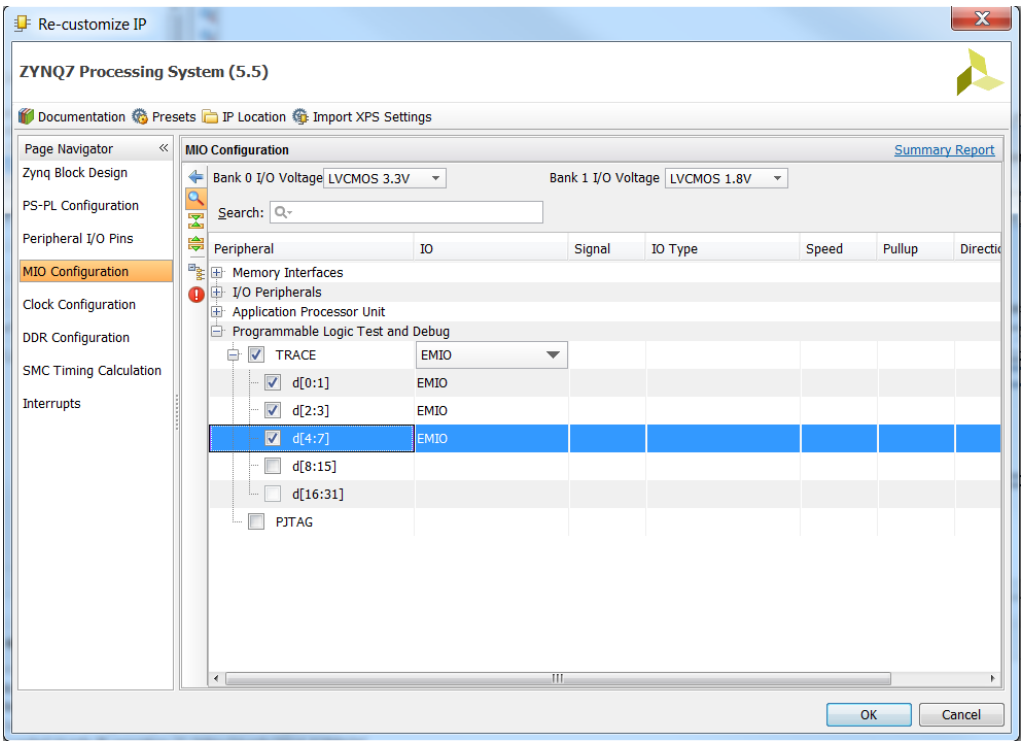


Figure 3.7 – TPIU trace connected to EMIO using Xilinx Vivado.

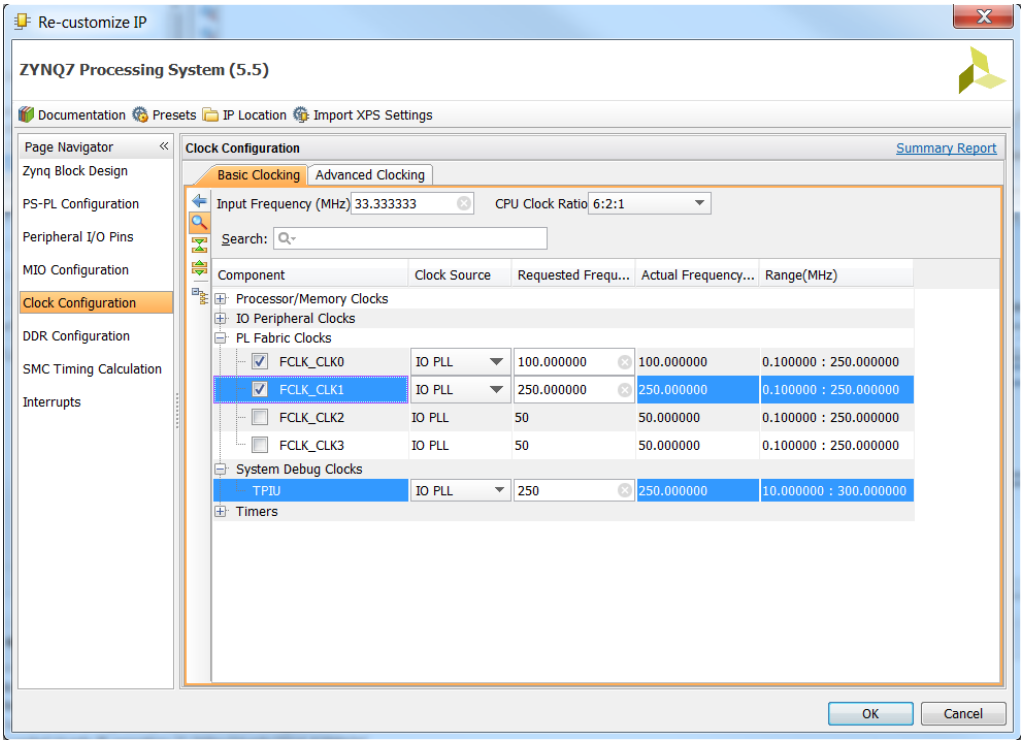


Figure 3.8 – TPIU trace clock configuration using Xilinx Vivado.

```
 static void tpiu_enable_hw(struct tpiu_drvdata *drvdata)
@@ -67,7 +83,10 @@
  CS_UNLOCK(drvdata->base);
-
+ /* set trace port width */
+ writel_relaxed(1<<7, drvdata->base + TPIU_CURR_PORTSZ);
+ /* clear formatter control register */
+ writel_relaxed(0, drvdata->base + TPIU_FFCR);
  CS_LOCK(drvdata->base);
 }

@@ -84,11 +103,28 @@
 static void tpiu_disable_hw(struct tpiu_drvdata *drvdata)
 {
  CS_UNLOCK(drvdata->base);
+ unsigned int ffcr;
+ ffcr = readl_relaxed(drvdata->base + TPIU_FFCR);
+ /* stop formatter when a stop has completed */
+ ffcr |= TPIU_FFCR_STOP_FI;
+ /* Clear formatter control reg. */
+ writel_relaxed(ffcr, drvdata->base + TPIU_FFCR);
+ /* Generate manual flush */
+ ffcr |= TPIU_FFCR_FON_MAN;
+ writel_relaxed(ffcr, drvdata->base + TPIU_FFCR);
+
+ if (coresight_timeout(drvdata->base, TPIU_FFCR, TPIU_FFCR_BIT, 0)) {
+     dev_err(drvdata->dev,
+         "timeout observed when probing at offset %#x\n",
+         TPIU_FFCR);
+ }
+
+ if (coresight_timeout(drvdata->base, TPIU_FFSR, TPIU_FFSR_BIT, 1)) {
+     dev_err(drvdata->dev,
+         "timeout observed when probing at offset %#x\n",
+         TPIU_FFCR);
+ }

- /* Clear formatter controle reg. */
- writel_relaxed(0x0, drvdata->base + TPIU_FFCR);
- /* Generate manual flush */
- writel_relaxed(FFCR_FON_MAN, drvdata->base + TPIU_FFCR);

  CS_LOCK(drvdata->base);
 }
```

Listing 3.5 – Patch developed for adding the CS TPIU driver in Linux kernel (Excerpt from patch).

Listing 3.5 shows that the registers are configured with the values shown in Table 3.6. The function `coresight_simple_func` is used to provide access from `sysfs` to read or write configuration registers.

3.4 Proposed approach

In [4], Heo et al. showed that the instrumentation time required to send information to the FPGA part is the major reason for slowdown in their proposed solution for DIFT. Therefore, a DIFT implementation is efficient when required information is obtained in the shortest possible time. To monitor information flows, at least four pieces of information are required to compute tags propagation:

1. PC register value.
2. Instruction encoding.
3. `load`/`store` memory addresses.
4. Basic system calls and arguments.

PC register value and some memory addresses are partially recovered using CS components described in Subsection 3.4.1. Missing information about memory addresses and instruction encoding is obtained through static analysis (explained in subsection 3.4.2) and instrumentation (detailed in subsection 3.4.3). Information about system calls is recovered using KBlare [6] described in subsection 3.4.4.

3.4.1 CS components

CoreSight components are programmed as explained in section 3.3. Trace generated by the PTM is recovered on the FPGA using the TPIU via EMIO interface. Trace recovered from the sample code in Listing 3.1 is shown in Listing 3.6.

```
1 00 00 00 00 00 80 08 18  06 01 00 21 cb 04 08 34
2 06 01 00 21 cb 04 08 48  06 01 00 21 cb 04 08 5c
3 06 01 00 21 b3 04 08 78  06 01 00 21 b3 04 08 90
4 06 01 00 21 bf 04 08 98  06 01 00 21 b9 04 08 a4
5 06 01 00 21 ad 04 08 a8  06 01 00 21 86 6b a1 04
6 08 e4 06 01 00 21 cb 9b  d2 db 0d 01 00 00 00 00
7 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

Listing 3.6 – Raw trace of .text section related to the compiled binary of C code in 3.1.

Different PFT packets can be noticed (Table 3.7). In order to get branch addresses, trace needs to be decoded.

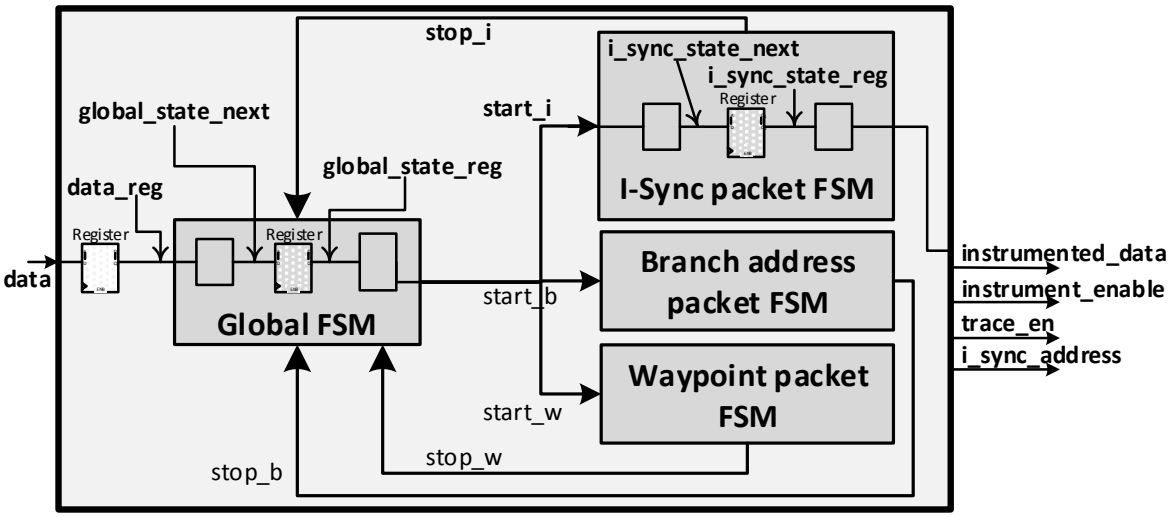


Figure 3.9 – Overall architecture of PFT Decoder.

3.4.1.1 PFT Decoder

The PTM generates trace according to the PFT protocol [56]. Table 3.7 presents PFT packets and their corresponding headers. As trace is recovered at the frequency of 250 MHz, it needs to be decoded at the same frequency in order to avoid unnecessary storage overhead.

Table 3.7 – PFT packet formats.

PFT packet name	Header	Remarks
A-sync	0x00 00 00 00 00 00 80	Alignment synchronization
I-sync	0x08 XX XX XX XX IB CA	Instruction synchronization packet contains PC value, information byte and cycle accurate count
Atom	0b1xxx xxx0	Precises whether a branch was taken or not taken
Branch address	0bCxxx xxx1	C is 1 if another byte follows, 0 otherwise.
Waypoint update	0x72	Indicates a change in program execution (e.g. exception)
Trigger	0x0C	Indicates a trigger condition has occurred.
Context ID	0x6E	Contains context ID of the process
VMID	0x3C	Contains VMID (Virtual Machine ID) value.
TimeStamp	0b0100 0x10	Holds timestamp value
Exception return	0x76	Generated on return from an exception handler
Ignore	0x66	Indicates insufficient trace

The PFT decoder receives the packet from the TPIU. A global FSM governs all these three packet FSMs as shown in Figure 3.9. I-Sync, branch address, and waypoint packets have their own packet FSMs. The other packets are decoded by the global FSM. The PFT architecture specification [56] states that trace starts with synchronization packets. Any other packet can follow these packets. It means that the next state logic of state machine must check for each type of packet. The PFT decoder is designed taking into consideration constraints of the PFT protocol.

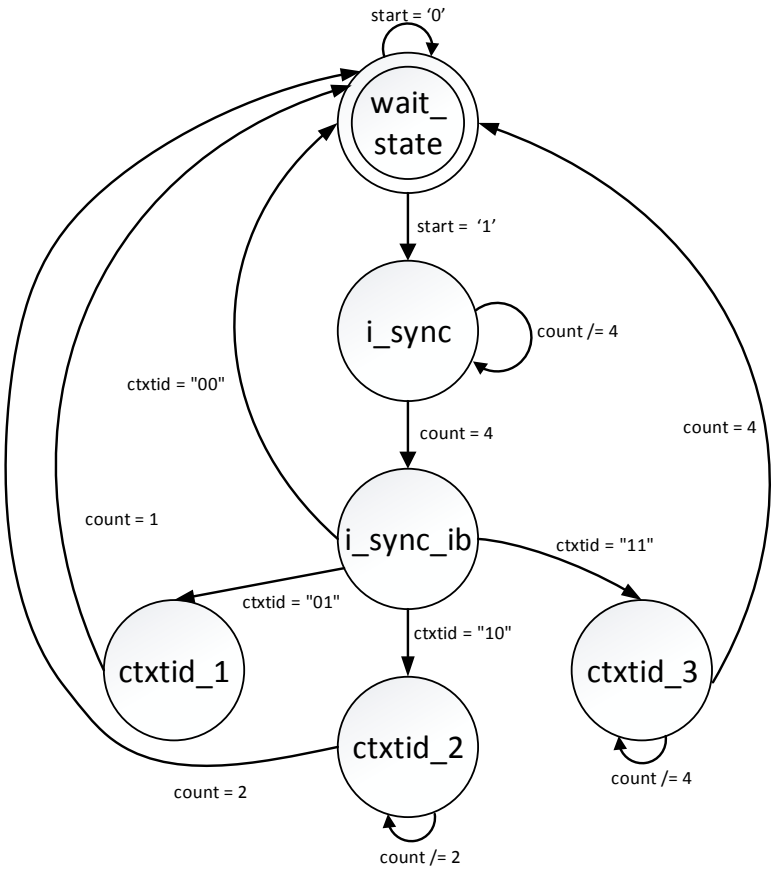


Figure 3.10 – State machine diagram for I-sync packet.

The most important ones are:

1. At each global FSM state transition, any PFT packet can be received.
2. For some packets (e.g. `bap`, `atom`), the size of the packet is unknown before trace is received.

The global state machine detects packet type and enables the corresponding packet FSM by setting start signal. There are other state machines for I-Sync, branch address, and waypoint packets. Figure 3.10 shows the state machine diagram for I-Sync packet decoding. The I-Sync packet FSM uses counters in state `i_sync`, `ctxtid_2` and `ctxtid_3`. The state machine is in `wait_state` by default. When the start signal is set, it goes into `i_sync` state and starts counting trace samples. Once four samples are received, the state machine goes into `i_sync_ib` state. If context ID tracing is enabled, it can send one, two or four bytes. Depending on the generic `ctxtid` value, the context ID packet is decoded. When the packet FSM finishes decoding packet, it goes back to `wait_state` and sends the stop signal to the global FSM which then looks for the next packet type.

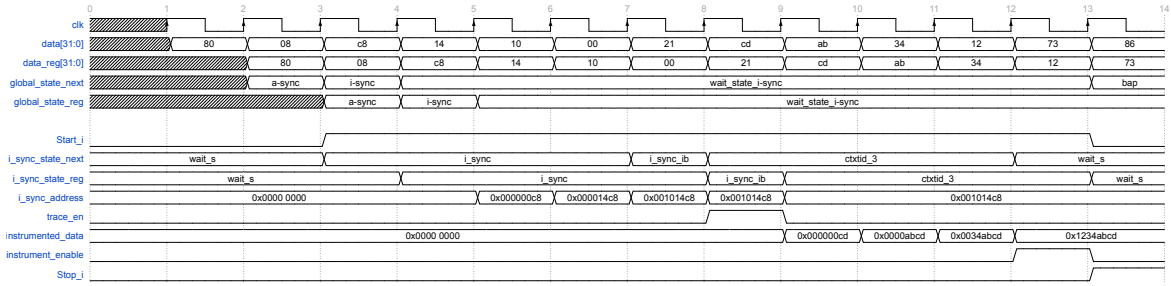


Figure 3.11 – Timing diagram of global FSM and I-Sync FSM of PFT decoder.

Figure 3.11 is a timing diagram showing how the PFT decoder works at each clock cycle. All inputs are registered: for instance, input `data` is registered to obtain `data_reg` signal. All FSMs work with the `data_reg` registered signal. When the global FSM detects an I-Sync packet, it enables a `start_i` signal (e.g. this is the case at the third clock cycle) to enable the corresponding slave FSM. Then, the packet FSM decodes the packet according to the PFT architecture as shown by `i_sync_state_reg`, `i_sync_state_next` and `i_sync_address` signals. In the meantime, the global FSM waits for a stop signal which is enabled by the packet FSM when the packet is decoded. Enabling `stop_i` signal at the ninth clock cycle modifies the global FSM state according to `data_reg` signal. It allows to decode the received trace on-the-fly. The outputs of the PFT decoder are also registered to avoid timing failures due to longer critical paths.

3.4.1.2 Implementation details

Hardware is designed using Xilinx Vivado. Synthesis and implementations are performed in order to generate the bitstream. Vivado also generates hardware specification files that are exported towards the Xilinx SDK tool. Then, C code is written for the targeted OS. Once the C code is compiled, the binary is either run using Xilinx SDK if standalone OS is used or it is copied to Linux kernel file system. The most important difference between standalone OS and Linux kernel is the address space used: standalone OS works on physical addresses whereas Linux kernel operates on virtual addresses.

The design shown in Figure 3.12 allows to recover raw trace from the FPGA part. Zynq7 Processing system IP is configured to send trace at 250 MHz using `FCLK_CLK1`. The utility vector logic is used to invert the `trace_ctl` output of processing system. This signal is equal to '0' when valid trace data is being sent on `trace_data` bus. It is inverted so that it equals '1' when `trace_data` is valid. As trace is sent at 250 MHz and 8 bits are sent, the maximum bandwidth of `trace_data` port is 250*8 = 2000 Mbits/s. Trace is stored in a true dual port BRAM (*Block RAM*) in order to read it back from processing system using AXI BRAM Controller IP. Trace shown in Listing 3.6 is recovered using this design.

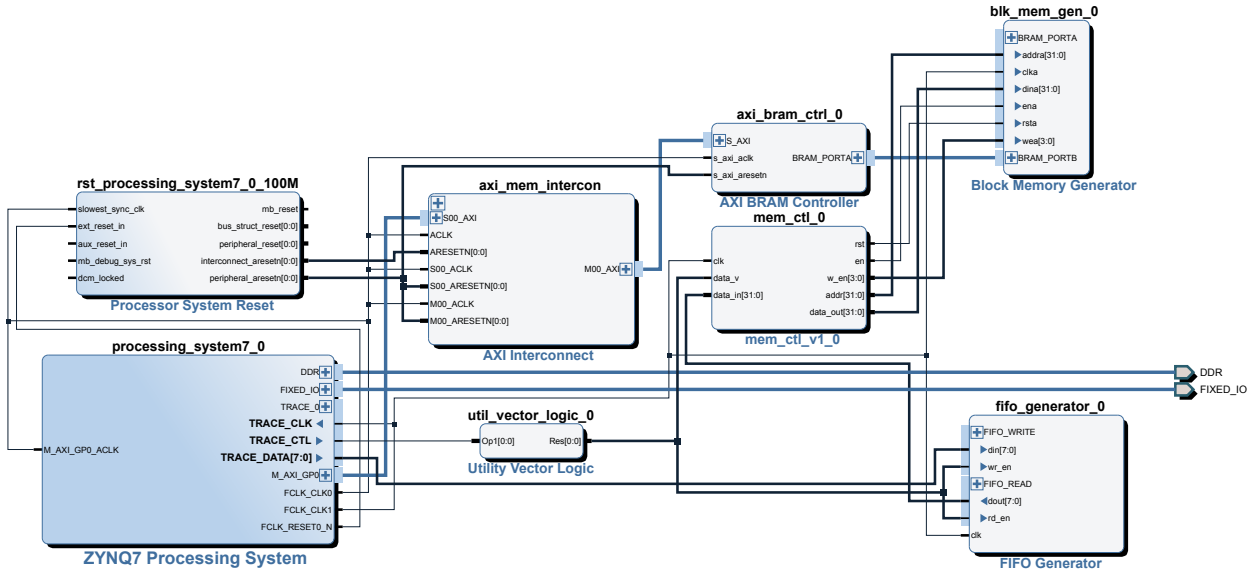


Figure 3.12 – Vivado design to recover trace on the PL (*Programmable Logic*). The Processing_system7_0 IP contains the configuration of ARM core and CS components. The util_vector_logic IP is used to implement a not gate in order to obtain an enable signal that equals '1' when valid trace data is available. Trace is written to blk_mem_gen_0 memory which can be read back by the ARM core using the second port. Trace is also stored in another FIFO (fifo_generator_0 IP) in order to be readable by the FPGA part.

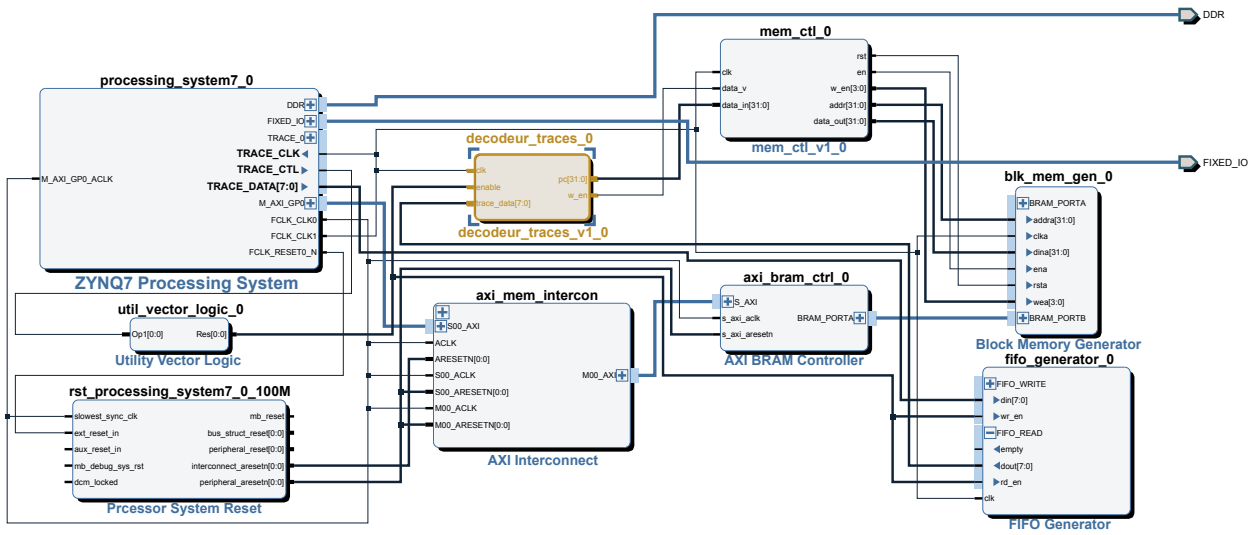


Figure 3.13 – This design shows how to use PFT Decoder. Trace is sent to the decodeur_trace IP which takes care of decoding trace packet in order to obtain control flow addresses of the program that is being executed on the ARM core. The decoded trace is stored in blk_mem_gen_0 IP to read it back from the ARM core using the second port of the BRAM.

The PFT decoder is tested using the design presented in Figure 3.13. The clock `FCLK_CLK1` is set to 250 MHz in order to recover and decode trace. The clock `FCLK_CLK0` is set to 125 MHz. BRAM is used in its true dual port configuration making it possible to write decoded trace using one port and read it back using the second port. Decoded trace shown in Listing 3.2 is recovered using this design.

3.4.2 Static analysis

Table 3.2 shows a sample code used to illustrate how static analysis is used in this thesis. Instructions at address `860c`, `8614`, `8618`, `861c` and `8624` do not produce any trace. In order to recover information about all other instructions not contained in the trace, the source code is statically analyzed before program execution. The static analysis will generate a tag dependencies instruction also called annotation that must be executed by the DIFT monitor.

Table 3.8 – Example annotations.

Example Instructions	Annotations
`sub r0, r1, r2`	$\underline{r0} = \underline{r1} + \underline{r2}$
`mov r3, r0`	$\underline{r3} = \underline{r0}$
`str r1, [PC, #4]`	$\underline{@Mem(PC+4)} = \underline{r1}$
`ldr r3, [SP, #-8]`	$\underline{r3} = \underline{@Mem(SP-8)}$
`str r1, [r3, r2]`	$\underline{@Mem(r3+r2)} = \underline{r1}$
(a)	**(b)**

Table 3.8 shows a code which does not produce any trace (a) and corresponding annotations (b). $\underline{r}$ is used to denote the tag of register `r`. For instance, for the first instruction in Table 3.8(a), the corresponding annotation, shown in Table 3.8(b), is to associate tags of operands `r1` and `r2` towards the tag of destination register `r0`. Annotations are generated by analyzing the assembly representation. The analysis can be done using the binary file (described in subsubsection 3.4.2.1) or using program source code (described in subsubsection 3.4.2.2).

3.4.2.1 Method 1: static analysis after compilation using Capstone disassembly engine

Figure 3.14 shows the schematic of the overall approach using the Capstone disassembly engine [7]. The binary is compiled using a compiler (LLVM or GCC). In this thesis, LLVM is used as the main compiler. Once the binary is generated, it is disassembled using the Capstone disassembly engine and annotations are generated for each instruction in the disassembled binary. Another tool spedi [58] is used in order to recover all basic blocks of the binary.

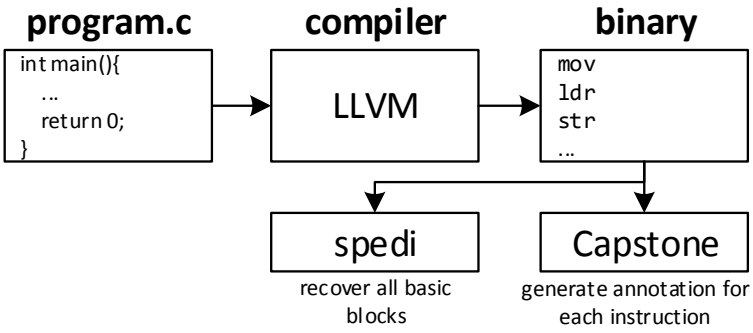


Figure 3.14 – Overall schematic of static analysis using the Capstone disassembly engine [7].

Details Annotations specify the operation to be realized by the DIFT monitor. For each instruction, operands type (register, memory or immediate) and their number or value are determined. By using operands type, information flows that take place between the operands can be determined. For instance, if the first instruction of Table 3.8 (a) is considered, it can be noticed that the three operands are registers. By determining registers number, the information flow can be determined from source registers towards destination register. For all instructions that do not produce trace, instruction encoding and semantics are analyzed in order to generate annotations. Furthermore, an instruction is added to keep track of the SP (*Stack Pointer*) value by DIFT monitor. It is done for each CPU instruction that changes the SP value directly (e.g. `sub SP, SP, #4`) or indirectly (e.g. `push`,`pop`).

Algorithm 1 illustrates how annotations are generated using the Capstone disassembler[7]. The idea is to analyze each instruction encoding to find operands and determine information flows between these operands. This algorithm is applied to all instructions of the application. Information flows obtained through this process are encoded for an hybrid monitor and stored in a memory section.

Line 2 shows that all instructions are analyzed one by one. Some special instructions are analyzed separately on line 14 while all other instructions are analyzed from line 4 to 13. Line 3 shows that a function `RecoverOperands`, detailed in Algorithm 2, is called to analyze instruction encoding in order to recover operands. This function returns `operandsType` and

corresponding `operands` value. There are multiple different cases but only three cases are shown as an example.

```
   Input  : basic block
   Output : annotations (annot)
1  i = 0; operandsType = []; operand = [];
2  foreach instruction instr in basic block do
3      operandsType[], operand[] = RecoverOperands (instr);
4      if instr != (push OR pop OR ldm OR stm) then
5          switch ⋃_i operandsType[i] do
6              case [Reg, Reg] do
7                  if operand[1] != operand[2] then
8                      annot = operand[1] ← operand[2];
9              case [Reg, Reg, Reg] do
10                 operand[1] = operand[2] OR operand[3];
11             case [Reg, mem] do
12                 annot = operand[1] ← Mem[operand[2]];
13             case ... do
14     else
15         annot = generateAnnotations (instr);
16     InstructionChangesSP (instr) // Strategy 2 only
```

Algorithm 1: Algorithm for generating annotations.

If the instruction has two operands and both are registers (line 6 to 8) then the information flow takes place from the destination register towards the source register. Furthermore, if the destination register is the same as the source register, there are no information flows to propagate. Therefore, the `if` operation on line 7 checks whether both registers are different. Similarly, if the instruction has three register operands (line 9 to 10), then the corresponding information flow is to associate tags of source registers and store the result into the tag of the destination register. If the instruction has two operands type: such as register and memory address operand (line 11 to 12), then the corresponding annotation is to associate the tag of memory address towards the tag of register.

```
1  Function RecoverOperands
       Input: instruction (instr)
       Output: Operands type (operandsType[]),
               Operands (operand[])
2      foreach operand in instr do
3          switch operand do
4              case reg do
5                  operandsType[i] = reg;
6                  operand[i] = reg_number;
7              case imm do
8                  operandsType[i] = imm;
9                  operand[i] = imm_value;
10             case mem do
11                 operandsType[i] = mem;
12                 operand[i] = mem_offset;
                   // instrumentation
13         i++;
```

Algorithm 2: RecoverOperands function.

The special instructions (`push, stm, ldm, pop`) are dealt from line 14 to 15. The `generateAnnotations` function takes care of instructions analysis and annotations generation. Line 16 contains an optional analysis that takes care of detecting instructions changing the `SP` register value. This analysis, detailed in Algorithm 3, is an optimization and is required in order to reduce the number of instrumented instructions.

Algorithm 1 considers only instructions that are part of the ARM instruction set. The same algorithm can be applied as well to the Thumb instruction set but minor differences may exist and it may need to be adapted. Furthermore, the algorithm shown here works for architecture ARM v7-A. It can be easily adapted to any other ARM architecture by taking into consideration minor differences that exist between instruction sets.

```
1  Function InstructionChangesSP
       Input: instruction (instr)
       Output: Stack pointer offset
2      Function InstructionChangesSP(instr)
3          count = 0;
4          switch instr do
5              case Reg do
6                  if (isAdd && reg_dst == SP) then
7                      SP = SP + offset;
8                  else
9                      (isSub && reg_dst == SP)
10                 SP = SP - offset;
11             case mem do
12                 if (isLoad && reg_src == SP) then
13                     SP = SP + offset;
14                 if (isStore && reg_src == SP) then
15                     SP = SP - offset;
16                 if isPush then
17                     SP - offset;
18                 if isPop then
19                     SP + offset;
20         return SP;
```

Algorithm 3: InstructionChangesSP function.

The `RecoverOperands` function on line 3, detailed in Algorithm 2, determines operands for each instruction. For each operand, its type (register, immediate, or memory) and its value (register number, immediate value or memory address) are determined and stored in two lists `operandsType` and `operand` respectively.

Limits The proposed method in section 3.4.2.1 analyzes only explicit information flows. Analyzing information flows require keeping track of the `PC` register value and propagate the tag of `PC` inside loops. Furthermore, this method uses a disassembler which has its own limitations. For instance, the code obfuscation renders any kind of static analysis very difficult.

3.4.2.2 Method 2: static analysis during compilation using LLVM

Static analysis can be done during compilation. LLVM compiler [59] provides more information than a disassembler making it more suitable for static analysis. Furthermore, it avoids previous approach limitations of method #1 by providing implicit information flows support while being able to generate information flows for obfuscated code.

3.4.3 Instrumentation IP

Decoded trace allows resolving which basic block is currently being executed. Static analysis allows determining information flows inside each basic block. However, some information flows cannot be determined statically. For instance, if a load (`ldr`) instruction is statically analyzed, the memory address cannot be determined. Therefore, missing `ldr`/`str` memory addresses are recovered using instrumentation. An IP named *instrumentation IP* is used to receive instrumented memory addresses.

In table 3.8, `ldr` and `str` instructions contain memory addresses. These addresses need to be known in order to propagate the associated tags. There are three types of memory instructions in the ARM instruction set:

(i) PC-relative (e.g. 3^{rd} instruction of Table 3.8)
(ii) SP-relative (e.g. 4^{th} instruction of Table 3.8)
(iii) Register-relative (e.g. 5^{th} instruction of Table 3.8)

Two strategies can be designed in order to recover addresses contained in memory instructions. Both strategies provide the same code coverage as related work instrumentation techniques.

- **Strategy 1**. Each memory instruction is instrumented in order to send memory address(es) to the DIFT monitor.
- **Strategy 2**. From all memory instructions, only register-relative instructions are instrumented. The DIFT monitor knows the PC register value thanks to the decoded trace. Therefore, PC-relative memory instructions can be omitted in instrumentation. Furthermore, the SP-relative memory instructions do not need to be instrumented. All SP value changes can be tracked thanks to annotations obtained during static analysis (optional analysis in Algorithm 1).

3.4.3.1 Software considerations for instrumentation IP

The instrumentation IP has a memory-mapped register that is used to receive instrumented memory addresses. This memory-mapped address needs to be mapped into the virtual memory space of the application and stored in a register in order to be used for instrumentation.

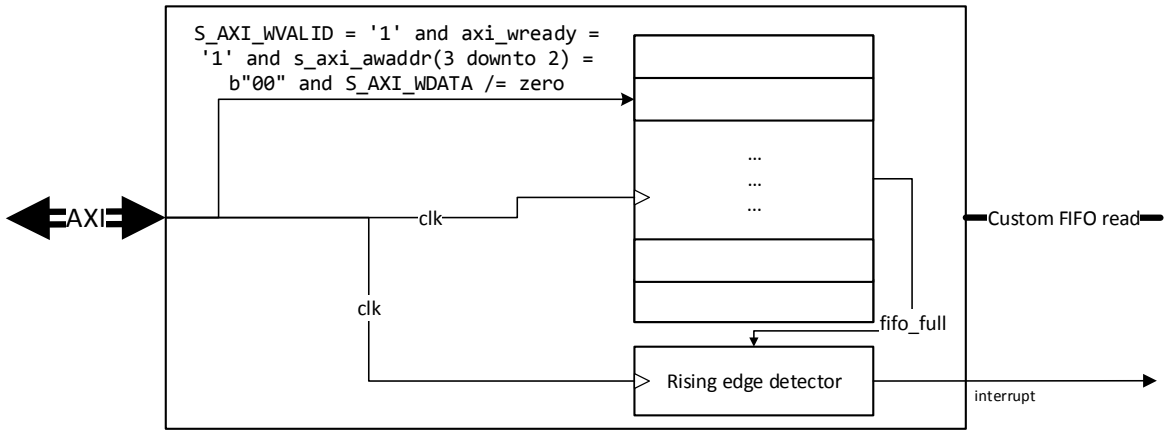


Figure 3.15 – Simplified schematic of the instrumentation IP.

For instance, it can be done using `mmap` syscall and by reserving a register during compilation so that it is not used by the application.

```
instrumentation_buffer: instrumentation@43c10000 {
        compatible = "generic-uio";
        interrupt-parent = <&intc>;
        interrupts = <0 30 4>;
        reg = <0x43c10000 0x1000>;
};
```

Listing 3.7 – Device tree entry for the instrumentation IP.

Another modification is to add an entry for the instrumentation IP in the device tree, as shown in Listing 3.7, so that the kernel can enable the right driver for this IP during boot. The device tree entry tells the kernel to use the `generic-uio` driver developed. Furthermore, the `interrupts` field specify the interrupt number that the kernel needs to associate with this IP. Using this number, the software handler can be programmed.

3.4.3.2 Implementation details for instrumentation IP

Figure 3.15 shows the architecture of the instrumentation IP. It mainly contains a FIFO and a rising edge detector entity. The IP is written through an AXI bus interface and is read using custom FIFO interface. The FIFO is written when the following condition (`S_AXI_WVALID = ‘1’ and axi_wready = ‘1’ and s_axi_awaddr(3 downto 2) = b"00" and S_AXI_WDATA /= zero`) is true. This condition allows to determine whether the instrumented data is available through AXI bus. When a read request is received through the custom FIFO read interface, it is transferred to the FIFO which outputs the data read and forwards it to the output.

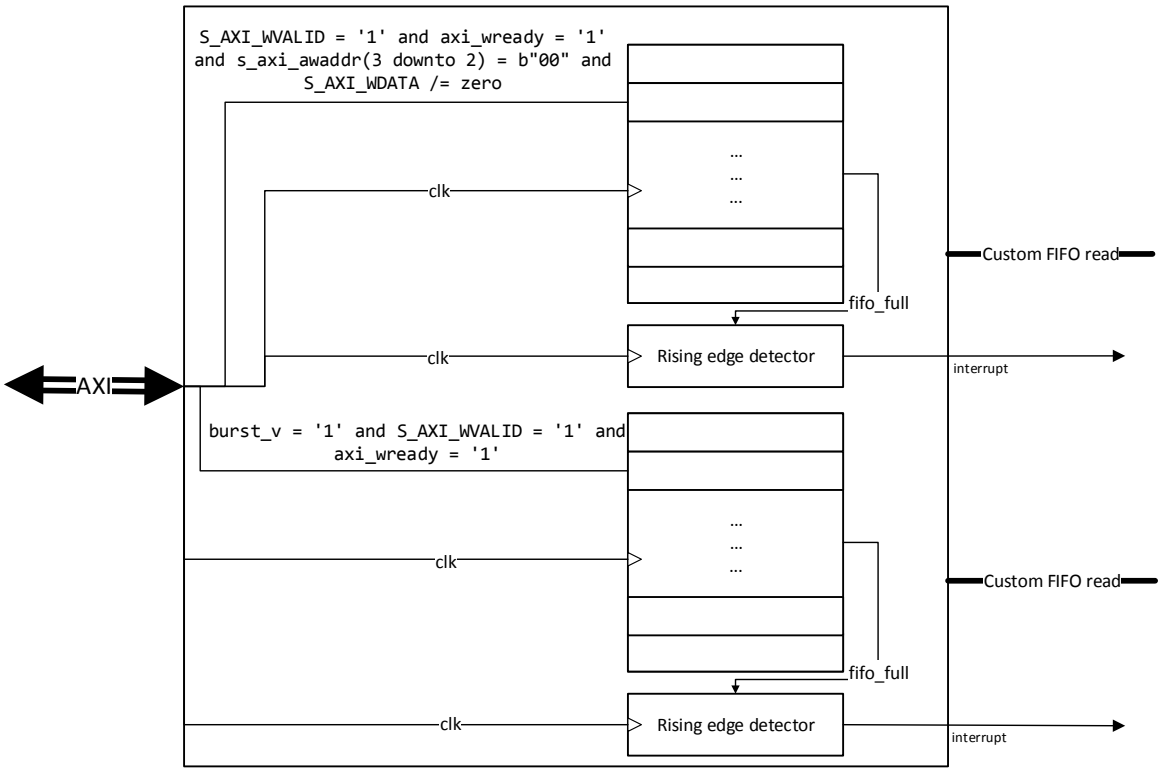


Figure 3.16 – Simplified schematic of the optimized instrumentation IP.

When the FIFO is full, a signal enables the rising edge detector entity which enables an interrupt to CPU signal. Using this interrupt signal, the CPU can put the process in wait state using a custom interrupt handler. Figure 3.16 shows an optimized version of the instrumentation IP. Writing instrumented memory addresses one by one implies high runtime overhead. Therefore, in order to decrease the instrumentation runtime overhead, memory addresses are written by bursts of multiple (two) memory addresses. The IP was modified in order to add another FIFO and a rising edge detector entity in order to receive burst memory addresses.

3.4.4 OS-related information

Most of the existing works do not precise how information from the OS is sent to the DIFT monitor. However, it is mandatory to explain kernel modifications in order to provide an end-to-end approach. Therefore, all kernel-related modifications and hardware IPs required in order to provide that support are described. Table 3.9 shows the registers available in FIFO IPs (**kernel2monitor** and **monitor2kernel**) with their relative addresses. There are three registers available:

- The first register allows the ARM CPU to read the FIFO status in order to determine whether the values can be written (if FIFO not full) or read (if FIFO not empty).
- The second register allows clearing the interrupt by writing any value to it. The hardware detects a write to this register to clear the interrupt.
- The data is written to or read from the third register.

Table 3.9 – Registers for kblare FIFO IPs.

Register number	Register name	Relative address	Format
1	FIFO status	0x00000000	31 … 3 | 2 | 1 | 0 - | err | full | empty
2	Clear interrupt	0x00000004	31 … 0 Write any value to clear interrupt
3	Data	0x00000008	31 … 0 Data to write to FIFO
		0x0000000c	31 … 0 Data to write to FIFO
		0x00000010	31 … 0 Data to write to FIFO

Listing 3.8 shows an example code where OS-related information is required. `Read` and `write` functions used in the program require kernel support. For instance, `read` function reads the content of the file and copies it into a buffer. On the FPGA side, the DIFT monitor needs to know the tag value of the file being read by the program so that it can initialize the memory address of `buffer` with the correct tag. This information cannot be recovered statically as the tag can be modified at runtime. The tag is stored in extra attributes of the root file system and its value can be modified by another running process.

```
1 #define SIZE 6
2 char buffer[SIZE+1];
3 int fd = open("testfile.txt", O_RDWR);
4 size_t sz = read(fd, buffer, SIZE);
5 // modify buffer
6 write(fd, buffer, SIZE);
```

Listing 3.8 – Example code for OS-related information. The `open`, `read`, and `write` on lines 3, 4, and 5 respectively require an information from the kernel in order to determine the tag.

3.4.4.1 Kernel to monitor

The `kernel2monitor` IP sends information about tag initialization of memory addresses. Listing 3.8 shows that kernel support is required. The `read` function call, on Line 4, needs to send the buffer address, its size and the tag of the file being read to the DIFT monitor.

Using these three pieces of information, the DIFT monitor can initialize all memory addresses starting from the buffer address until size with the tag value provided. Three pieces of information need to be sent: as a consequence, `kernel2monitor` IP has three registers as shown in Table 3.9.

3.4.4.2 Monitor to Kernel

The `monitor2kernel` IP recovers tag value of memory addresses. The `write` function call in Listing 3.8 writes the provided buffer into the file. Writing buffer contents to the file also requires the write of buffer tag to the extra attributes of the file. However, the buffer tag is known only by the DIFT monitor. Therefore, the `monitor2kernel` IP allows recovering the tag of buffer. The kernel writes the buffer address and the size to the first two registers and then waits for the tag value. The DIFT monitor looks up for the tag value and writes it to the third register. When the tag is written, the process can update the extra attributes of the file and carry its execution.

3.4.4.3 Software considerations

Similarly to the instrumentation IP, the hardware IPs need to be added to the device tree. Listing 3.9 shows the new device tree entries.

```
kernel2monitor: kernel2monitor@43c00000 {
        compatible = "kblare-ip-ps2pl";
        interrupt-parent = <&intc>;
        interrupts = <0 32 4>;
        reg = <0x43c00000 0x1000>;
};

monitor2kernel: monitor2kernel@43c20000 {
        compatible = "kblare-ip-pl2ps";
        interrupt-parent = <&intc>;
        interrupts = <0 31 4>;
        reg = <0x43c20000 0x1000>;
};
```

Listing 3.9 – Device tree entry for OS-related information.

3.5 Evaluation of the proposed approach

3.5.1 Evaluation goals

The main goal in this section is to study feasibility i.e. make sure that all information required for DIFT is being sent to the FPGA part. This study is done in five steps:

- Evaluating the time overhead of CS components.
- Evaluating the time overhead of instrumentation (number of instructions added).
- Evaluating why the proposed strategies allow to recover required information for DIFT.
- Evaluating the efficiency of proposed strategies compared to related works.
- Describing area and power overheads of each IP developed.

3.5.2 Results evaluation method

Implementations were done with Vivado 2017.1 tools on a Xilinx Zedboard [1] including a Z-7020 SoC (dual-core Cortex-A9 running at 667MHz and an Artix-7 FPGA). The FPGA logic has around 85K logic cells and 560 KB of Block RAMs. The DIFT monitor is implemented in a Microblaze softcore for this proof-of-concept. The evaluation method is explained in Appendix B.

3.5.3 Time overhead analysis

3.5.3.1 CS components overhead

MiBench applications were tested with and without enabling CS components. The negligible time difference observed in Figure 3.17 is introduced by non-deterministic events (such as context switches). Vergé et al. [60] showed that an execution time overhead can occur if trace is stored in the ETB. In this work, as shown in Figure 3.17, the time overhead of CS components is negligible for two main reasons. First of all, the CS PTM trace component is non-intrusive as it operates in parallel on a list of committed CPU instructions. The second reason is due to the configuration of CS components: the TPIU is used as a trace sink rather than the ETB.

3.5.3.2 Static analysis

Static analysis does not add execution time overhead as it is performed before executing the application. However, it adds a static binary size overhead that corresponds to the storage of

[1] The ZedBoard development board is presented in Appendix A

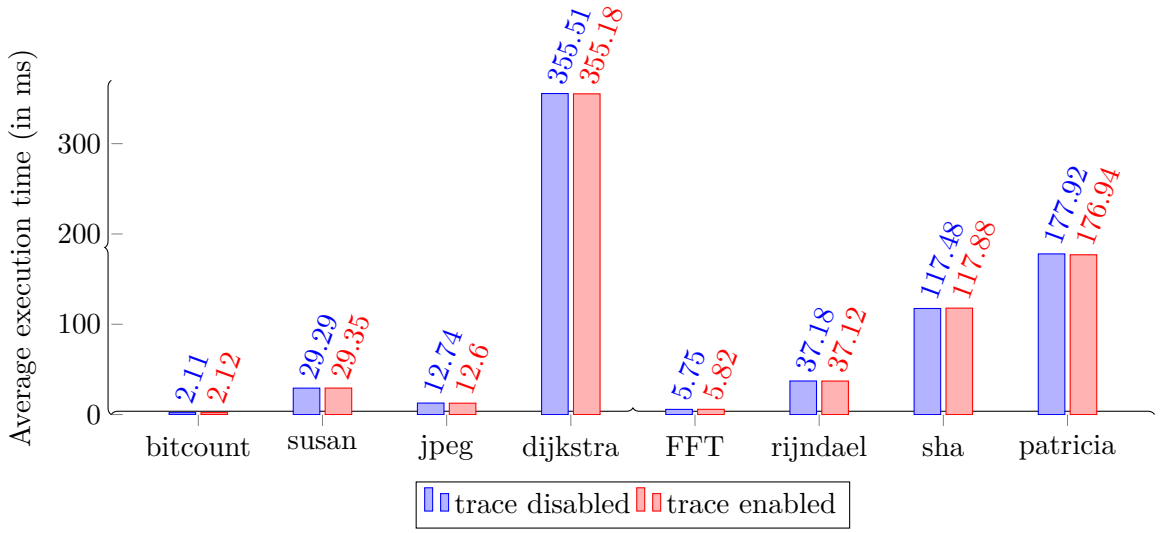


Figure 3.17 – CoreSight components time overhead for MiBench benchmark.

dependencies (i.e. output of static analysis). The communication overhead of the approach developed in this thesis is only due to code instrumentation.

3.5.3.3 Instrumentation time overhead

Assumptions: The instrumentation time overhead is measured by taking two important assumptions:

1. The library code does not need to be instrumented. This is the case in most existing works i.e. information flows originating from library code are not taken into account. In other words, the library code is considered to be secure.
2. The runtime overhead of instrumentation, also called instrumentation time overhead, is directly proportional to the number of instrumented instructions in a program.

Figure 3.18 shows the percentage of instrumented instructions over different strategies. In [4], in order to recover memory addresses, each branch instruction (`b/beq/bne/...`, `bl/blx`) as well as memory instructions are instrumented. Furthermore, another instruction is added for each direct branch in order to detect changes in the program flow. This strategy is referred as *Related work instrumentation* in Figure 3.18.

In this thesis, there is no need to instrument branch instructions thanks to CS components. Then, there are two strategies. Strategy #1 consists in instrumenting all memory instructions while strategy #2 only takes care of memory instructions relative to registers. For both strategies, instructions are added to send needed information to the instrumentation IP implemented in the reconfigurable logic.

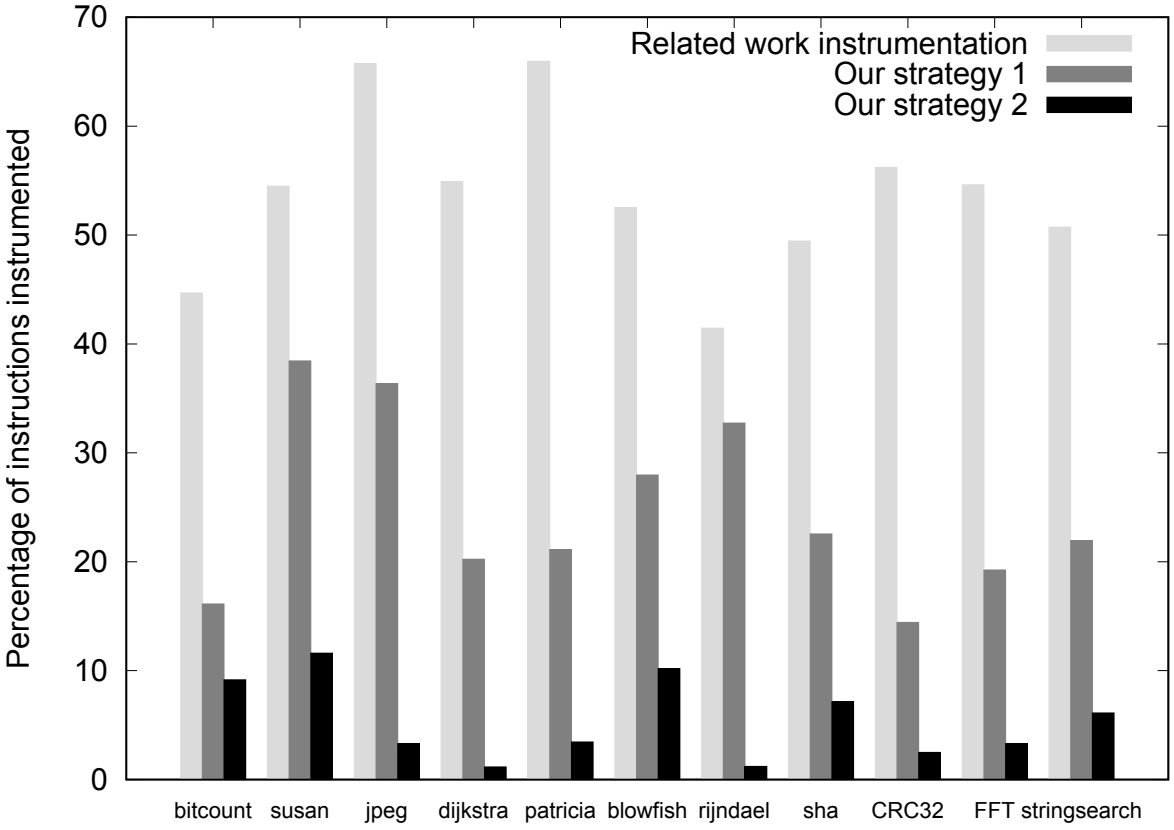


Figure 3.18 – Instrumentation overhead.

The average time overhead for strategy #1 is 24.6% while it reaches 53.7% for related work instrumentation strategy. The average time overhead for strategy #2 is 5.37% which is better than the overhead of 60 % reported by Heo et al. [4]. For some applications (such as `dijkstra`, `rijndael` or `CRC32`), the instrumentation overhead (for strategy #2) is less than 2%: it is due to the fact that there are few register-relative memory instructions in these applications.

3.5.4 Area overhead

Table 3.10 – Area results of developed IPs on Xilinx Zynq Z-7020.

IP Name	Slice LUTs	Slice Registers	BRAM Tile
PFT decoder	121 (0.23%)	231 (0.22%)	0
Instrumentation	676 (1.27%)	2108 (1.98%)	0
Monitor2kernel	662 (1.24%)	2106 (1.98%)	0
Kernel2monitor	662 (1.24%)	2106 (1.98%)	0
Total Available	**53200**	**106400**	**140**

Table 3.10 shows the post-implementation area results of developed IPs on Xilinx Zynq SoC. The area results show that the developed IPs take 4 % of FPGA area on Zedboard system.

3.5.5 Power overhead

To evaluate power overhead of the designed IPs, two hardware designs were realized.

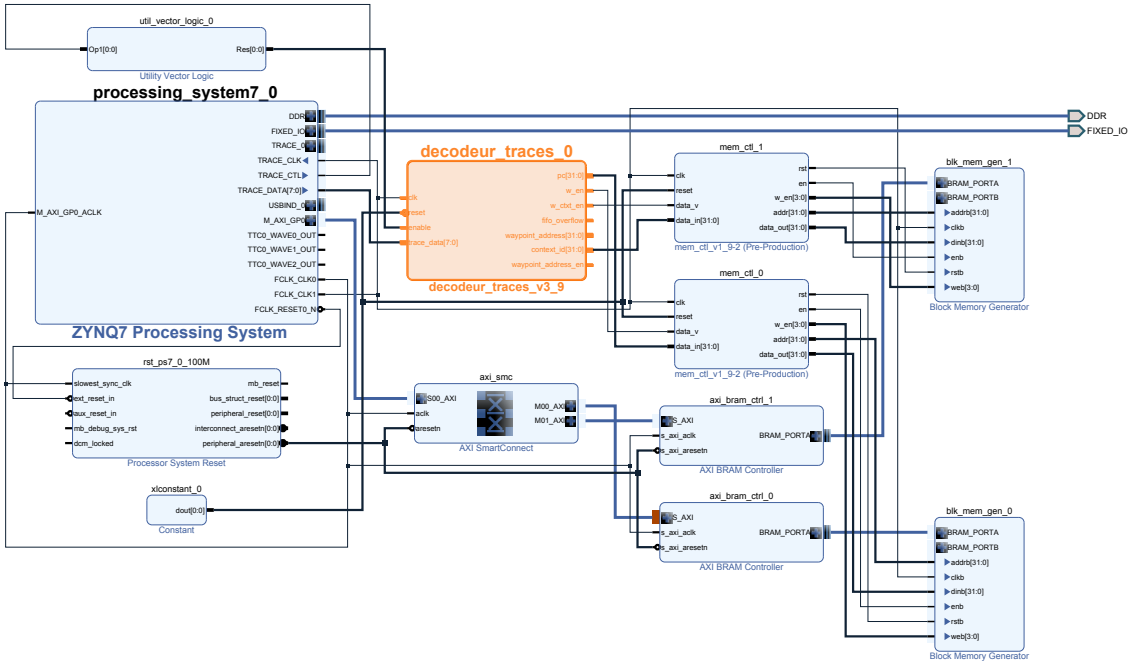


Figure 3.19 – Hardware design to evaluate power overhead of the PFT decoder.

Table 3.11 – Power results of PFT Decoder on Xilinx Zynq Z-7020.

	Power consumption (in Watt)
Clocks	0.028
Signals	0.018
Logic	0.015
BRAM	0.006
PS7	1.529

The first design (Figure 3.19) is used to evaluate the power overhead of the PFT decoder and the memory storage. Table 3.11 shows the power overhead of this design.

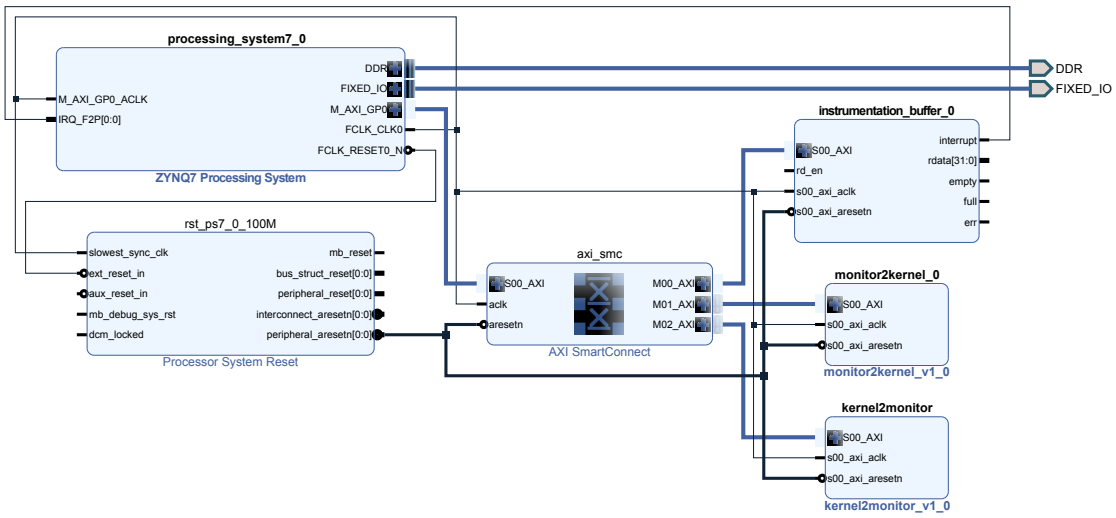


Figure 3.20 – Hardware design to evaluate power overhead of communication buffers.

The second one (Figure 3.20) is used to evaluate the power overhead of the communication buffers (Instrumentation, Monitor2kernel, Kernel2monitor).

Table 3.12 – Power results of communication buffers on Xilinx Zynq Z-7020.

	Power consumption (in Watt)
Clocks	0.0.022
Signals	0.019
Logic	0.016
BRAM	0.000
PS7	1.527

Table 3.12 shows the power overhead of this design.

All IPs designed in this chapter consume 0.124W while both designs are running at their nominal speed: 250 MHz for PFT decoder and 100 MHz for communication buffers.

3.6 Summary

This chapter presents the proposed strategy to recover required information for DIFT on ARM-based SoCs. The results presented in this work have been published in [61, 62]. The proposed approach takes advantage of ARM CoreSight components in order to recreate the CFG of the running application on the FPGA part. Trace generated by CoreSight PTM component and exported to the FPGA part through TPIU component via EMIO interface is decoded using the PFT decoder entity. It is shown that the recovery of trace do not slowdown the application running on the ARM core as CS components are non-intrusive.

The information flows that take place in between basic blocks of the CFG cannot be recovered using decoded trace. Therefore, static analysis allows to recover annotations that specify what operations are being done in each basic block of the application. Load/Store memory addresses cannot be obtained during static analysis. So, they are instrumented and recovered during runtime using instrumentation IP. Nonetheless there is still one piece of information is missing: information about syscalls that are handled by Linux kernel. The Blare Linux kernel monitor is used in order to recover this information and is sent to the FPGA part using monitor2kernel and kernel2monitor IPs. Once all this information is recovered on the FPGA, the DIFT monitor can be studied. The power overhead introduced by all these IPs is negligible when compared to that of the ARM core running the application.

Chapter 4

ARMHEx: First proof of concept

This chapter proposes an architecture that allows verifying the proposed scheme in chapter 3. An example security policy of data execution prevention is explained and implemented in order to evaluate the feasibility.

Contents

4.1 Introduction

The information required for DIFT on a hardcore CPU can be recovered using the methodology described in chapter 3. The goal of this chapter is to describe the first PoC (*Proof Of Concept*) design based on the proposed methodology, named ARMHEx (ARM Hardware Extension), implemented in order to make sure that all information required for DIFT is being recovered on a real system. In this chapter, the information related to the OS is not being recovered, as in existing related works, in order to simplify the test procedure and the comparison with previous works. Furthermore, existing designs target softcore CPUs reducing development times. However, in this thesis, Zynq SoC is the main target making it more challenging and more practical if an industry partner wants to implement the solution proposed in this work. First, the architecture is detailed in section 4.2. Then, the security of ARMHEx and its components is discussed in section 4.3 with the help of an example attack. Implementation details and results are explained in section 4.4. Section 4.5 presents a summary of this chapter.

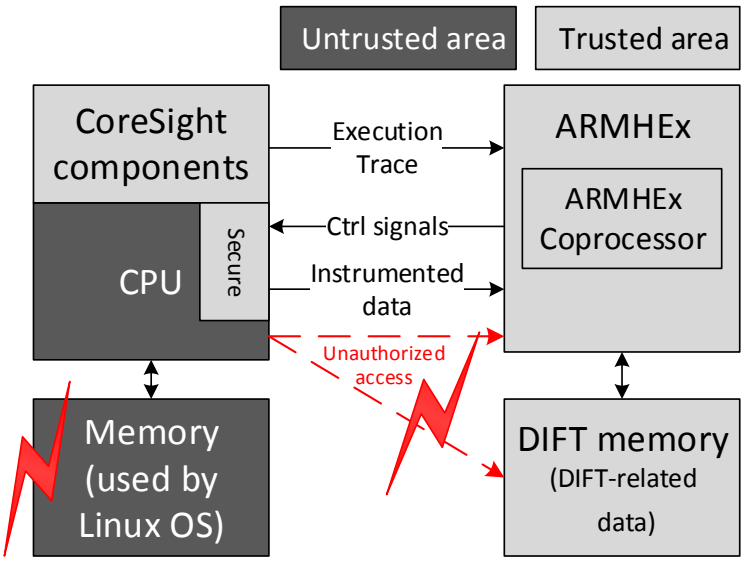


Figure 4.1 – Threat model.

4.2 Proposed architecture

4.2.1 Threat model

Figure 4.1 presents an overview schematic of an embedded system where ARMHEx is implemented. A default ARMHEx implementation is composed of a CPU (possibly multicore) and the ARMHEx extension which is implemented in the reconfigurable logic (both with a dedicated memory).

It is assumed that adversaries can launch a malicious application that contains software vulnerabilities. For instance, they can try to take advantage of buffer overflows, SQL injection, cross-site scripting or data leakage. Furthermore, adversaries may try to access the content of ARMHEx components or DIFT memory. For instance, an attacker can try mapping and writing to the DIFT memory section used by ARMHEx. In this work, following communication channels between CPU and ARMHEx are considered secure: (*Execution trace*, *Control signals*, and *instrumented data* shown in Figure 4.1). Furthermore, physical attacks such as side-channel attacks, probing, JTAG attacks, memory attacks such as cold boot, EM injections ... are not considered in this work.

The first goal in this work is to prevent from user-space software attacks. ARMHEx uses DIFT to monitor information flows at execution time in order to prevent from such software attacks. An example of a straightforward buffer overflow attack is shown in Listing 4.1.

The buffer overflow attack happens because of the use of `gets()` and `strcmp()` functions which do not enforce bounds checking. The user can type a value that overflows the buffer `buff` and can take advantage of privileges offered to the user who provides the right password.

The second goal is to make sure that the specified trusted area is inaccessible from the untrusted area through non-secure channels (colored in red in Figure 4.1). As an attacker may read or write to ARMHEx or its DIFT memory, reads/writes to the FPGA or the DIFT memory section must not be allowed through non-secure channels. ARMHEx uses ARM TrustZone [63] to prevent from unauthorized accesses (writes or reads) to the trusted area. Any attacks originating from the secure world are not taken into account.

Table 4.1 sums up the type of attacks protected in an ARMHEx-based system. In addition to using the DIFT coprocessor to detect software attacks, the DIFT coprocessor itself is protected, unlike existing works. No side-channel attacks are considered in this thesis.

Table 4.1 – Threat model summary.

Attack types	**ARMHEx**	**Related works**
Software attacks (overflows, injections SQL, ...)	✓	✓
Illegal accesses to the DIFT coprocessor	✓	✗
Side-channel attacks	✗	✗

```c
#include <stdio.h>
#include <string.h>

int main(void){
    char buff[15];
    int pass = 0;

    printf("\n Enter the password : \n");
    gets(buff);

    if(strcmp(buff, "Fp72k17")){
        printf ("\n Wrong Password \n");
    }
    else{
        printf ("\n Correct Password \n");
        pass = 1;
    }

    if(pass){
       // Now Give root or admin rights to user
        printf ("\n Root privileges given to the user \n");
    }

    return 0;
}
```

Listing 4.1 – The buffer `buff` can be overflown if the user writes more than a precise number of characters. In this case, the message **Correct Password** would appear even if the user has not typed the correct password specified in the second argument of `strcmp` function `Fp72k17`.

4.2.2 ARMHEx design

ARMHEx (Figure 4.2) is quite different from existing solutions, especially regarding how information needed for DIFT is recovered. This section explains ARMHEx components and the operations done by the ARMHEx coprocessor.

The PFT decoder (1) (Figure 4.2) is a state machine that decodes trace packets received from CS components as detailed in 3.4.1.1. As the trace is sent at 250 MHz by the TPIU, it is decoded at the same frequency to avoid unnecessary storage overhead. Decoded trace is stored in AXI BRAM.

The TRF (*Tag Register File* (2)) is a register file that stores tags for each of the 16 ARM CPU registers and the 32 floating point registers. This IP has an AXI-lite interface in order to communicate with the ARMHEx coprocessor. The Config IP (3) is an AXI slave IP

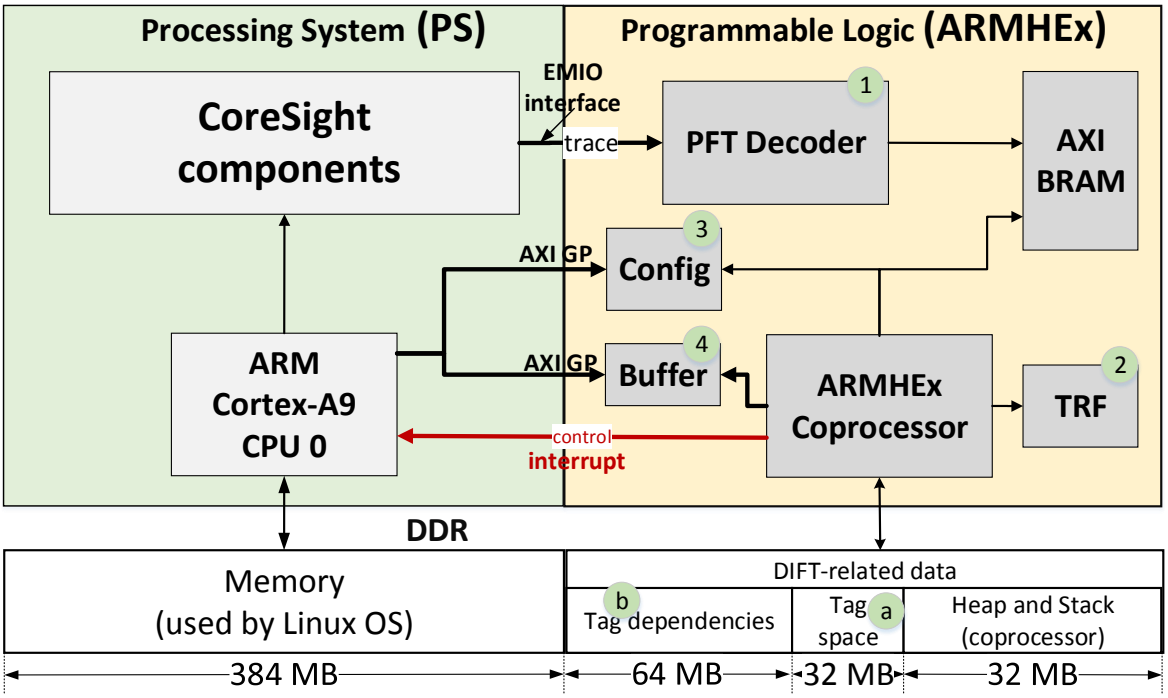


Figure 4.2 – Internal architecture of an ARMHEx-based system.

containing a set of registers that provide a communication channel between the CPU and the ARMHEx coprocessor: it is used to configure tag propagation rules, send the initial value of the `SP` and for debugging purposes. Buffer (4) is a FIFO (AXI slave interface) that contains instrumented memory addresses.

Tags for memory addresses are stored in the tag space memory section (a). Tag dependencies ((b) in Figure 4.2) is also a memory section containing annotations obtained through static analysis (currently implemented with the Capstone disassembler engine [7]). The structure of this memory section is shown in Figure 4.3 (inspired from a memory section layout in [4]). The upper part contains offset/jump addresses used to locate annotations related to basic blocks. The address received in the decoded trace is used to find the jump address where annotations are located for the basic block. Figure 4.4 shows an example to illustrate the content of tag dependencies memory section. If an application has `n` basic blocks, the basic block jump table has `2n` entries. At each basic block jump address, the first entry is the header that contains information such as the number of annotations, information on how to decode the instructions. Then, there are annotations to be executed by the ARMHEx coprocessor.

Figure 4.5 shows the design implemented using Xilinx Vivado. The design shows the `Zynq processing system` IP which represents the ARM core configuration. The other important IP used is the `Microblaze` which represents the ARMHEx coprocessor. There are two clocks used in this design. `FCLK_CLK0` is fixed to 100MHz and `FCLK_CLK1` is fixed to 250 MHz in order to receive trace and decode it.

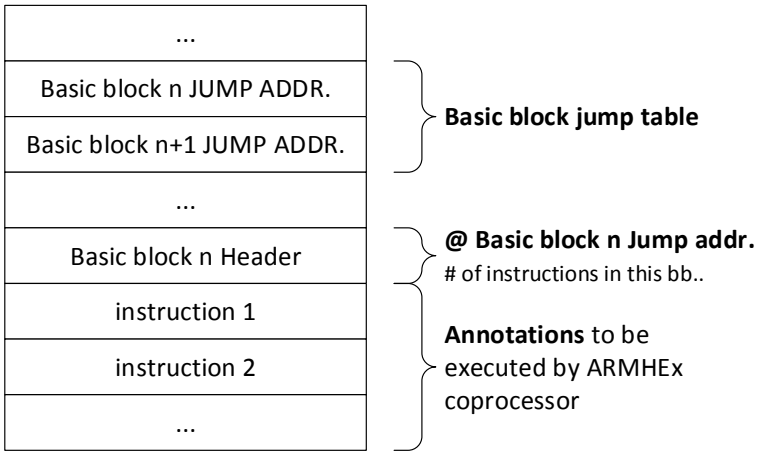


Figure 4.3 – Tag dependencies structure layout.

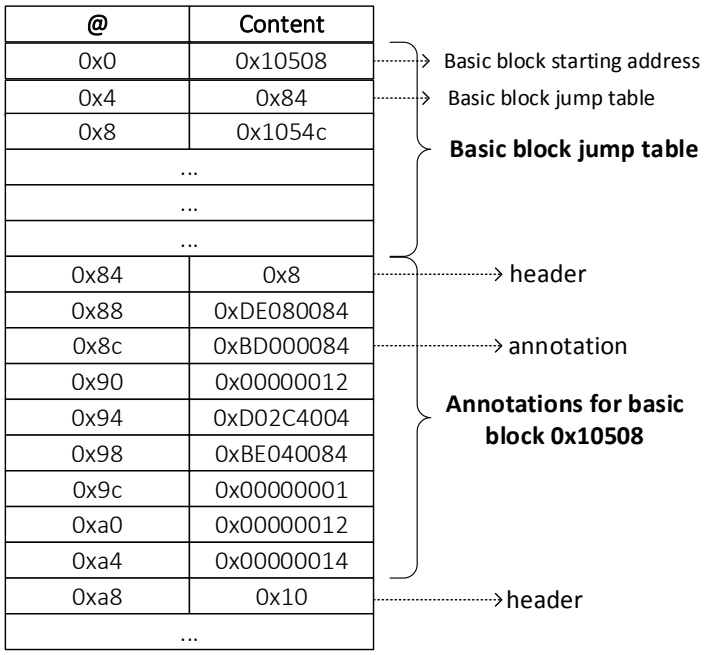


Figure 4.4 – Example of tag dependencies memory section content.

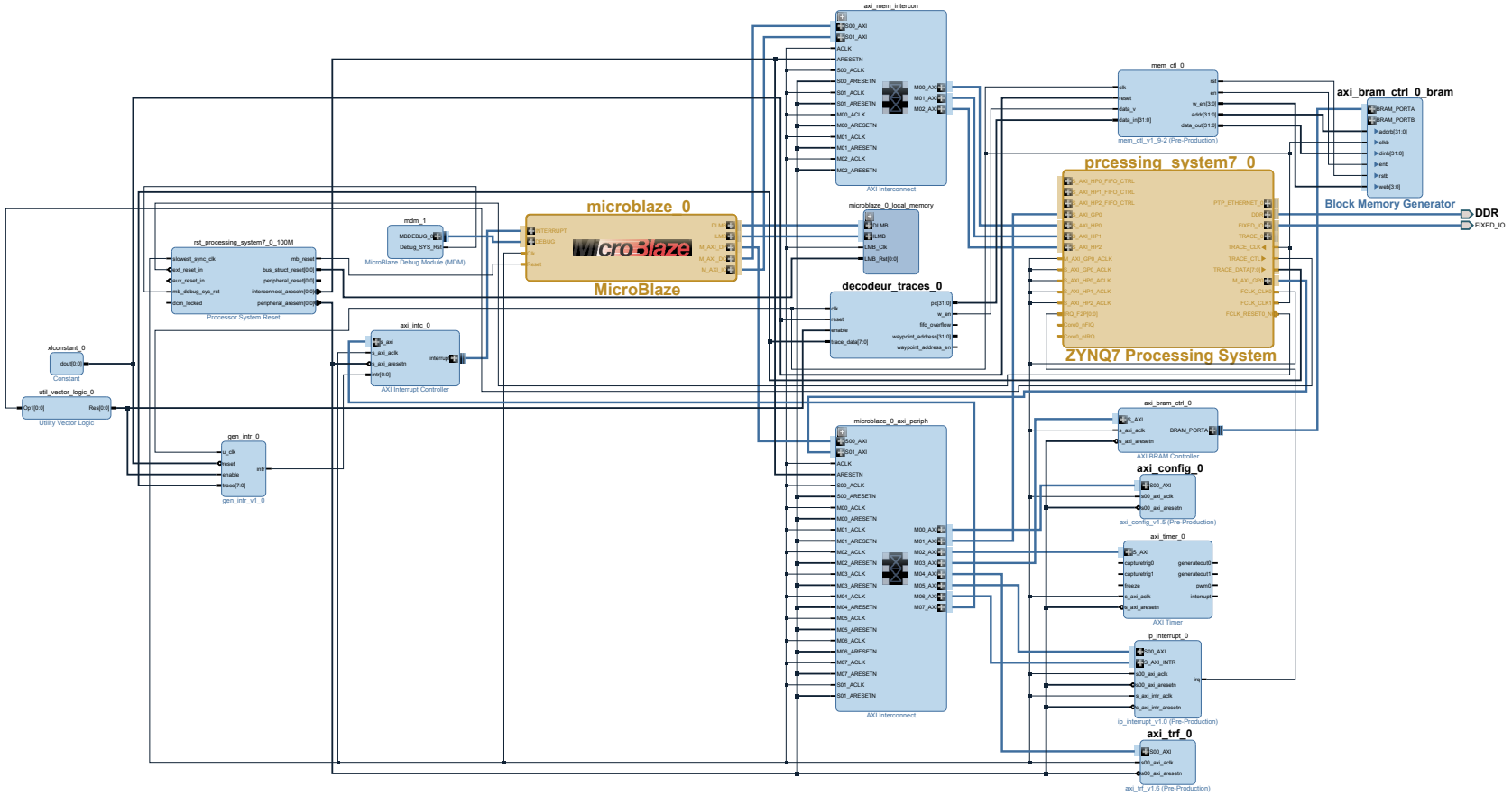


Figure 4.5 – This design shows the implementation of the architecture shown in Figure 4.2. The Processing_System_IP contains the default configuration of the ARM core. The application runs on the ARM core. The Microblaze softcore IP represents the ARMHEx coprocessor. The ARMHEx coprocessor is responsible for managing all the IPs in the PL. The decodeur_traces IP decodes trace coming out from the TPIU via EMIO interface and stores it in the local memory. Other important IPs are marked in bold: axi_trf which contains tags for the ARM registers, axi_config which contains some configuration values and the DDR which is accessible on the PL via the AXI HP (*High_Performance*) Slave ports.

4.2.3 ARMHEx operations

The ARM core runs the application. Before the application is executed, an important amount of tasks are done by the OS without user intervention. In order to take advantage of CS components, they need to be configured as explained in section 3.3.1. The application loader, which is responsible of loading all application sections (`.text`, `data`, ...), also configures CS components, Config IP and loads annotations in tag dependencies memory section before executing the application. Once the application is launched, trace starts to appear on the FPGA part. The trace is decoded and stored in AXI BRAM. On the first storage of decoded trace in BRAM, the ARMHEx coprocessor (MicroBlaze) receives an interrupt to start processing tags. The ARMHEx coprocessor computes tags for each annotation and checks tags to detect a possible attack.

```
while(1){
  // read decoded trace to get basic block starting address
  decodedTrace = Read_mem(DECODED_TRACE_MEM);
  // find offset to fetch annotations for basic block by reading the basic
    block jump table
  offset = Search_annotations(decodedTrace);
  if (offset != 0){
    // get annotation size for current basic block by reading basic block
    header
    annotation_size = Read_mem(TAG_DEPENDENCIES+offset);
    // Get annotations, decode and compute the operation
    for (i = 0; i < annotation_size; i++){
      // Get annotation
      tag_dependencies_instruction = Read_mem(TAG_DEPENDENCIES+offset);
      // Decode and compute the specified operation by the annotation
      Decode_Execute(tag_dependencies_instruction);
    }
  }
}
```

Listing 4.2 – Pseudo-code of the program running on the ARMHEx coprocessor.

The ARMHEx coprocessor, running at 100 MHz, needs to be synchronized with the ARM CPU, running at 667MHz, in order to catch up. The synchronization is done on system calls as in previous works. Listing 4.2 shows the pseudo-code of algorithm running on the ARMHEx coprocessor. Following steps are done in order to perform DIFT operations.

1. Reading decoded trace stored in AXI BRAM.
2. Looking for the corresponding basic block in the tag dependencies memory section by reading the basic block jump table.
3. Reading basic block header, reading the annotation and decoding it.

4. For each annotation, looking for tags of source operands either in memory or TRF.
5. Computing the tag of destination operand depending on current propagation rules stored in Config IP.
6. Updating the corresponding tag in memory or TRF.
7. Checking for security policy violation and if a violation occurs raising an interruption.

Listing 4.3 shows the pseudo-code of `Decode_Execute` function which decodes the annotation and computes the required operation specified by the opcode. The pseudo-code shows only two instructions and how they are decoded. There are other operations such as writing and reading the tag of register or memory addresses that are not shown for the purpose of simplification and readability. These operations are encoded during static analysis according to a defined instruction set. The software running on the ARMHEx coprocessor decodes instructions with respect to the instruction set defined for the output of static analysis.

```
switch (annotation) {
  case DIFT_INIT_REG:
    // bits 31:28
    register_dst = (tag_dependencies_instruction & 0xF0000000) >> 28;
    // bit 27
    tag_v = (tag_dependencies_instruction & 0x08000000) >> 27;
    // Update AXI_TRF
    update_axi_trf(register_dst,tag_v);
    break;
  case DIFT_INIT_MEM:
    // bit 27
    tag_v = (tag_dependencies_instruction & 0x08000000) >> 27;
    update_mem(MEM_ADDRESS, tag_v);
    break;
  }
  case DIFT_UPDATE:
    ...
  case DIFT_CHECK:
    ...
}
```

Listing 4.3 – Pseudo-code of the **Decode_Execute** function.

4.3 PL security and the memory used by the ARMHEx coprocessor

The overall architecture is shown in Figure 4.2. Related works do not take into account the security of hardware DIFT extensions as summed up in Table 4.1. It is important to protect hardware modules and memory sections from being modified through unauthorized

channels. For instance, if a memory section used by the ARMHEx coprocessor is modified by the software running on the Cortex-A9 core, the ARMHEx coprocessor may produce false negatives or false positives. If the example code shown in Listing 4.1 is being protected using ARMHEx, then it is important that any other application does not modify both the memory section used by ARMHEx and the hardware IPs used by ARMHEx.

Listing 4.4 shows a sample attack against DIFT memory section used by ARMHEx. The same attack can be done against any hardware IP used by ARMHEx. The only change that needs to be done is to modify the `dev_base` address by the physical address of the targeted hardware IP. This change seems minor as only a single value is changed in the DIFT memory section. However, it can corrupt the entire algorithm shown in Listing 4.2. The modified value could be one of these four values: basic block starting address received in trace, basic block jump address, basic block header information or annotation. All of these values have an impact on the DIFT process. For instance, if the basic block starting address is modified by the attack then the received trace will not match with this modified address resulting in a failure to process annotations for the basic block. Therefore, some information flows will not be propagated resulting in a non-detection of the attack. Similarly, if any other value in the DIFT memory section is modified by an attacker, the tracked program may produce a false positive (i.e. the attack might be wrongly detected) or false negative (i.e. the attack may be wrongly ignored). It is important to note that existing solutions are not protected against such type of attacks.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <sys/mman.h>
5  #include <linux/types.h>
6
7  int main(void){
8    int memfd;
9    volatile void *mapped_base, *mapped_dev_base;
10   // Physical address of DIFT memroy section
11   off_t dev_base = DIFT_MEMORY_ADDRESS;
12
13   memfd = open("/dev/mem", O_RDWR | O_SYNC);
14   if (memfd == -1) {
15     printf("Can't open /dev/mem.\n");
16     exit(0);
17   }
18
19   // Memory map the physical address to get virtual address
20   mapped_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, memfd,
       dev_base & ~MAP_MASK);
21   if (mapped_base == (void *) -1) {
22     printf("Can't map the memory to user space.\n");
23     exit(0);
24   }
25
26   // get the address of the device in user space which will be an offset from
27   // the base that was mapped as memory is mapped at the start of a page
28   mapped_dev_base = mapped_base + (dev_base & MAP_MASK);
29   // ******************************
30   // ATTACK
31   // ******************************
32   // Modify DIFT memory content
33   *(volatile u32 *) (mapped_dev_base + (10*4))) = 0xABCD1234;
34     return 0;
35 }
```

Listing 4.4 – This Listing shows an example attack against DIFT memory section (Figure 4.1). The Listing maps the memory section using the `mmap` system call. Then, the attack takes place on line 33 by accessing and modifying a memory section that is required by the ARMHEx coprocessor.

4.3.1 ARM TrustZone

ARM TrustZone [63, 64, 65] provides two worlds: the secure world and the normal world. The normal world can only access IP designed for the normal world while the secure world can

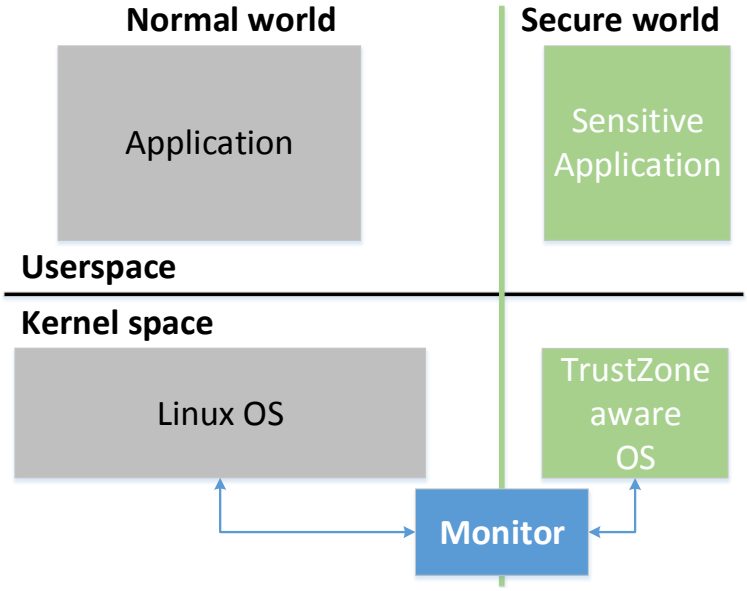


Figure 4.6 – TrustZone hardware protected isolation.

access only secure IPs. This way, the software running in the secure world has a completely different view of the system compared to the software running in the normal world. On the hardware, the separation is made using a dedicated bit called the NS (*Non-Secure*) bit. This distinction between normal and secure world is completely orthogonal to the protection between user-level and kernel-level code as shown in Figure 4.6. Similarly to context switches in software side to switch between userspace and kernel space, there are world switches between normal and secure world. The switches are done using a special `SMC` (*Secure Monitor Call*) instruction with the help of a monitor.

To avoid unauthorized access to ARMHEx, ARM TrustZone is used to ensure that the CPU (untrusted part) cannot access components used by the ARMHEx coprocessor. Figure 4.7 shows that the Linux OS runs in the normal world. In addition, the devices and memory used by Linux OS are also declared as non-secure while ARMHEx and memory used by ARMHEx are declared as secure. When a non-secure element writes to a secure element, the operation will fail on a check of the NS bit. Similarly, if a non-secure element tries to read a secure element, the read will fail. ARM TrustZone is used to isolate ARMHEx with ARM CPU core. Listing 4.5 shows the boot procedure with ARM TrustZone enabled.

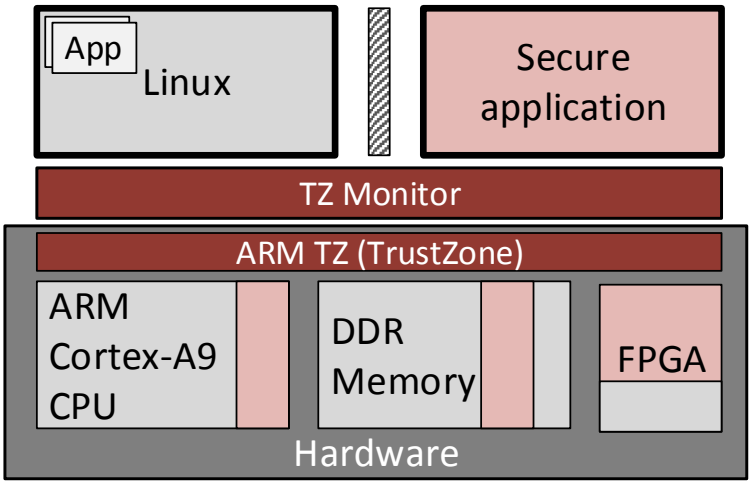


Figure 4.7 – This Figure shows how TrustZone can be used to protect ARMHEx. The ARM core is divided into two worlds: the normal world in which the Linux kernel is running and the secure world in which some secure application is running. The DDR memory is also divided into normal and secure sections: the memory sections used by the ARM core are declared as normal while the memory section used by the ARMHEx are declared as secure. The FPGA area is declared as secure except for some area which can be used as a shared area between normal and secure world.

```
1  # Load compressed kernel image
2  fatload mmc 0 0x8000 zImage;
3  # Load device tree
4  fatload mmc 0 0x1000000 devicetree.dtb;
5  # Load ramdisk file
6  fatload mmc 0 0x800000 ramdisk8M.image.gz;
7  # Load world switch monitor
8  fatload mmc 0 0x1c000000 monitor.bin;
9  # Load TrustZone trusted execution environment
10 fatload mmc 0 0x1c100000 fmp_t_com.bin;
11 # Launch monitor
12 go 0x1c000000;
```

Listing 4.5 – This Listing shows the order of boot operations when TrustZone is used. First, the kernel image is loaded into memory from the SD card. Then, the device tree is loaded. After, the file system is loaded. These first three steps are also required when TrustZone is disabled. The specific actions for enabling the TrustZone feature are shown in lines 8, 10, and 12. The world switch monitor is loaded into memory from the SD card. Then, the TEE (*Trusted Execution Environment* is loaded. After these steps, the monitor can be launched.)

4.3.1.1 Protecting DDR memory

On Zynq SoCs, ARM TrustZone can also be used to protect DDR memory. The only constraint in Zynq SoCs is that the region to be protected should be a multiple of 64MB. As a consequence, the DIFT memory section used by ARMHEx is 64 MB and is protected by unauthorized accesses using ARM TrustZone. The tag space and heap and stack memory sections also have a size of 64 MB.

4.3.1.2 Protecting FPGA

In order to protect the FPGA, the NS-bit is set so that the IPs in the FPGA are all in the secure world. There are two interfaces used to recover information on the FPGA part: AXI and EMIO interface. The AXI interface has special bits: AWPROT and ARPROT that are used to provide either a secure (if the bit is low) or a non-secure (if the bit is high) feature provided by additional bits. The EMIO interface, used to recover trace, has no such security feature provided by an additional bit. However, CS components are configured by the kernel making the configuration process not vulnerable to unauthorized access attack originating from userspace. Furthermore, the kernel is considered secure. Therefore, both interfaces are protected.

4.4 ARMHEx evaluation

Implementations were done with Vivado 2016.4 tools on a Xilinx Zedboard including a Z-7020 SoC (dual-core Cortex-A9 running at 667MHz and an Artix-7 FPGA). The FPGA logic has around 85K logic cells and 560 KB of Block RAMs. The ARMHEx coprocessor is implemented in a Microblaze softcore for this proof-of-concept. The evaluation proves the following points:

- Implement a sample security policy using ARMHEx.
- Evaluation of ARMHEx system: area and power overheads.
- Efficiency of ARMHEx compared to related works.
- Security level provided by ARMHEx.

4.4.1 Example security policy: DLP (*Data Leakage Prevention*)

As an introduction to DIFT implementation within the ARMHEx framework, an example of DLP is shown in Listing 4.6. It opens a file depending on user type (root or normal user), copies the file contents in a buffer and prints it. In order to avoid data leakage, the buffer should not be printed if it contains secret data as it may be the case if the user is root.

```
1  char buffer[20];
2  FILE *fs;
3  // normal user
4  if(geteuid() != 0){
5    fs = fopen("welcome", "r"); //public
6    if(!fs)
7      exit (1);
8  }
9  // root user
10 else{
11   fs = fopen("passwd", "r"); //secret
12   if(!fs)
13     exit(1);
14 }
15 fread(buffer, 1, sizeof(buffer), fs);
16 fclose(fs);
17 printf("Buffer Value: %s \n", buffer);
```

Listing 4.6 – This Listing shows an example code for DLP. After declaring the buffer and the file pointer on lines 1 and 2 respectively, a verification is done based on the user type. If the user is a normal user, then a file declared as public (accessible by everyone i.e. can be printed on a public interface) is opened. Otherwise, if the user is root, then a secret file (not printable on a public interface) is opened. Then, the content of the file is read and printed on the `stdout`. The goal here is to avoid leaking data to a public interface. If the user is a normal user, the data contained inside the buffer comes from a public file and therefore, it can be printed without any issues. However, if the user is root, then the `printf` should not write any value to the `stdout`.

ARMHEx operates on assembly instructions which facilitates the DIFT implementation for all programming languages. The example code shown in Listing 4.6 is compiled for Zynq SoC to obtain the assembly code. System calls (e.g. `write` system call that is called for the `printf` function call) need to be modified in order to send tag related information to the ARMHEx coprocessor. In this chapter, it is considered that static analysis can determine the tag of files being read: `welcome` (tagged public) and `password` (tagged secret).

After the code is statically analyzed, annotations are generated and stored in memory as shown in Table 4.3. The jump table is located from address 0x0 to 0x54. There are some basic blocks that have no annotations. For instance, this is the case for basic block starting at address 0x10528. In this case, there is no need for the ARMHEx coprocessor to compute any tag propagation for this basic block. Therefore, by convention, an offset value of 0x0 is set which tells the ARMHEx coprocessor to jump to the next basic block. From address 0x58 to 0x158, annotations are stored. These are the instructions that the ARMHEx coprocessor decodes and computes tags.

```
1  char buffer[20];
2  FILE *fs;
3  // normal user
4  if(geteuid() != 0){
5    fs = fopen("welcome", "r");
6    if(!fs)
7      exit (1);
8  }
9  // root user
10 else{
11   fs = fopen("passwd", "r");
12   if(!fs)
13     exit(1);
14 }
15 fread(buffer, 1, sizeof(buffer), fs);
16 fclose(fs);
17 printf("Buffer Value: %s \n", buffer);
```

```
tag[Mem[buff]] = 0; //Tag
    initialization
tag[Mem[fs]] = 0;
// normal user
// Decoded trace (if taken)
tag[Mem[fs]] = 0;


// root user
// Decoded trace (else taken)
tag[Mem[fs]] = 1;


tag[Mem[buffer]] = tag[Mem[fs]];

if (tag[Mem[buffer]] == 1)
    interrupt
```

Listing 4.7 – Simplified view of corresponding DIFT operations for example DLP code.

Listing 4.7 shows the corresponding DIFT operations for the example DLP code. The first operation, in order to implement DIFT, is tag initialization. On lines 1 and 2 of Listing 4.6, the tags of `buffer` and `fs` need to be initialized. Two levels of security (private and public) are considered in this work. The OS sends the tag value, `buffer` address and its size to the ARMHEx coprocessor. The allocated space for `buffer` is marked with a tag that can be of different size. The `fs` FILE pointer is initialized with a tag according to the executed branch (`if` or `else`). Information about which branch is executed is obtained using the decoded trace. If the user is not root, lines 5 and 6 are executed. In this case, the tag of `fs` is set to public as `welcome` file is public. Otherwise, lines 11 and 12 are executed. As `passwd` file is considered as secret, the tag of `fs` is set to secret.

The tag check operation happens on line 17 when `buffer` is sent outside to the standard output. The ARMHEx coprocessor sends tag value of `buffer` to the CPU: if it is secret, then a violation has occurred (a secret information is being sent outside the system) and an exception is raised. This example shows how ARMHEx uses the three operations required to implement DIFT described in section 1.1. Tag initialization and tag check operations need OS support while the ARMHEx coprocessor alone is responsible for tag propagation.

By running this application as a normal user, ARMHex does not produce any error because a public file is opened which is authorized by the DLP security policy. However, if the application is run by a root user, an error is thrown to indicate the user that the program violates the security policy. This sample use case shows that ARMHex is fully capable of detecting security policy violations.

Table 4.2 – DIFT memory section containing annotations for the DLP example.

Memory address	Memory content
0	10508
4	58
8	10528
c	00000000
10	1052C
14	80
18	10554
1c	a8
20	10544
24	00000000
28	10548
2c	d4
30	1056C
34	00000000
38	10570
3c	e0
40	10550
44	00000000
48	10578
4c	00000000
50	1057C
54	ec

Table 4.3 – This Table shows the jump table contained in the DIFT memory section. The table is read by the ARMHEx coprocessor using two consecutive reads. The first value shows the basic block addresses while the second value shows the offset at which the annotations are located for the corresponding basic block.

Memory address	Memory content
58	9
5c	BD040084
60	ED080084
64	BD000084
68	00000012
6c	D02C4004
70	00000001
74	00000001
78	00000012
7c	00000000

Table 4.4 – This Table shows the annotations stored at the offset found in the jump table.

Nonetheless, the simplifying assumption that the tags of files can be determined statically is not realistic even though this is considered to be true in most existing works. Therefore, the support of OS is required in order to provide tags of files used by a program at runtime.

4.4.2 Results

4.4.2.1 Area

Area results are shown in Table 4.5. Most of the used FPGA area is filled by the AXI interconnect (5.87%), Config IP (5.20%) and the Microblaze softcore (4.62%). Other IPs occupy less than 1% of the FPGA area in terms of slices. Overall, 10% of the FPGA area is used in order to implement DIFT for one ARM Cortex-A9 CPU. The Zedboard (Zynq Z-7020) includes an Artix-7 FPGA and it could easily fit into the smallest Zynq Z-7010 SoCs (available on Zybo boards).

Table 4.5 – Area results of ARMHEx on Xilinx Zynq Z-7020.

IP Name	**Slice LUTs**	**Slice Registers**	**Slice (in %)**	**BRAM Tile**
Microblaze	1578	1407	614 (4.62)	6
MDM	102	110	40 (0.30)	0
Local memory	14	4	11 (0.08)	32
PFT Decoder	105	211	60 (0.45)	0
AXI TRF	53	105	24 (0.18)	1
Config	914	2141	692 (5.20)	0
AXI Interconnect	1788	2436	781 (5.87)	0
BRAM	2	0	1 (0.01)	2
BRAM Controller	157	168	59 (0.44)	0
Miscellaneous	641	586	171 (1.29)	0
Total Design	5354 (10.06%)	7168 (6.74%)	2453 (18.44%)	41 (29.29%)
Total Available	**53200**	**106400**	**13300**	**140**

4.4.2.2 Power

Table 4.6 shows the power overhead of ARMHEx with default synthesis and implementation strategies on the ZedBoard. The power overhead of ARMHEx consists only of the power overhead of the FPGA part because the ARM core is running the application. Therefore, the power overhead of ARMHEx is the sum of the power consumption of clocks, signals, logic, and BRAM that are located in the FPGA part which equals to 0.13 W.

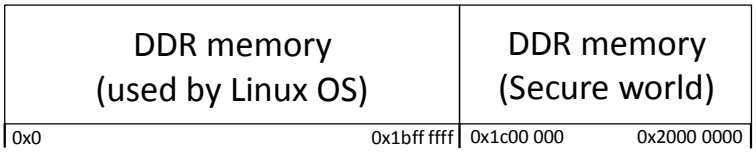


Figure 4.8 – DDR memory protection using ARM TrustZone.

It represents 8.45 % of the baseline power consumption of Zynq processing system (1.538 W). The results obtained are provided by Vivado tools using the post-implementation design.

Table 4.6 – Power consumption results of ARMHEx on Xilinx Zynq Z-7020.

	Power consumption (in Watt)
Clocks	0.039
Signals	0.034
Logic	0.017
BRAM	0.040
PS7	1.538

4.4.3 Security evaluation

In section 4.4.1, it is shown that ARMHEx can detect security policy violations allowing to detect software attacks. To protect against unauthorized access attack discussed in section 4.3, SafeG [66] dual-OS monitor has been used to take profit of ARM TrustZone. Linux runs in the normal world and the TOPPERS/FMP kernel [67] runs in the secure world. In order to evaluate ARMHEx isolation, secure memory region used by the ARMHEx coprocessor is accessed from Linux running in the normal world.

ARM TrustZone is used to protect one secure memory section (from `0x1c00 0000` to `0x1fff ffff`) (Figure 4.8). This secure region is only accessible from the secure world. The kernel that runs in the normal world is not able to access this region and it results in an AXI bus error (DECERR or SLVERR depending on the type of transaction) which is the expected error as a response to a Non-Secure read/write transaction. Listing 4.8 shows the error received when a secure address is accessed with a read and write transaction. It shows that unlike existing approaches, ARMHEx does not allow any unauthorized accesses to the memory section and IPs used by ARMHEx.

```
# read secure address (0x1c00 0000)
zynq> devmem 0x1c000000
Unhandled fault: external abort on non-linefetch (0x18) at 0xb6fe9004
Bus error
# write 0x1234 abcd to secure address (0x1c00 0000)
zynq> devmem 0x1c000000 32 0x1234abcd
Unhandled fault: external abort on non-linefetch (0x18) at 0xb6f04000
Bus error
```

Listing 4.8 – This Listing shows that the Linux kernel, running in the normal world, cannot access a secure memory region. It fails on a bus error which is the expected result as the AXI bus error verifies for each transaction whether the NS-bit matches or not.

4.4.4 Comparison with previous works

Table 4.7 – Performance comparison with related works.

Approaches	**Kannan [2]**	**Deng [3]**	**Heo [4]**	**ARMHEx**
Hardcore portability	No	No	Yes	Yes
Main CPU	Softcore	Softcore	Softcore	Hardcore
Communication overhead	N/A	N/A	60%	5.4%
Area overhead	6.4%	14.8%	14.47%	0.47%
Area (Gate Counts)	N/A	N/A	256177	128496
Power overhead	N/A	6.3%	24%	8.45%
Max frequency	N/A	256 MHz	N/A	250 MHz

Table 4.7 shows a performance comparison of ARMHEx with previous off-core approaches. Unlike previous works, ARMHEx is based on an ARM hardcore processor: it opens interesting perspectives as this work is easily portable to existing embedded systems. Approaches proposed by Heo [4] and Lee [5] are not portable on Zynq SoC due to CS PTM component.

Furthermore, the time cost for communication between a CPU and the coprocessor is 5.4% in this work compared to 60% in [4]. In terms of area, ARMHEx has the best coprocessor/processor ratio: the reason is that the CPU used in this work is a Cortex-A9 which has around 26 million gates [68]. Other works use softcores as their main CPU which have a lower number of gates. If area results (in terms of gate counts) are compared, this approach performs better than existing works. Moreover, regarding the power ratio of a DIFT-enhanced architecture, ARMHEx is better than [4] (8.45 % instead of 24%). However, Deng et al. [3] have a better power overhead than ARMHEx because they implemented a dedicated hardware module for DIFT instead of a coprocessor-based approach as in ARMHEx.

ARMHEx is able to operate at a maximum frequency of 250 MHz (bridled at 100 MHz for the first implementation because of a Microblaze used for DIFT computations).

4.5 Summary

This chapter proposed an implementation, named ARMHEx, of a system taking advantage of the proposed strategy to recover information required for DIFT in Chapter 3. The results presented in this work have been published in [62, 69]. A simplifying assumption was made, similarly to most existing works, that the tags of files can be determined statically. It can be the case for simple examples or use cases. However, on a real-world system, this is not the case and OS support is required. This helped us build a full prototype which is used to verify that the proposed strategy allows recovering all information required for DIFT. The obtained results show that using a small communication runtime overhead due to instrumentation, the DIFT can be implemented on ARM hardcore CPU. Furthermore, the area ratio between the coprocessor and processor is less than 1 % which shows that ARMHEx can be added to existing systems without incurring huge area overhead. ARMHEx propagates and checks tags using the ARMHEx coprocessor (MicroBlaze). As a softcore CPU is used in order to decode and compute tags, the execution of annotations requires multiple cycles. To overcome this issue, a dedicated DIFT coprocessor needs to be developed that decodes and executes annotations.

Chapter 5

Dedicated coprocessor for DIFT

This chapter presents the architecture of the dedicated DIFT coprocessor developed in order to provide missing features in related works: specification of the security policy (compile-time or runtime), support for FP operations and multithreaded applications.

Contents

5.1 Introduction

The information required for DIFT can be recovered and used to propagate and check tags in order to make sure that the program running on the ARM core does not violate the security policy. However, in the previous chapter, propagation and checks have been done in software (MicroBlaze used as the DIFT coprocessor) requiring multiple clock cycles in order to compute a single tag operation. Therefore, it is important to decrease tag operations execution time: this is the main goal of this chapter. Furthermore, a new design for the DIFT coprocessor is required in order to improve features missing in related works. Existing works target a softcore CPU (often LEON3) which results in design considerations that are not suitable for hardcore processors. For instance, the DIFT coprocessor proposed in existing works such as [2] cannot be used to compute tags for floating point code. Section 5.2 describes the DIFT coprocessor requirements considered in this chapter. Then, the system design is explained in section 5.3 in order to understand mandatory operations and software requirements that

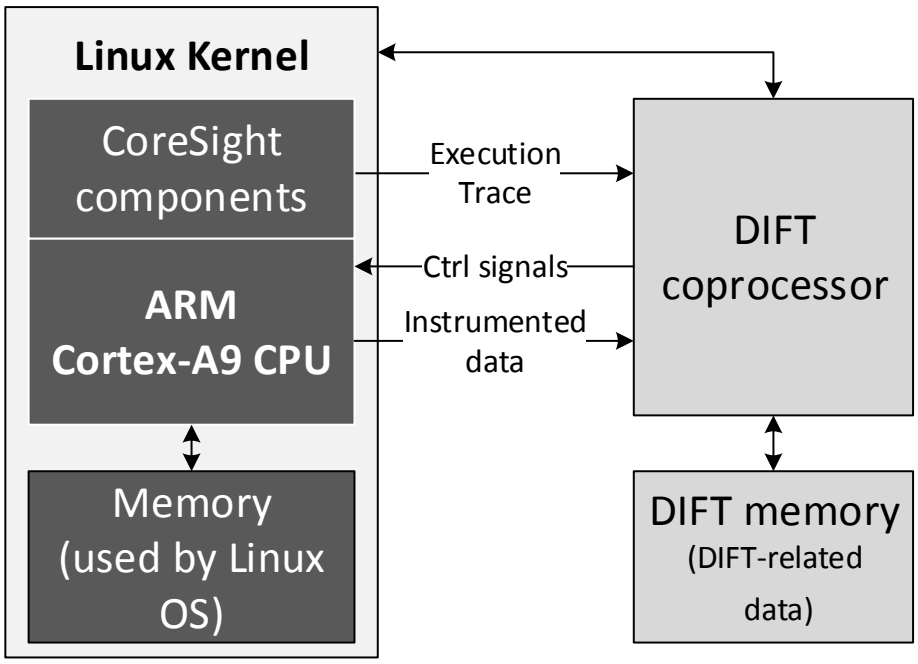


Figure 5.1 – DIFT coprocessor design requirements.

are required by the system. Thereafter, the evaluation is proposed in section 5.5 in order to compare the architecture with existing works and to discuss case studies where the design proposed in this work outperforms existing works.

5.2 Requirements for the DIFT coprocessor

5.2.1 ARM architecture related

Figure 5.1 presents the overall design of the architecture which looks similar to the one proposed in chapter 4. The main core, an ARM Cortex-A9 core, executes the application and sends information to the DIFT coprocessor. The coprocessor is responsible for reconstructing the control flow graph of the application, finding annotations for basic blocks and computing tag operations. Furthermore, an additional information is being sent from the kernel in order to determine the tag of files used by the kernel. It results in another information flow that the DIFT coprocessor should manage.

The ARM Cortex-A9 includes an FPU (*Floating Point Unit*) which is responsible for computing floating point operations. The DIFT coprocessor requires another register file to store tags for floating point registers used by the FPU. Therefore, it also requires specific instructions to compute operations on such registers. In addition, general instructions are required to transfer data between general-purpose and floating point register files.

5.2.2 Flexibility

The DIFT coprocessor has been designed in order to provide maximum flexibility in terms of security policy specifications. The security policy specifies how tags are checked and propagated in order to detect a specific attack. Two aspects related to security policies are missing in existing works: flexibility in security policies specification and tag granularities.

5.2.2.1 Security policy

Security policies can be specified at compile-time or at runtime. The compile-time solution [4] consists in hard-coding propagation and checking operations using dedicated opcodes, during the application compilation. For instance, let us consider the security policy that states that all arithmetic and logic instructions on the ARM core result in the logical `OR` operation on their corresponding tags. Then, during static analysis, for all arithmetic and logical instructions of the program code, an `OR` operation is hard-coded to compute tags of their operands. The runtime solution [1] requires a special register called TPR (*Tag Propagation Register*) to specify the operation that must be done on tag values. In this case, the static analysis gives the operands for each instruction and the class of ARM instruction: Arithmetic/logical, Load/Store, Branch, Floating point Load/Store. Knowing the instruction class, the DIFT coprocessor can determine, with the help of the TPR register, the operation to be done to propagate tags corresponding to the instruction executed on the main CPU. The main advantage of runtime solution is that the TPR value can be modified at runtime in order to modify the security policy without recompiling the application.

No existing works provide the flexibility to specify security policies using both methods. This work proposes an architecture that can implement either one of these approaches providing developers more flexibility to implement security policies. The DIFT coprocessor ISA (*Instruction-Set Architecture*) has two different types of instructions: specific instructions for the compile-time method and specific instructions for the runtime method. The last ones are used in combination with TPR and TCR (*Tag Check Register*) [1] to implement a runtime security policy.

5.2.2.2 Tag granularity

Existing off-core approaches provide a fixed tag size of one bit in most cases. Therefore, security policies that require multiple bits for a tag (such as heap overflow detection [15]) cannot be implemented. This work offers hardware support for a tag size up to 32 bits. In addition, it can support multiple security policies as discussed in section 5.5.1.

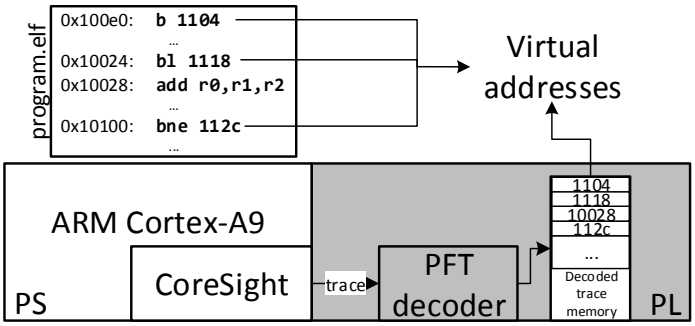


Figure 5.2 – Virtual addresses in the decoded trace.

5.2.3 Tag virtual memory

In hardware-assisted DIFT existing works, tags are associated with physical memory addresses. However, Linux applications are compiled to use virtual addresses (as shown in Figure 5.2) and the MMU is responsible to translate them into physical addresses during execution. Existing solutions consider that it is possible to recover translation information from the MMU. This assumption is only realistic if the main CPU is a softcore tightly coupled with the DIFT co-processor. On a hardcore, a solution could be to modify the Linux kernel in order to send to the coprocessor information about PTEs (*Page Table Entries*), which are managed by the kernel. However, this can be costly because each time a translation is done, the virtual and physical page numbers need to be sent to the coprocessor as well. This information can be difficult to obtain from the kernel and requires lots of minor modifications to the kernel source code which is constantly evolving. Furthermore, the trace generated by the PTM contains virtual addresses (as shown in Figure 5.2). Therefore, in order to limit kernel modifications and to avoid virtual to physical address translation overhead, a tag is associated to a virtual address rather than a physical address.

5.2.4 Tag memory management

Table 5.1 – TMMU entry.

Virtual page number	Physical tag page number
0x12345	0x1c000

A TMMU (*Tag Memory Management Unit*) is required to associate tags to a given memory region used by the main CPU. The TMMU (Figure 5.3) is functionally similar to an MMU used in CPUs. It translates each process virtual address to a physical tag address i.e. the physical address which contains the tag associated to the virtual address. It is implemented as an associative array of 64 entries with some additional logic. Each entry, as shown in

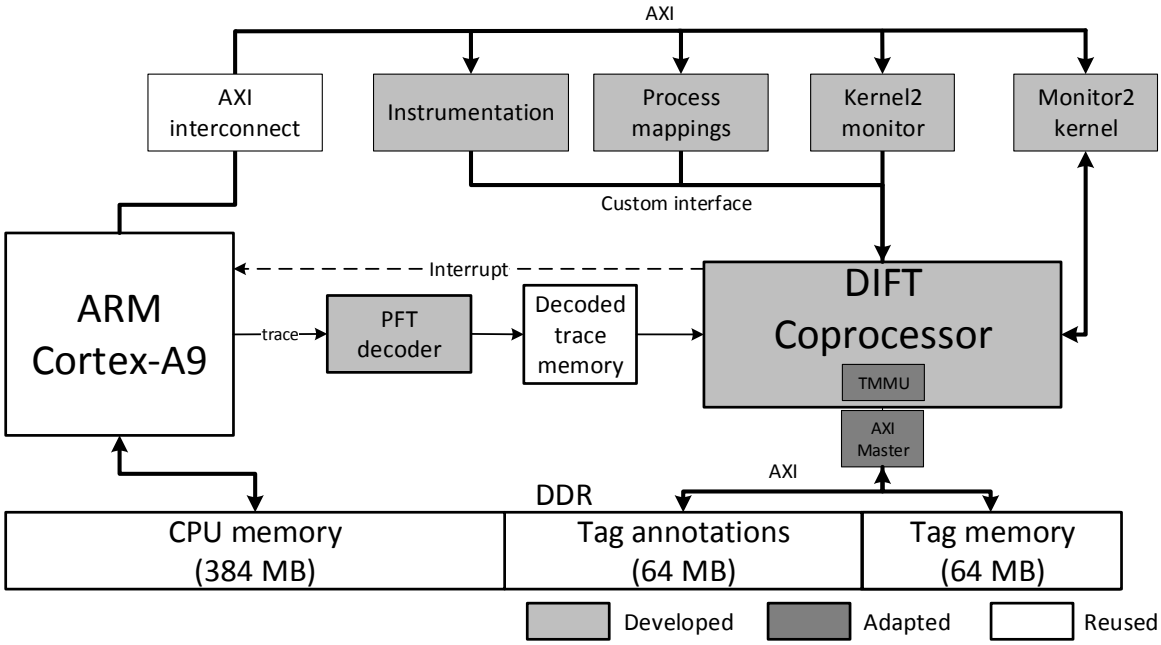


Figure 5.3 – Global architecture with the DIFT coprocessor.

Table 5.1, contains a virtual page number and the corresponding physical page number where the tag is located. In order to initialize the TMMU, mappings of different segments of the process (e.g. code and data) are sent to the FPGA (Process mappings IP in Figure 5.3). This information is retrieved while the application is being loaded by the kernel and is sent to the FPGA part via the process mappings IP (Figure 5.3).

5.3 System design with the DIFT coprocessor

This section explains the overall system design with the DIFT coprocessor and software requirements needed in order to manage DIFT operations.

5.3.1 Overall system design with the DIFT coprocessor

Figure 5.3 shows the global architecture of a system with the DIFT coprocessor. The DIFT coprocessor is responsible for IPs/tags management and program flow reconstruction. There are four new IPs that are used with the custom DIFT coprocessor. The TMMU IP is in charge of finding the physical address where the tag is located for a given virtual address. The AXI Master IP takes care of DDR memory read and write transactions. In chapter 4, this IP was not mandatory as the MicroBlaze processor uses the AXI interface whereas the DIFT coprocessor uses a custom interface and requires an AXI Master IP to communicate with the DDR memory. The other two IPs that were missing in the previous chapter are

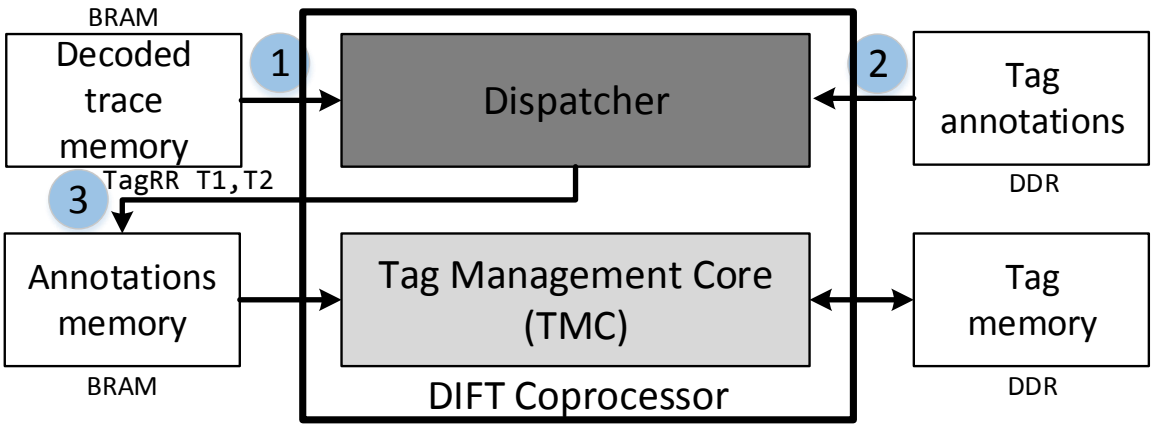


Figure 5.4 – [Architecture of the DIFT coprocessor.]Architecture of the DIFT coprocessor in the black box and its connetions with memory blocks.

kernel2monitor and *monitor2kernel*. The first one is required in order to send the file tag from the kernel to monitor while the second one is used to transmit the file tag from the monitor to the kernel.

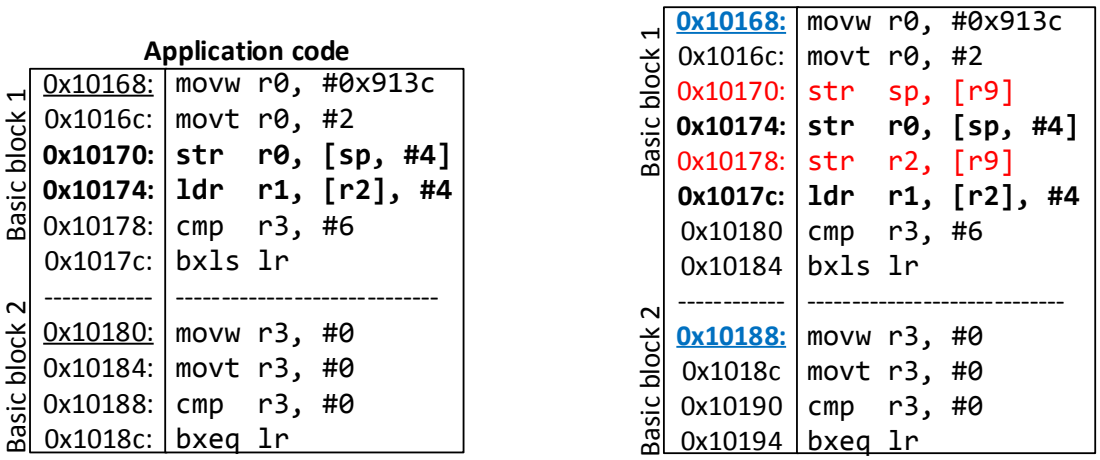
Figure 5.4 illustrates the high-level architecture of the DIFT coprocessor. The DIFT coprocessor has two main functions: manage the sequence of operations and compute annotations. It has been designed in two submodules: the dispatcher responsible for managing the order of operations and the TMC (*Tag Management Core*) unit responsible for decoding annotations i.e. computing and propagating tags.

The dispatcher initializes and manages all IPs. Its main task is to find annotations in program execution order (thanks to the decoded trace) and to store them in the TMC local annotations memory. The dispatcher is implemented as a classic five stages pipelined MIPS CPU that allows executing general operations. It reads the decoded trace (1), finds annotations corresponding to this trace by reading tag annotations memory section in DDR (2) and stores them in the TMC local annotations memory (3).

The TMC core is in charge of propagating and checking tags according to a specified security policy. If a tag check fails, then an interrupt is sent to the ARM core to trigger a counter-measure (e.g. stopping the application).

5.3.2 Software requirements

All existing off-core approaches require software modifications. However, existing works do not provide enough information to easily reproduce their work. As a consequence, following software modifications proposed in this work are explained here in order to make this work reproducible:



(a) Original application code **(b)** Instrumented application code

Figure 5.5 – Binary instrumentation using modified LLVM.

- Instrumentation required in order to retrieve memory addresses.
- Kernel support needed in order to recover information about system calls.
- Process mapping used in order to initialize the tag memory management unit.

5.3.2.1 Instrumentation

The DIFT coprocessor can reconstruct the program execution path by creating a CFG using the decoded trace. However, it has no information on what happens inside each basic block of the CFG. The output of the static analysis, as explained in the chapter 3, states how tags propagation is performed for each CPU instruction. However, for memory instructions, the static analysis cannot compute the memory addresses that are only known at runtime.

This situation is illustrated in Figure 5.5. Figure 5.5a shows the original application code. The starting address of basic blocks 1 and 2 (underlined addresses) are recovered from the decoded trace. For memory instructions (in bold in Figure 5.5a) such as `ldr` and `str` at addresses `0x10170` and `0x10174` of the original application, the value of registers (`sp` and `r2`) cannot be computed from static analysis. This information is needed by the DIFT coprocessor to propagate tags from a register to a memory address (in case of a store operation) or from a memory address to a register (in case of a load operation).

In this work, this information is obtained by instrumenting the original application binary. The instrumented application is shown in Figure 5.5b. Before each memory instruction, another instruction is added (store instructions at addresses `0x10170` and `0x10178` of Figure 5.5b) that sends the missing register value to the memory address contained by `r9`. The `r9` register has been reserved for that and is not used by the application. It contains the virtual address associated with the physical address of the instrumentation IP (shown in Figure 5.3) so that

any store to `r9` results in a write to the instrumentation IP. Compared to the instrumentation as done in chapter 4, the whole code is instrumented rather than only the `.text` section.

5.3.2.2 Kernel support

Existing hardware DIFT approaches only handle tags associated to RAM and registers. However, they do not take into account information stored on mass storage, i.e. tags associated to files. This feature is important to handle inter-process communications and data persistence (after a reboot). Moreover, users are more inclined to specify a security policy at the file level rather than at the memory address or register level. For example, a user can easily identify the files that are supposed to contain confidential information, such as passwords.

Handling file implies some kernel support. However, most of kernel modifications are limited to file Input/Output interface. Only a limited number of system calls such as `read` or `write` have to be modified. In order to handle tags associated to files, RFBlare [70], a modified Linux kernel that implements an OS-level DIFT monitor, is used. This monitor saves tags as file meta-data using file system extended attributes. RFBlare original behavior consists in propagating tags from files to the process memory whenever a file is read and from memory to files whenever a file is written. This OS-level approach is coarse-grained since only one tag is used to abstract the whole memory of a process.

RFBlare is modified to disable the OS-level tag propagation and the feature allowing to link tags to files is used. Some code is added in order to enforce a communication between RFBlare and the DIFT co-processor for each file I/O. For instance, when a `read` system call occurs on the ARM core, RFBlare allows retrieving the tag of the file being read, the address of the buffer where the read data is stored and the number of bytes read. These three values are sent by the kernel to the FPGA part using the kernel2monitor FIFO IP (shown in Figure 5.3). Similarly, if a program writes to a file (e.g. `write` system call), the kernel sends the memory address of the buffer being written and the size of the buffer. Then, the DIFT coprocessor fetches the corresponding tag and sends it back to the kernel. RFBlare uses this tag to set the new tag of the file. All the communication for the `write` system call use the monitor2kernel FIFO IP (shown in Figure 5.3). These FIFOs have an AXI-Lite interface to communicate with the ARM core and a custom FIFO interface to communicate with the TMC.

The main CPU and the DIFT coprocessor must be synchronized since the main core runs faster than the DIFT coprocessor. In this work, this synchronization is done thanks to the monitor2kernel and kernel2monitor FIFO mechanism. As the ARM core and the DIFT coprocessor runs at different frequencies, the attack might be detected after execution of the malicious code. However, it will not compromise the system because an attack needs a

system call (for example `write`) to damage the system. Besides, each system call waits for the DIFT coprocessor to finish tag computation before carrying on execution. Therefore, the synchronization mechanism makes sure that the software attack does not affect the system.

5.3.2.3 Process mappings

The DIFT coprocessor needs to know memory mappings of the program in order to properly initialize the TMMU. The ELF binary loader (`binfmt_elf.c`) is modified in order to send memory mapping of the process. This way, the DIFT coprocessor starts by initializing TMMU before any other operation is done.

5.4 The DIFT coprocessor

This section provides architectural and implementation details of the DIFT coprocessor which is made up of two cores as shown in Figure 5.4: the dispatcher and the TMC core. The DIFT coprocessor is responsible for managing all operations that are listed below.

- The dispatcher manages all IPs and determines the execution order of the program running on the ARM core. The main steps needed to be done for that are:
  - Read trace.
  - Fetch Tag dependencies (store dependencies for the basic block in local memory).
  - Initialize different modules for tag management and trace handling.
- The dispatcher manages inputs and outputs. The main operations are:
  - Read instrumentation buffer.
  - Read kernel2monitor.
  - Write monitor2kernel.
  - Keep track of ARM SP register value.
  - Keep track of ARM PC register value.
  - Configure TPR, TCR registers and TMMU IP.
- The TMC core should manage tags. It consists of the following operations:
  - Configure security policy and tag ALU.
  - Fetch tags from memory.
  - Update register tags.
  - Store tags to memory.
  - Check for security policy violation.

5.4.1 ISA of the dispatcher

The dispatcher is responsible for storing annotations for the TMC core in the same order as the program that is being executed on the ARM core. It also manages other inputs and outputs (e.g. kernel2monitor and monitor2kernel IPs). Therefore, it is implemented as a general purpose MIPS processor providing the advantage of using the existing MIPS compiler in order to compile the software. Furthermore, it allows to provide maximum flexibility in terms of management of IPs. If a change is required for other program analysis techniques such as detection of CRA (*Code Reuse Attacks*), the software running on the dispatcher can be easily adapted. The ISA is detailed by explaining the data types, instruction formats, instruction types, and register files used by the dispatcher. All instructions available are explained in Appendix C.

5.4.1.1 Data type(s)

The dispatcher operates only on the general data type.

5.4.1.2 Instruction formats

There are three types of instruction formats that are used to encode the instruction. The R format instructions are the most common and used by arithmetic and logical operations. The I format instructions are used by memory or branch operations while the J format instructions are used by jump operations.

1. R (*Register*) format

31 26	25 21	20 16	15 11	10 6	5 0
Opcode	Rs	Rt	Rd	shamt	Funct

2. I (*Immediate*) format

31 26	25 21	20 16	15 0
Opcode	Rs	Rt	Immediate

3. J (*Jump*) format

31 26	25 0
Opcode	Target address

Table 5.2 – Overview of instructions executed by the dispatcher.

Instruction type	Opcode type	Format type	Example annotation	Action
General (Arithmetic/ Logical)	add	R	add $1, $2, $3	$1 = $2 + $3
	sub	R	sub $1, $2, $3	$1 = $2 - $3
	addi	I	addi $1, $2, 100	$1 = $2 + 100
	addiu	I	addiu $1, $2, 100	$1 = $2 + 100
	and	R	and $1, $2, $3	$1 = $2 & $3
	or	R	or $1, $2, $3	$1 = $2 | $3
	xor	R	xor $1, $2, $3	$1 = $2 ^ $3
	andi	R	andi $1, $2, 100	$1 = $2 & 100
	ori	I	ori $1, $2, 100	$1 = $2 | 100
	xori	I	xori $1, $2, 100	$1 = $2 ^ 100
	sll	I	sll $1, $2, 10	$1 = $2 << 10
	srl	I	srl $1, $2, 10	$1 = $2 >> 10
	lui	I	lui $1, 100	$1 = 100 x 2^{16}
Load/Store	lw	I	lw $1, 100($2)	$1 = Mem[$2 + 100]
	sw	I	sw $1, 100($2)	Mem[$2+100] = $1
Branch	beq	I	beq $1, $2, 100	if ($1 == $2) go to PC+4+100
	bne	I	bne $1, $2, 100	if ($1 != $2) go to PC+4+100
	blez	I	blez $1, 100	if ($1 <= $2) go to PC+4+100
	bgtz	I	bgtz $1, 100	if ($1 > $2) go to PC+4+100
Jump	j	J	j 10000	go to 10000
	jr	R	jr $31	go to $31
	jal	J	jal 10000	$31 = PC + 4; go to 10000

5.4.1.3 Instruction types

The dispatcher has all non-proprietary instructions of the classic MIPS architecture. Table 5.2 shows important instructions available in the dispatcher. There are four types of instructions available: General, Load/Store, Branch, and Jump. Instructions of general type are responsible for computing arithmetic and logical operations. Load/Store instructions deal with memory load and store operations. Branch instructions allow to execute conditional code and jump instructions are used to jump to a specific location in the code.

5.4.1.4 Register files

The dispatcher has one register file that contains 32 registers with the first register always containing zero value.

5.4.2 ISA of the TMC unit

The TMC core decodes annotations and executes the operation specified in the annotation. It also takes care of storing tags in memory by using the TMMU and AXI Master IPs. As the

dispatcher is pipelined with five stages, the TMC is also pipelined with five stages providing higher throughput. All annotations available are explained in Appendix C.

5.4.2.1 Data types

The TMC core operates on three different data types: tags of general purpose ARM registers, tags of FPU and general data. Tags of general purpose ARM registers type is used to store the tag value of general purpose ARM registers. Tags of FPU data type is used in order to store the tag value of the FPU registers. The general data type is required in order to compute general operations.

5.4.2.2 Instruction formats

There are three types of instruction formats that are used to encode the annotation.

1. Init format used to initialize tags of registers and tags of memory addresses.

31 26	25 21	20 16	15 0
Opcode	Reg_dst	Reg_src	Immediate

2. TR (*Tag Register*) format used to update tags of registers.

31 26	25 22	21 17	16 12	11 7	6 5	4 0
Opcode	ARM_OPCODE_TYPE	Reg_dst	Reg_src1	Reg_src2	n.u	Funct

3. TI (*Tag Immediate*) format used to update tags of memory addresses.

31 26	25 22	21 17	16 12	11 0
Opcode	ARM_OPCODE_TYPE	Reg_dst	Reg_src1	Immediate

5.4.2.3 Instruction types

There are four instruction types as shown in Table 5.3: Tag initialization, Tag ALU, Tag Load/Store, and Compound annotations. The same set of annotations is also included for the floating-point code. Tag initialization annotations can be used in both runtime and compile-time methods to initialize tags of registers or memory addresses. Tag ALU annotations propagate tags for registers. For instance, `TagRRR` annotation (on the fourth row of Table 5.3) shows a runtime annotation that contains `type` field and operands `T1`, `T2`, and `T3`. The operation `op` for this annotation will be determined, at runtime, by reading the TPR register value for the corresponding `type`. If the `type` field is arithmetic and the TPR register states that an `AND` operation must be done on source operands to compute the destination tag, then the operation done on the TMC unit of the DIFT coprocessor is `T1 = T2 AND T3`. The compile-time solution will use the fifth annotation (`TagRRR2`) in order

to specify the operation that will be hard-coded in the annotation itself using the function (`Funct`) field. If the security policy is changed, the compile-time solution must recompile the application while the runtime solution will simply require changing the TPR register value.

Tag Load/Store annotations are used in order to fetch tags from the memory or to store tags to the memory. They are similar to classical load/store operations: the value fetched/stored from/to memory is the tag. The last type of annotation is the compound type which, as the name suggests, consists of doing multiple operations using a single annotation. For instance, if the first annotation of compound type is considered (`TagITR`), it consists of loading value from instrumentation IP, then using the TMMU unit to find the physical tag address i.e. address where the tag is located for the given virtual address.

Then, the tag store operation can happen: it stores the value of register `T3` into the physical address obtained from the TMMU. Therefore, compound annotations allow executing multiple tasks from a single annotation making the output of static analysis more compact in memory.

Table 5.3 – Overview of annotations executed by the TMC.

Instruction type	Opcode	Format type	Operation type	Example annotation	Action
Tag initialization	`TagRImm`	Init	irrelevant	`TagRImm T1,#1000`	`T1` = 1000
	`TagRR`	Init	irrelevant	`TagRR T2,T1`	`T2` = `T1`
	`TagMR`	Init	irrelevant	`TagMR R1,T1`	Mem[`R1`] = `T1`
Tag ALU	`TagRRR`	TR	runtime	`TagRR type T1,T2,T3`	`T1` = `T2` op `T3`
	`TagRRR2`	TR	compile-time	`TagRRR2 AND T1,T2,T3`	`T1` = `T2` AND `T3`
Tag Load/Store	`TagMTR`	TI	runtime	`TagMTR type R1,T1,#4`	Mem [`R1`+4] = `T1`
	`TagTRM`	TI	runtime	`TagTRM type T1,R1,#4`	`T1 = Mem[R1+4]`
	`TagMTR2`	TI	compile-time	`TagMTR2 R1,T1,#4`	Mem[`R1`+4] = `T1`
	`TagTRM2`	TI	compile-time	`TagTRM2 T1,R1,#4`	`T1` = Mem[`R1`+4]
Compound	`TagITR`	TI	runtime	`TagITR T3,T1,#4`	Mem[TMMU(Instrumentation)] = `T3`
	`TagTRI`	TI	runtime	`TagTRI T4,T2,#4`	`T4` = Mem[TMMU(Instrumentation)]
	`TagITR2`	TI	compile-time	`TagITR2 T12,#4`	Mem[TMMU(Instrumentation + 4)] = `T1`
	`TagTRI2`	TI	compile-time	`TagTRI2 T2,#4`	`T2` = Mem[TMMU(Instrumentation + 4)]
	`TagKTR`	TI	compile-time	`TagKTR T1`	Mem[monitor2kernel] = `T1`
	`TagTRK`	TI	compile-time	`TagTRK T2`	`T2` = TMMU(Mem[kernel2monitor])

5.4.2.4 Register files

The TMC unit has three data types. Therefore, it contains three register files (one for each data type): TRF (*Tag Register File*), TRF_FP ((*Tag Register File for Floating-Point numbers*)), and GRF (*General Register File*) (Figure 5.6). TRF contains tags corresponding to the ARM registers (`r0` to `r15`) while TRF_FP contains tags for the ARM floating point registers (`s0` to `s31`). GRF contains 16 general purpose registers.

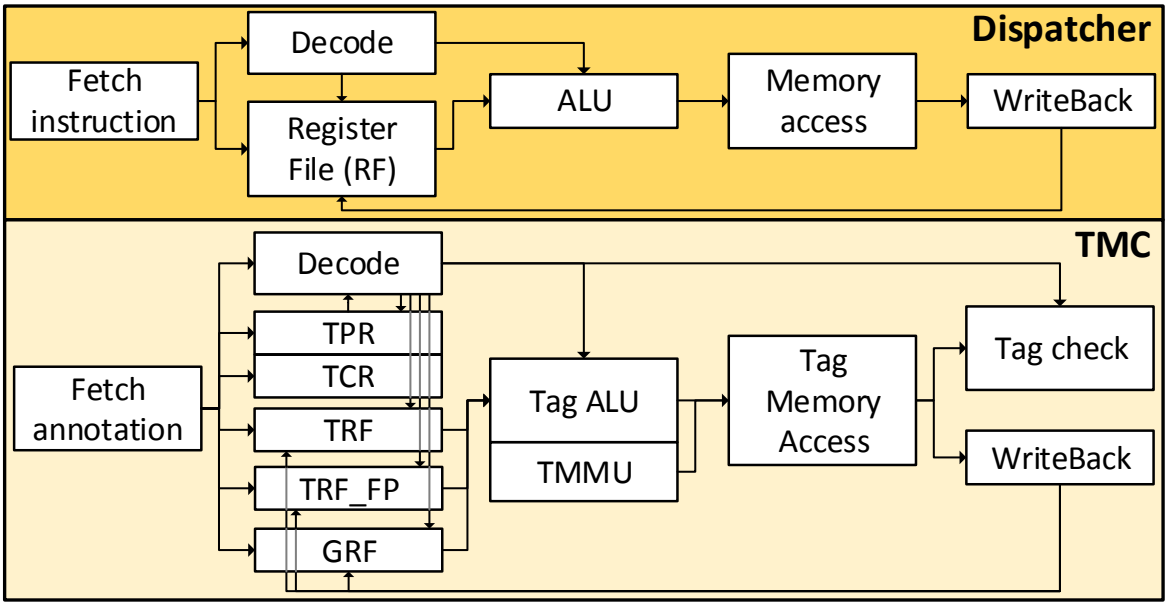


Figure 5.6 – Microarchitecture of the DIFT coprocessor with 5 pipeline stages.

5.4.3 Microarchitecture of the DIFT coprocessor

Figure 5.6 shows the internal architecture of the DIFT coprocessor. The DIFT coprocessor is made up of two units: the dispatcher and the TMC core. The Dispatcher is implemented as a five-stage fully pipelined MIPS CPU (with no branch-prediction unit) while the TMC core is implemented as a five-stage pipelined custom coprocessor to decode annotations.

An important implementation detail about the TMC unit concerns the number of register files. Figure 5.6 shows that there are three register files: TRF (16 registers), TRF_FP (32 registers) and a GRF (16 registers). It means that 13 bits (4 bits for TRF, 5 bits for TRF_FP and 4 bits for GRF) are required in order to address each of the register files. However, there are not enough bits available to store this information. Therefore, TRF and GRF are implemented together as a one big register file. Register 0 to 15 are used for TRF while registers 16 to 31 are reserved for GRF. It allows to use 10 bits (5 bits for TRF_FP and 5 bits for both TRF and GRF) rather than 13 bits required if there were three separate register files implemented.

Pipeline There are five stages of pipeline for each unit as shown in Figure 5.6: instruction fetch, instruction decode, execution, memory access and write-back stage. The MIPS processor is fully pipelined with some additional delays added either by the hardware or the software depending on the instruction. The TMC core is pipelined for Init and TR formats of instruction. All memory accesses on the TMC core require to stall the pipeline because an external DDR memory is used rather than the local BRAM as in case of the dispatcher.

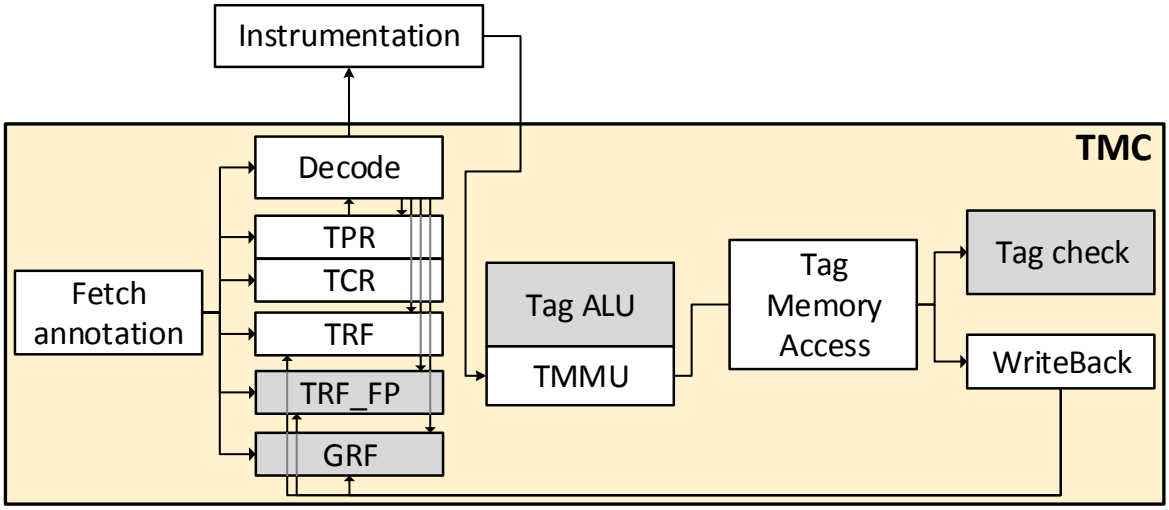


Figure 5.7 – Compound annotation (`TagITR`). example on the TMC core

The TMC core has a very simple instruction fetch unit compared to the dispatcher because it does not deal with branches or jumps. The dispatcher takes care of sending annotations in the right execution order. During the decode stage, the annotation is decoded. If the annotation is of compile-time operation type, then all the information required to compute tag is stored in the annotation itself. The decoder decodes the operands and sets the operation type. If the annotation is of runtime operation type, then the operation is not specified in the annotation. Therefore, the `ARM_OPCODE_TYPE` field in the annotation is used along with TPR and TCR registers to determine the type of operation that the tag ALU must compute. The execute stage of the TMC core performs the operation decoded and takes also care of forwarding a value if any data hazard is detected. In addition, for memory operations, the execute stage also enables the TMMU unit in order to look for the physical address where the tag must be loaded from or stored to. The memory access stage enables the AXI Master IP to enable the transaction from memory. The write-back stage performs one or two operations depending on the annotation type. If the annotation type is compile-time, it only writes back the value. If the annotation type is runtime, then it also verifies the tag value that is being written using the `tag check` module. If a violation is detected, the processor sends an interrupt signal to the ARM core.

Compound annotation Figure 5.7 shows the data path taken by the compound annotation *TagITR* on the TMC core. Blocks that are not used in the TMC core are grayed out. The decode stage enables a signal that allows reading the instrumentation IP which contains the virtual address obtained through instrumentation. During the execute stage, the instrumentation IP delivers this virtual address which is used as input to the TMMU in order to find the physical address where tags are stored for the virtual address. Then, the

memory access stage can access the physical address in order to store the value to the TRF register file. For instance, if the instrumentation IP delivers the address `0x12345678` to the TMMU, the TMMU will look in its associative table the corresponding entry for virtual page number `0x12345`. The TMMU output gives the physical address where the tag is located for the virtual address `0x12345678`. Then, this physical address is used to access the memory using the AXI Master IP and writing back the value read into the TRF register file. Another annotation with a specific opcode is used in order to store the value to the `TRF_FP` floating point register file.

Dispatcher implementation tests Figure 5.8 shows the design developed with Vivado tools for Zynq SoC in order to test the dispatcher core. The design shows the Zynq processing system IP that contains the configuration for the ARM core. The dispatcher (mips_cpu_v2_3 IP) represents the dispatcher. The instruction_mem memory block contains instructions generated using GCC MIPS compiler [71]. The data_mem memory block contains data for the dispatcher. The trace_mem block contains the trace while anotations_bb block contains annotations generated by the static analysis. The AXI GPIO IP is used in order to reset the dispatcher. This design is used in order to make sure that the dispatcher operates correctly. The Zynq system is responsible for writing all previously detailed memory blocks. The dispatcher determines the execution order on the ARM CPU from the decoded trace stored in trace_mem and finds corresponding annotations. These annotations are stored in the annotations_bb block which is used as the instruction memory of the TMC unit.

TMC implementation tests Figure 5.9 shows the design developed with Vivado tools for the Zynq SoC in order to test the TMC core. The design shows the Zynq processing system IP and the TMC core IP. The instruction memory of the TMC core (instruction_mem in Figure 5.9) is filled by the dispatcher core and corresponds to the annotations_bb memory block in Figure 5.8. There is one simplifying assumption in order to easily test the TMC core: the interface used for the IPs in this design does not match the interface in the real design. It is replaced by the AXI BRAM interface in order to use BRAM memory blocks (configured as dual port BRAM). All memory blocks are filled by the Zynq processing system except for instruction_mem block which contains data filled statically during initialization of the FPGA. The TMC core reads annotation using instruction_mem and computes the operation specified in the opcode.

DIFT coprocessor implementation Figure 5.10 shows the complete design developed with the DIFT coprocessor (dispatcher and TMC). The dispatcher IP is actually a sub-design that contains all IPs with the correct interface shown in Figure 5.8 while the TMC core here

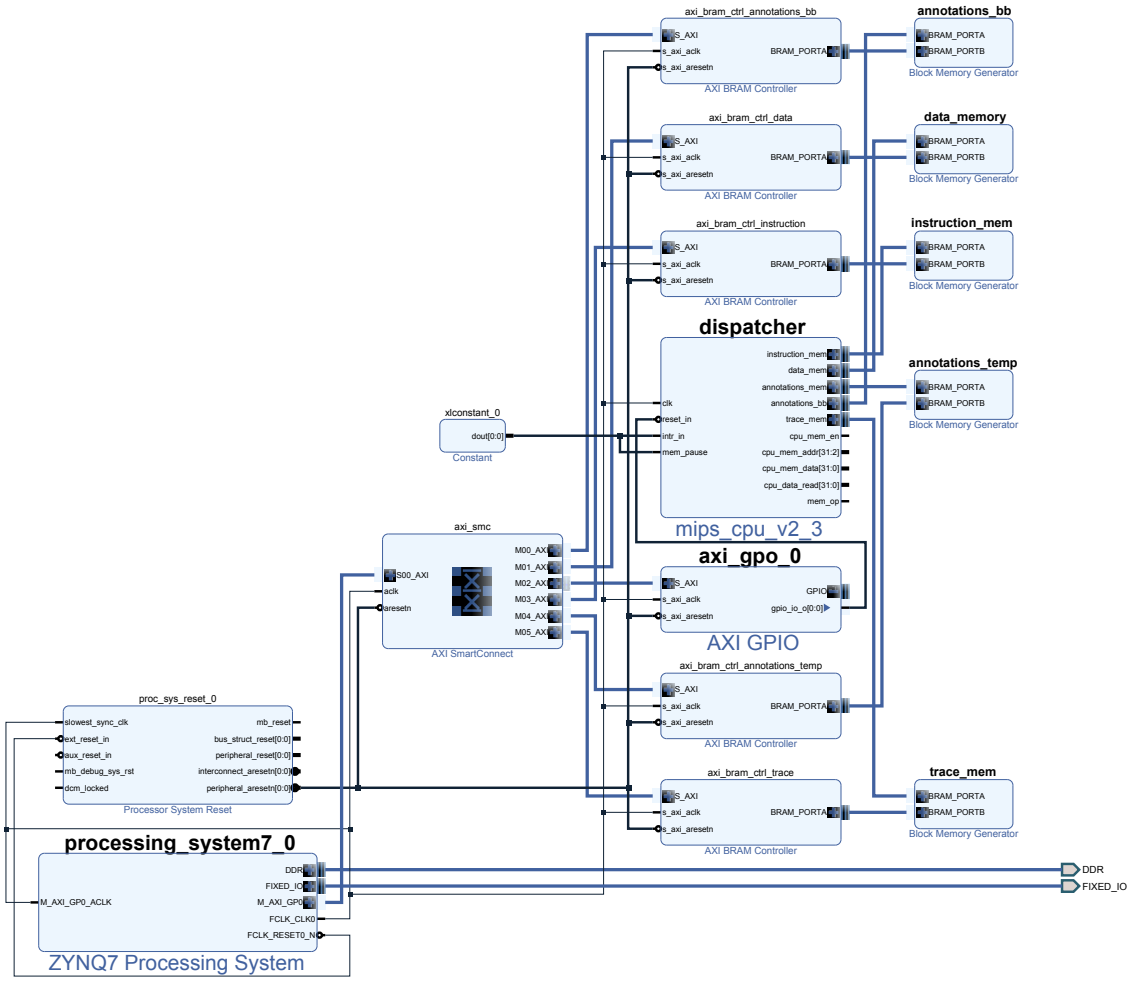


Figure 5.8 – This Figure shows the design made in order to implement the dispatcher core on the FPGA and evaluate it on the FPGA. The Zynq **Processing_system7_0** IP is used in order to initialize the content of instruction memory (**instruction_mem**) and set the appropriate value for the reset signal using **axi_gpio_0** IP. Once the ARM core sets the value '0' to the reset signal, then the dispatcher starts executing instructions specified in the instruction memory. The code running on the dispatcher is shown in Listing 5.1. Once the program has finished execution, the Zynq reads back the content of the **annotations_bb** memory block in order to make sure that the dispatcher has properly dispatched the annotations in the right execution order. This design allows verifying that the dispatcher properly fulfills its role.

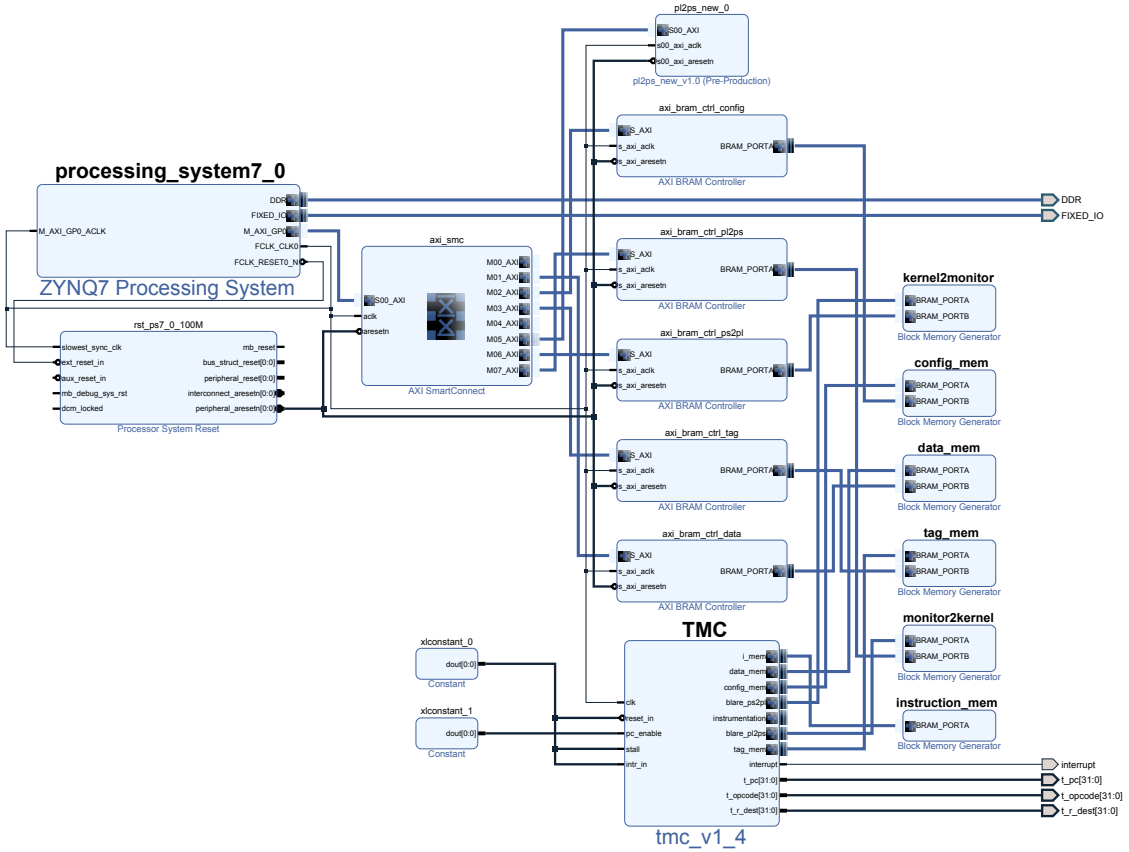


Figure 5.9 – TMC design with Vivado tools on Zynq SoC.

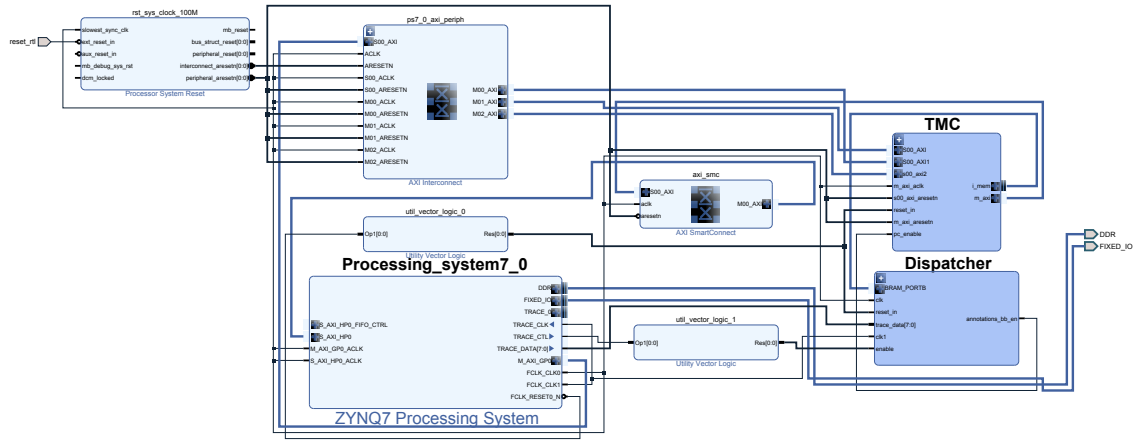


Figure 5.10 – System design with the DIFT coprocessor.

is also a sub-design with all the blocks shown in Figure 5.9 with the correct interface as explained in chapter 3. This design is implemented and evaluated in the next sections.

5.4.4 Software running on the DIFT coprocessor

The dispatcher core runs the program shown in Listing 5.1 which looks quite similar to the pseudo-code shown in Listing 4.2 except for the last operations. In Listing 4.2, the last operation called another software function to decode the annotation. Here, the last operation stores the annotation in the instruction memory of the TMC core in order to decode the annotation in hardware. Furthermore, there is another operation done by the `Check_annotation` function which is required for the management of ARM system calls: it allows to initialize memory addresses with the help of the kernel.

The `Check_annotation` function (Listing 5.2) takes care of managing memory addresses sent or received by the kernel. Depending on the annotation opcode, it will generate annotations. If the opcode is 0x11, then the dispatcher reads kernel2monitor IP and generates annotations to initialize memory addresses specified by the kernel. If the opcode is 0x12, then the dispatcher reads monitor2kernel IP and looks for the tag of the memory address specified by the kernel. When the tag is available from the DDR memory, it is written back to the monitor2kernel IP by the TMC core. This allows to set the tag of the file being written by the kernel.

```
while(1){
  // read decoded trace to get basic block starting address
  decodedTrace = Read_mem(DECODED_TRACE_MEM);
  // find offset to fetch annotations for basic block by reading the basic
    block jump table
  offset = Search_annotations(decodedTrace);
  if (offset != 0){
    // get annotation size for current basic block by reading basic block
    header
    annotation_size = Read_mem(TAG_DEPENDENCIES+offset);
    // Get annotations, decode and compute the operation
    for (i = 0; i < annotation_size; i++){
      // Get annotation
      annotation = Read_mem(TAG_DEPENDENCIES+offset);
      // ARM system call management
      Check_annotation(annotation);
      // Store annotation to the TMC instruction memory
      Store_annotation(annotation);
    }
  }
}
```

Listing 5.1 – This Listing shows the pseudo-code of the program running on the dispatcher core. The dispatcher reads the decoded trace and then, using the address read, determines the offset where to find corresponding annotations for the corresponding basic block. The annotations are copied from the memory to the instruction memory of the TMC core in order to be executed.

```
Check_annotation(annotation){
  // Check annotation for specific opcodes
  If (annotation && 0x11){
    // Read kernel2monitor IP
    buffer_address = Read_fifo(kernel2monitor);
    buffer_size = Read_fifo(kernel2monitor);
    tag = Read_fifo(kernel2monitor);
    // initialize tags of memory addresses
    for (i = 0; i < buffer_size; i++){
      generate_mtr2_annotation(buffer_address, i*4);
    }
  }
  else if (annotation && 0x12){
    // Read monitor2kernel IP
    buffer_address = Read_fifo(monitor2kernel);
    buffer_size = Read_fifo(monitor2kernel);
    // initialize tags of memory addresses
    for (i = 0; i < buffer_size; i++){
      generate_trm2_annotation(buffer_address, i*4);
    }
  }
}
```

Listing 5.2 – The Check_annotation function is responsible for managing memory addresses sent or received by the kernel. If the kernel sends a value using the Kernel2monitor IP, the dispatcher creates a single (or multiple) instruction(s) for the TMC in order to initialize the address with a specified tag value. Otherwise, the kernel sends a value using the Monitor2kernel IP in which case, the dispatcher creates a single (or multiple) instruction(s) for the TMC to fetch tag and store to the Monitor2kernel IP.

This code is compiled using the GCC MIPS compiler [71] and then stored in the instruction memory of the dispatcher.

5.5 Evaluation of the proposed coprocessor

Xilinx tools 2017.1 are used on a Xilinx Zedboard with a Z-7020 SoC (dual-core Cortex-A9 running at 667 MHz and an Artix-7 FPGA) to implement the architecture shown in Figure 5.3. The Clang compiler has been used with a customized LLVM pass to get the binary and annotations (instructions for TMC). The evaluation described in this section has the following goals.

- Study use cases where the proposed architecture performs better than existing architecture.
- Evaluate the feasibility of the proposed architecture.

- Evaluate the cost of software modifications (execution time overhead and memory footprint).
- Compare efficiency with related works.
- Evaluate the area and power overheads of the DIFT coprocessor.

5.5.1 Case studies

This section shows two use cases where the proposed architecture can outperform existing works: handling multiple security policies and multi-threaded applications.

5.5.1.1 Multiple security policies

Existing works propose to manage multiple security policies using the DIFT coprocessor in two ways. On the one hand, most off-core hardware assisted works [1], [2], [3], [4] target security policies that require only a single bit in order to determine the software attack. However, in practice, there are security policies that require more than a single bit in order to determine the attack. For instance, this is the case for the security policy that allows the detection of heap overflow attack. On the other hand, other works [40] modify the CPU architecture by increasing the CPU bus size in order to run multiple security policies. Instead of designing a 64-bit CPU, authors proposed to use a 128-bit CPU where 64-bits are used for data and the other 64-bits manage tags. This approach is not feasible on hardcore CPU as considered in this work because their architecture cannot be modified.

Therefore, it is important to be able to independently run multiple security policies on the DIFT coprocessor. The architecture has been designed in order to provide the ability to program multiple security policies using independent TMC cores. Figure 5.11 shows the design in order to run multiple security policies. There are two TMC cores: one per security policy. The dispatcher is responsible for writing annotations to both annotations memory of each TMC. The same annotation is written into both TMC cores. However, each TMC core is configured at runtime with a different security policy by programming TPR and TCR registers. When the annotation is decoded by the TMC, the decoder also determines the operation to be done for each security policy. For instance, the security policy one states that for a memory store operation, the tag to propagate is the tag of the register containing the memory address while the security policy two states that for a memory store operation, the tag to propagate is the tag of the memory address. Then, the decode stage of TMC units will set accordingly all the required signals in order to run these security policies separately on each TMC unit.

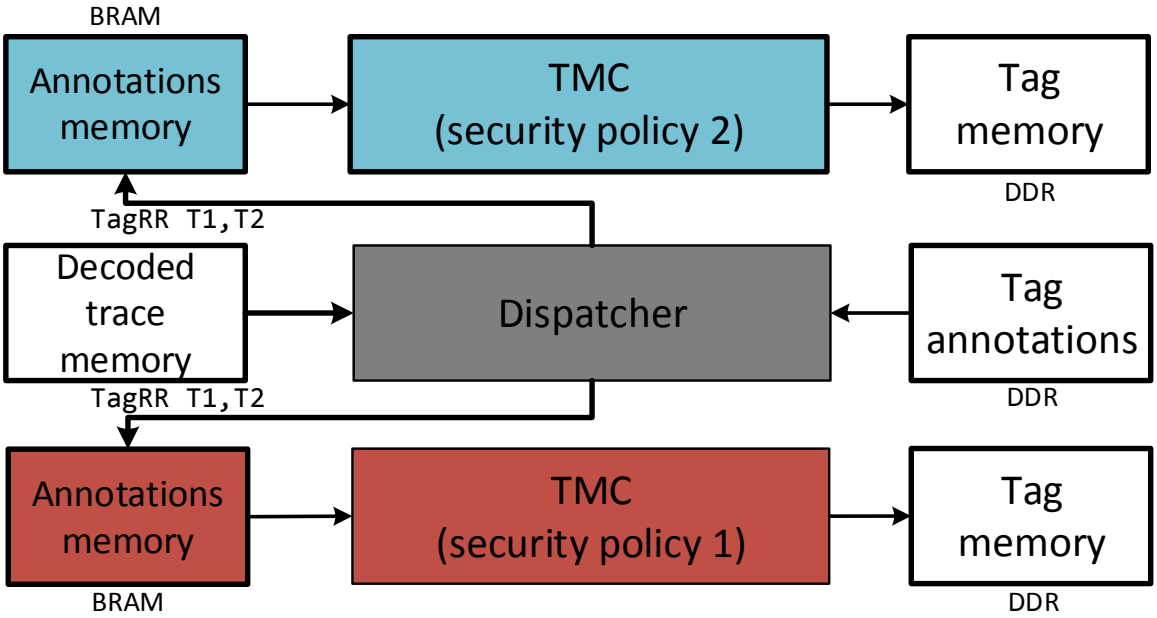


Figure 5.11 – DIFT coprocessor for multiple security policies.

5.5.1.2 Multiple threads

Real-world applications use multiple threads in order to speed up execution. However, existing DIFT mechanisms are not able to track multiple threads because the interface between the main processor and the DIFT coprocessor does not export information that allows determining the thread being executed on the CPU. We overcome this issue by exploiting CoreSight components to extract the context ID, which comprises the TID (*Thread ID*) and the ASID (*Application-Specific ID*).

Figure 5.12 shows the architecture of the DIFT coprocessor if two threads are to be monitored. The architecture looks similar to the architecture used in order to protect multiple security policies (Figure 5.11). The main difference is that annotations managed by TMC units are different. The dispatcher is responsible for writing annotations for each thread in its corresponding annotations memory. The dispatcher can determine the thread number thanks to the information contained in the decoded trace.

Single core If two threads are considered and both run on the same CPU core, the context ID field, retrieved by decoding the PTM trace, allows to determine which thread is currently being executed. Figure 5.13 shows the format of decoded traces in memory. The trace contains an I-sync packet (in green) including 4 bytes of context-ID (underlined). It can be noticed that the trace contains the same ASID value (`42`) and two different TID values (`4d2` and `4d3`) corresponding to each thread.

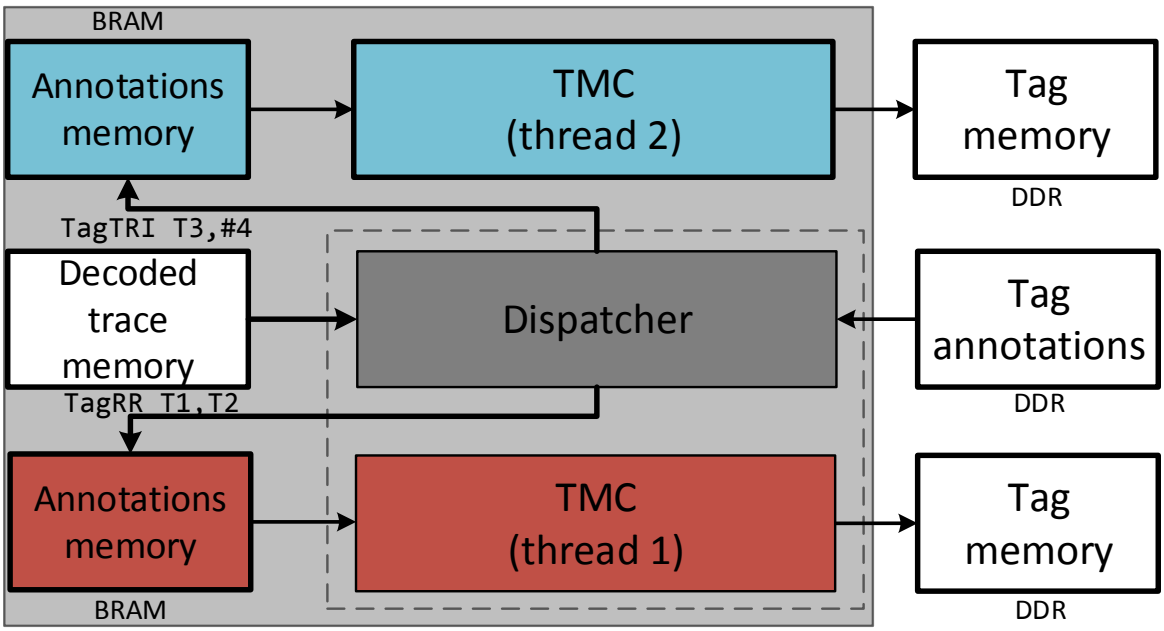
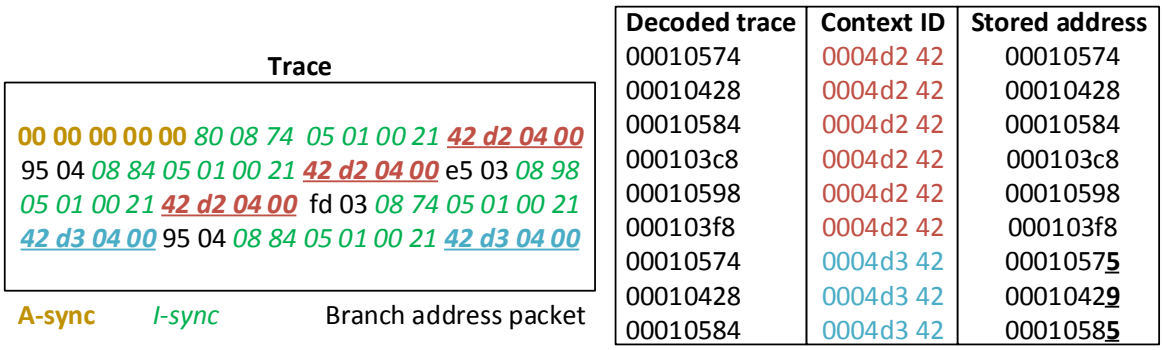


Figure 5.12 – DIFT coprocessor for multiple threads.



Decoded trace	Context ID	Stored address
00010574	0004d2 42	00010574
00010428	0004d2 42	00010428
00010584	0004d2 42	00010584
000103c8	0004d2 42	000103c8
00010598	0004d2 42	00010598
000103f8	0004d2 42	000103f8
00010574	0004d3 42	00010575
00010428	0004d3 42	00010429
00010584	0004d3 42	00010585

Figure 5.13 – Decoded trace for multiple threads.

The PFT decoder computes the branch address where the ARM core has jumped. These addresses are 4-byte aligned (in ARM state which is the only state allowed to be used during compilation) which means that two bits (0 and 1) are always equal to 0. The PFT decoder also recovers the context ID field and uses these unused two bits to specify whether the decoded trace is generated by the first thread or the second thread. Figure 5.13 shows that for the second thread, the stored addresses in the decoded trace memory are not 4-bit aligned due to the storage of context ID in last two bits. The value of the context ID still needs to be stored in separate registers so that if an attack is detected, the interrupt routine gets the TID of the program in order to kill the process in charge of generating unauthorized behavior.

Multi-core system If two threads are launched on two different CPUs, the same architecture can be used to propagate tags. However, the trace configuration and the PFT decoder

needs to be adapted. If the second core is being used to run the program, then the second PTM needs to be configured as well to trace the program. In terms of configuration, when both PTMs are enabled, each PTM needs to insert a trace ID packet so that the funnel [57] can merge traces onto a single bus. Furthermore, trace sinks (TPIU or ETB) must enable formatting to differentiate trace from different sources. Therefore, the PFT decoder should be adapted in order to consider formatted data instead of raw data considered in this work. The rest of the architecture remains the same as it is similar to multiple threads on a single core case. For multi-core case, a tracing overhead will appear due to trace formatting that adds a low overhead of 6% [72] and an overhead of one byte every time the trace bus switches between trace sources.

5.5.2 Performance overhead

In [4], authors show that the overall overhead of instrumentation is mostly affected by the runtime overhead of instrumentation. They show that it represents 90 % of the total execution time required for DIFT operations. Therefore, by improving this overhead, the overall time required for implementing DIFT is reduced.

5.5.2.1 Runtime overhead of instrumentation

Figure 5.14 shows the average normalized execution time overhead for instrumented application binaries. The related work implementation strategy is detailed in [4]. Two instrumentation strategies (strategy 1 and 2) are adapted from ARMHEx ([62]) which has the same software considerations as existing work [4]. In this work, all information flows are considered in this work. Library code is instrumented unlike [62, 4] and a different configuration of Coresight components is used allowing to recover the context ID information which changes the design of hardware modules such as the PFT decoder.

Figure 5.14 shows that if the related work strategy (as in [4]) is used, the instrumentation overhead is 12.79 times higher than the original execution time. The instrumentation strategy 1, where all memory instructions are instrumented, adds significant execution time overhead (on average 10.43 times higher than the original execution time). The instrumentation overhead is high for applications that require more memory operations (such as `lu` and `matrix`). However, if only register-relative (other than `PC`, `SP` and `FP`) memory instructions are instrumented, as in strategy 2, the average communication time overhead is reduced by a factor 3.8 to achieve 3.35 times higher execution time on average.

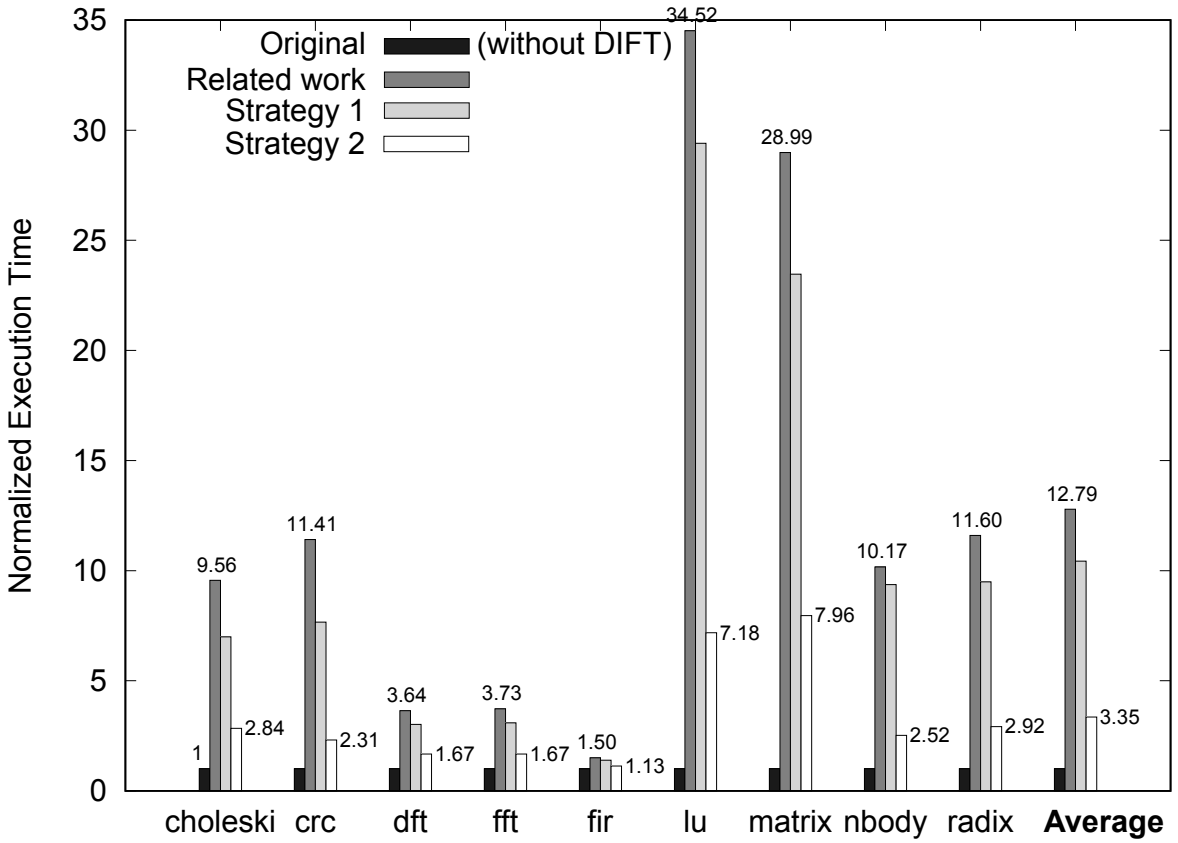


Figure 5.14 – This Figure shows the runtime overhead of instrumentation using a custom benchmark. The solution proposed in this work outperforms the related work strategy thanks to the negligible runtime overhead of CoreSight components. In average, it can reduce the runtime overhead by a factor 3.8. It can reduce the runtime overhead up to 5 times better than the related work instrumentation strategy.

The three main reasons why this overhead remains high are:

- This work targets hardcore CPU which is not the case in most existing works such as in [3, 4, 1, 40].
- The static analysis considers all information flows rather than function level information flows as in [4, 62].
- It also instruments library code used by the applications unlike most existing works [3, 4, 62].

5.5.2.2 Memory footprint of instrumentation

Figure 5.15 shows the memory space overhead of custom benchmark applications using both strategies. Binaries are statically compiled i.e. all the code executed by the application is inside the code section of the binary. Therefore, results obtained here take into account all modifications of user code and the library code. The instrumentation strategy 1 adds in

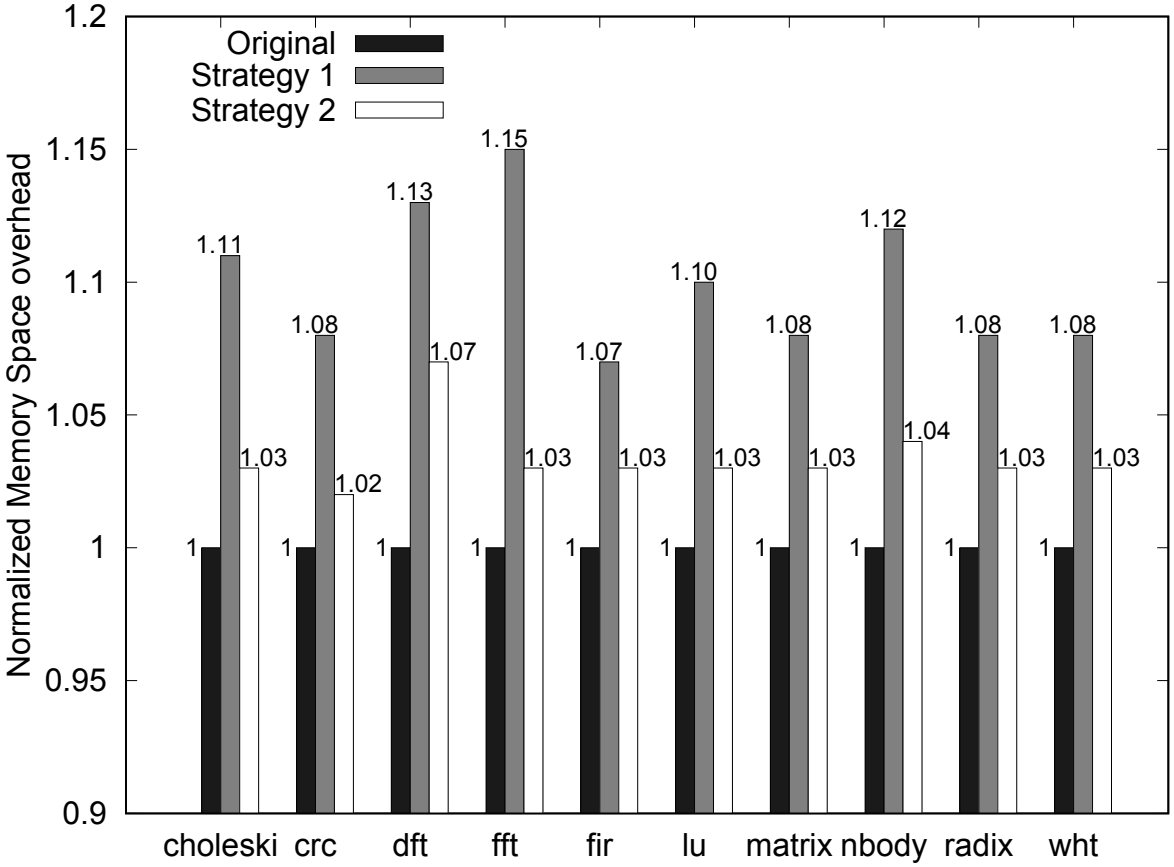


Figure 5.15 – Memory footprint of custom benchmark.

average 10% of memory space overhead and instrumentation strategy 2 adds in average only 3% of memory space overhead.

5.5.2.3 Area

Table 5.4 – Post-synthesis area results on Xilinx Zynq Z-7020.

IP Name	Slice LUTs (in %)	Slice Registers (in %)	BRAM Tile
Dispatcher	2223 (4.18%)	1867 (1.75%)	3
TMC	1837 (3.45%)	2581 (2.43%)	6
PFT Decoder	121 (0.23%)	231 (0.22%)	0
Instrumentation	676 (1.27%)	2108 (1.98%)	0
Blare PS2PL	662 (1.24%)	2106 (1.98%)	0
Blare PL2PS	62 (0.12%)	56 (0.05 %)	0
Decoded trace memory	0	0	2
AXI Master	858 (1.61%)	2223 (2.09 %)	0
TMMU	295 (0.55%)	112(0.10 %)	3
AXI Interconnect	2733 (5.14%)	2495 (2.34 %)	0
Miscellaneous	1381 (2.6%)	2160 (2.03%)	0
Total Design	**10848 (20.39%)**	**15939 (14.98%)**	**14 (10%)**
Total Available	**53200**	**106400**	**140**

Table 5.4 shows the area required to implement the architecture presented in Figure 5.4. Most of the FPGA area is filled by AXI interconnect (5.14%), dispatcher (4.18 %) and TMC (3.45%). The overall design takes 20.4% of the FPGA area.

If two security policies are required at the same time, the design would be modified as shown in Figure 5.11. The overall design, in case of two security policies, would require an additional TMC core and some IPs used the TMC core that all would take additional 4095 slice LUTs, 9074 slice registers (i.e. 8% additional FPGA logic) and 6 BRAM tiles. Tools do not allow to use all available FPGA area due to the efficiency of placing and routing algorithms used as well as the characteristics of targeted FPGA. In practical, around 80 % of FPGA area can be successfully routed and placed on the FPGA. As 20 % of the FPGA area is filled by the overall design, 60 % of FPGA area can be used to run other TMC units as well as the IPs required. In other words, the proposed design can run more than 8 security policies or protect more than 8 processes at the same time in this targeted SoC which is the second smallest SoC of Zynq family.

5.5.2.4 Power

Table 5.5 shows the power overhead of the overall architecture with default synthesis settings for the ZedBoard. The power overhead consists only of the power overhead of the FPGA part because the ARM core is running the application. Therefore, the power overhead of the overall architecture is the sum of the power consumption of clocks, signals, logic and BRAM that are located in the FPGA part which equals to 0.294 W (with a deviation of 20 %). It represents 16.2 % of the baseline power consumption of Zynq processing system (1.815 W). The obtained results are provided by Vivado tools using the post-synthesis design.

Table 5.5 – Power results of overall architecture on Xilinx Zynq Z-7020.

	Power consumption (in Watt)
Clocks	0.061
Signals	0.039
Logic	0.029
BRAM	0.011
PS7	1.522

5.5.3 Comparison with previous works

Table 5.6 – Performance comparison with previous off-core approaches.

Approaches	**[2]**	**[3]**	**[4]**	**[4] adapted**	**ARMHEx**	**This work**
Area overhead	6.4%	14.8%	14.47%	N/A	**0.47%**	0.95 %
Power overhead	N/A	**6.3%**	24%	N/A	8.45%	16.2%
Max frequency	N/A	**256 MHz**	N/A	N/A	250 MHz	250 MHz
Communication time overhead	N/A	N/A	60%	1280%	**5.4%**	335%
Hardcore portability	No	No	**Yes**	**Yes**	**Yes**	**Yes**
Main CPU	Softcore	Softcore	Softcore	**Hardcore**	**Hardcore**	**Hardcore**
Library instrumentation	N/A	N/A	partial	**Yes**	No	**Yes**
All information flows	No	No	No	**Yes**	No	**Yes**
FP support	No	No	No	No	No	**Yes**
Multi-threaded support	No	No	No	No	No	**Yes**

Table 5.6 shows the comparison of this work off-core approach with previous off-core approaches. Comparison of this work with in-core solutions is not done because they are not

hardcore portable and are very invasive contrary to the off-core solution used in this work. Compared to ARMHEx ([62]), this work has an higher area overhead because it provides support for FP and multi-threaded software that requires additional modules in the FPGA area. Furthermore, the power overhead of this work is higher than ARMHEx because this work uses two times more FPGA area. The maximum frequency achievable is comparable to existing works.

The most important difference resides in communication time overhead. It may appear that this overhead is higher than values reported in related works. The fifth column, Heo [4] adapted, shows that if the instrumentation strategy proposed in [4] is adapted with the same static analysis and information flows constraints as in this thesis, then the communication time overhead obtained on Zynq architecture reaches 1280%. However, the solution proposed in this thesis can reduce this overhead to 335% by a factor of $\frac{1280}{335} = 3.8$. The best communication time overhead reported in existing works is 5.4% (ARMHEx) but the solution proposed in ARMHEx lacks support for all information flows and library instrumentation. Furthermore, in ARMHEx, the time overhead has been estimated based on the number of instrumented instructions whereas the value reported in this work is obtained on the Zedboard platform and the time is measured using `perf` tool on Linux kernel v4.9. It is very important to consider all information flows and instrument libraries because it allows detecting an important range of attacks unlike most existing works. For instance, a simple attack on existing works could be to add a wrapper around library function and use Linux kernel dynamic `LD_PRELOAD` feature to avoid detection of any malicious library code. However, this work is able to detect the execution of malicious library code as its tracking is not ignored as in most existing works. Moreover, this work provides support for missing features in related works: support for floating point (through additional instructions and additional register file) and multi-threaded software thanks to context ID feature of the CoreSight PTM as described in section 5.5.1.

5.6 Summary

This chapter details the architecture of the custom DIFT coprocessor. The DIFT coprocessor has been designed for the ARM architecture and provides support for missing features in related works such as specifying security policies at compile-time or runtime, support for floating point, support for multiple security policies of different tag granularity and support for multiple threads. The DIFT coprocessor is divided into two units: the dispatcher and the TMC core. The dispatcher determines the execution order of the program being executed on the ARM core thanks to the decoded trace and stores corresponding annotations in order for the TMC. The TMC is responsible for computing and propagating tags. The TMC uses a TMMU in order to ease the management of memory tags. The DIFT coprocessor is modular allowing to reuse developed units for different purposes. The DIFT coprocessor can be easily

adapted to run multiple security policies or to monitor multiple threads. This is done by adding another dedicated TMC core along with the required IPs. The instrumentation time overhead, limiting factor in existing works DIFT architecture, is reduced by more than 380% if compared to existing work solution [4] with same software constraints. Area results show interesting perspectives in terms of implementing multiple security policies and protecting multiple processes. For instance, protecting two threads requires another TMC unit which adds an additional area overhead of 8%.

Chapter 6

Hardware accelerators for malware detection

This chapter presents the work done during a 3-month research stay within the ALaRI institute in Lugano, Switzerland. It focuses on the anomaly-based malware detection scheme rather than the taint-based information flow tracking approach previously used. It gives a complementary point of view on hardware-based software security approaches.

Contents

6.1 Introduction

In previous chapters, the dynamic taint analysis technique, DIFT, has been used in order to detect software attacks in embedded systems. There is an important amount of works that deal with detecting malware on the Android OS using taint-based detection scheme [23]. In this chapter, another dynamic approach, named anomaly-based detection, is studied in order to detect malware applications in Android. The work presented here has been done in the frame of a mobility, funded by the UBL (*Université de Bretagne-Loire*), at ALARI (*Advanced Learning and Research Institute*) lab of USI (*Università della Svizzera Italiana*) at Lugano, Switzerland. This work is the continuation of [16] where the author proposed an approach to detect malware applications. Figure 6.1 shows the proposed malware detection system with both offline and runtime components. The offline part basically consists in creating a dataset in order to train classification algorithms. This is done by executing applications

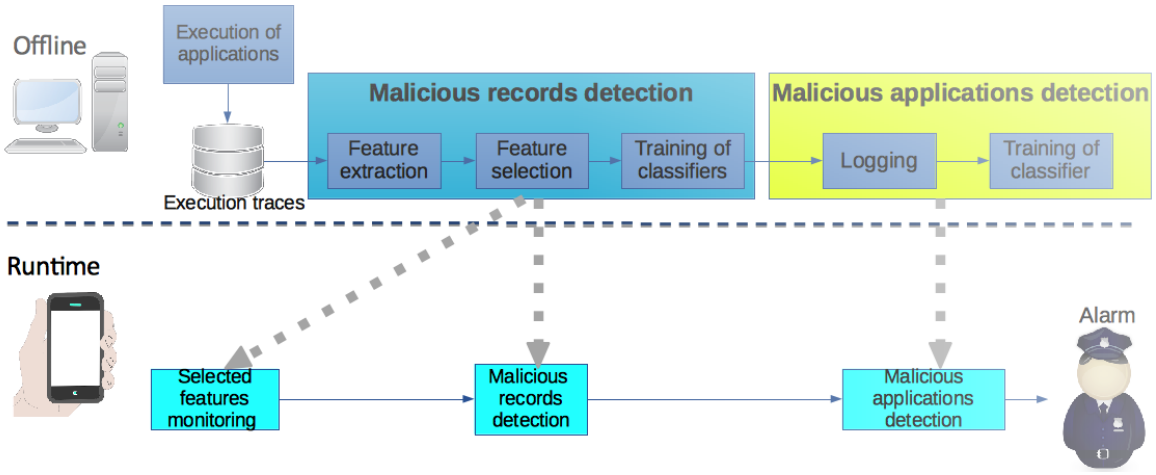


Figure 6.1 – Proposed detection system (taken from [16])

and recovering execution traces. These traces are used to train the classification algorithms. The runtime part consists of measuring some selected features and testing the application using the trained classification algorithms. The runtime part runs on a battery-based device making its energy efficiency a critical feature as it is one of the most important factors in battery-powered devices. If the detection scheme takes an important amount of power, it would result in a fast discharge of the device making it impossible to use it for other purposes. Besides, the FPGAs are starting to be used in the mobile devices [73] and can provide a solution to implement energy-efficient algorithms for important services such as security. The goal of this chapter is to provide an energy and performance efficient implementation of the runtime part of the proposed approach in [16].

6.2 Proposed hardware accelerators

The key idea of the solution proposed by Milosevic [16] is to use an FPGA in order to implement the malicious applications detection (more specifically, classification algorithms, see Figure 6.1). Another important goal is to maintain the accuracy of the software solution in the hardware-assisted upgrade.

6.2.1 Design methodology

The approach proposed in [16] is designed using the methodology shown in Figure 6.2. First, the classification algorithm is implemented in C code. Using the C code and some directives, Vivado HLS (*High-Level Synthesis*) tool is used in order to generate a hardware version of the C code. The generated hardware IP is tested in simulation, using Vivado HLS co-simulation, in order to make sure that the generated IP behaves correctly. Afterward, the IP is exported

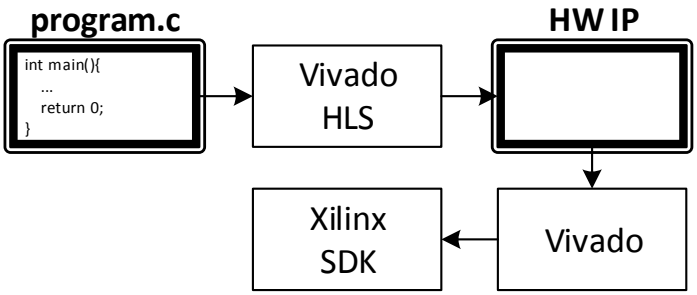


Figure 6.2 – Methodology used in order to develop hardware version of classification algorithms

to Vivado. Then, a Vivado design is made in order to make sure that the IP runs correctly. For this purpose, the design developed in Vivado is synthesized and implemented. Then, the FPGA is programmed using the bitstream generated from the implemented design. This design is exported towards Xilinx SDK and verified on the FPGA. This overall approach allows verifying that the proposed designed IP runs correctly in both simulations and on the hardware.

Once the IP is verified in implementation, a power-efficient hardware version of the C code is developed by using different optimization directives in Vivado HLS. Three classification algorithms are studied: LR (*Logistic Regression*), NB (*Naive Bayes*) and SVM (*Support Vector Machine*).

6.2.2 Power and performance optimizations

The primary goal of optimizations is to reduce power while the secondary goal is to not decrease performance. In order to develop a power efficient hardware version that does not decrease the performance when compared to the software version, some design considerations have been taken into account:

- All input interfaces are developed using BRAM interface. This is also true for the non-optimized hardware version as well. The other options for interfaces are AXI Master, AXI slave, ap_fifo, or ap_mem. All these interfaces take an important amount of FPGA logic and therefore have a higher power overhead than that of the BRAM interface. All output interfaces are designed using AXI-Lite interface.
- Loops are pipelined. This is not true in the non-optimized version. By pipelining loops, the hardware version takes more FPGA area but improves the performance by increasing the throughput.
- The “dataflow” directive is used in order to improve the execution time of the IP. As the goal is to focus on the improvement of both power and performance, this directive is used to parallelize functions when possible in order to improve the throughput.

6.2.2.1 LR

```
void classify(double x[TEST_SIZE][NB_FEATURES], double weights[NB_FEATURES],
    double output[TEST_SIZE]){
  double temp = .0;
  unsigned int i, k;
  double w, xtemp, prod;

  for (k = 0; k < TEST_SIZE; k++){
    temp = .0;
    for (i = 0; i < NB_FEATURES; i++){
      xtemp = x[k][i];
      w = weights[i];
      prod = w*xtemp;
      temp = temp + prod;
    }
    output[k] = sigmoid(temp);
  }
}
```

Listing 6.1 – Accelerated portion of the LR algorithm

Listing 6.1 shows the C code that is used in the runtime part in order to detect whether an application is a malware or not. If the output value is greater than a threshold (0.5 in this work), then the application is detected as a malware. Otherwise, the application is not detected as a malware. The `TEST_SIZE` number represents the number of applications that are being tested and `NB_FEATURES` is the number of features required in order to successfully classify the application. Basically, this function makes a dot product between features measured and a weight vector obtained by training the classification algorithm. Then, a mathematical function sigmoid is computed in order to determine the output value. Listing 6.2 shows the optimization directives applied to the source code shown in Listing 6.1.

```
set_directive_interface -mode bram "classify" output
set_directive_interface -mode bram "classify" weights
set_directive_interface -mode bram "classify" x
set_directive_interface -mode s_axilite "classify"
set_directive_pipeline "classify/classify_label0"
set_directive_unroll "classify/classify_label1"
set_directive_dataflow "classify"
```

Listing 6.2 – Optimization directives for the LR algorithm.

6.2.2.2 NB

```
void predict(double means[NB_CLASSES*NB_FEATURES], double variances[
    NB_CLASSES*NB_FEATURES], double priors[NB_CLASSES], double v[NB_FEATURES
    ], double *best_probability, unsigned *index){
  unsigned int i, j;
  static unsigned bestIndex;
  static double bestProb;
  static double probabilities[NB_CLASSES];
  double prob;

  calculateClassProbabilities(means,variances,priors,v,probabilities);
  bestProb = 0.0;
  bestIndex = 0;
  predict_label1: for (i = 0; i < NB_CLASSES; i++){
    prob = probabilities[i];
    if (prob > bestProb){
      bestProb = prob;
      bestIndex = i;
    }
  }
  *best_probability = bestProb;
  *index = bestIndex;
}
```

Listing 6.3 – Accelerated portion of the NB algorithm.

Listing 6.3 shows the runtime part of the NB classification algorithm. The runtime part consists of measuring some features of the application that are stored in vector `v`. The training part of the NB algorithm provides with `means` and `variances` in order to classify the malware application. The algorithm computes the probability using `means`, `variances`, and features provided in vector `v`. Then, the best probability is found using a simple comparison operation. If the application is a malware, the best probability is found for index 1. Otherwise, the best probability is 0. Listing 6.4 shows the optimization directives applied to the source code shown in Listing 6.3.

```
set_directive_interface -mode bram "predict" means
set_directive_interface -mode bram "predict" priors
set_directive_interface -mode bram "predict" variances
set_directive_interface -mode bram "predict" v
set_directive_interface -mode s_axilite "predict"
set_directive_dataflow "predict"
set_directive_pipeline "predict/predict_label1"
set_directive_pipeline "calculateClassProbabilities/
    calculateClassProbabilities_label2"
```

Listing 6.4 – Optimization directives for the NB algorithm.

6.2.2.3 SVM

```
unsigned classifier(float x[N], float weights[N]){
  float sum = 0;
  float res;
  unsigned output;
  classifier_label0:for (int i = 0; i < N; i++){
    sum += weights[i] * x[i];
  }
  output = (sum + bias) >= 0 ? 1 : 0;
  return output;
}
```

Listing 6.5 – Accelerated portion of the SVM algorithm.

Listing 6.5 shows the runtime part of the SVM classification algorithm that is implemented in hardware using Vivado HLS. The runtime part consists of performing a dot product and an addition operation. Then, the value is compared with a threshold value (0) in order to determine whether the application is a malware or not. Listing 6.6 shows the optimization directives applied to the source code shown in Listing 6.5.

```
set_directive_interface -mode bram "classifier" x
set_directive_interface -mode s_axilite "classifier"
set_directive_pipeline "classifier/loop"
set_directive_dataflow "classifier"
set_directive_interface -mode bram "classifier" weights
```

Listing 6.6 – Optimization directives for the SVM algorithm.

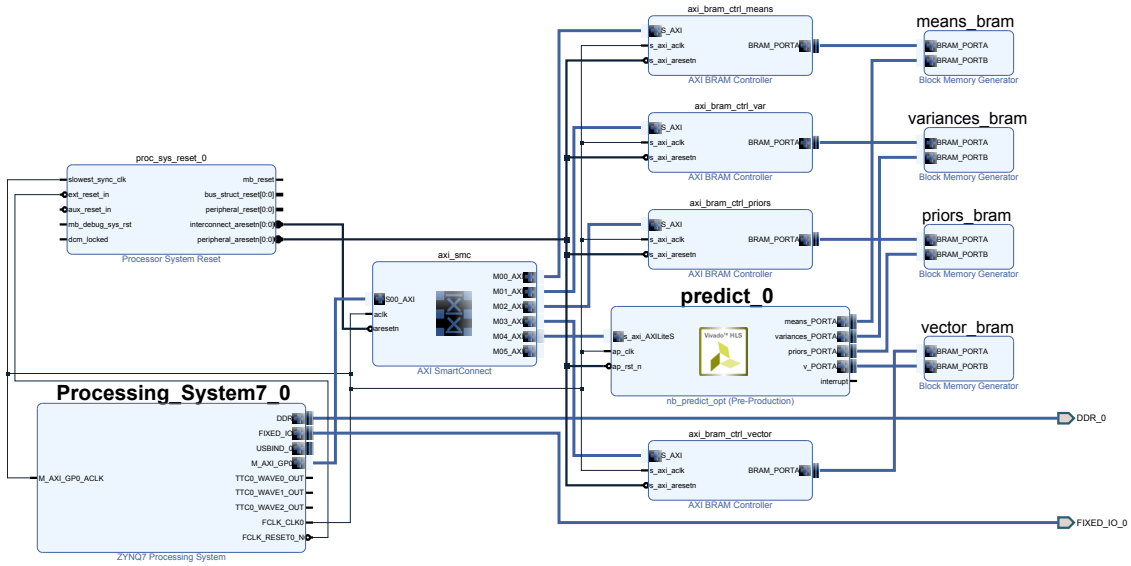


Figure 6.3 – This Figure shows the Vivado design used in order to verify the hardware version of the NB algorithm (**predict_0** IP). The Zynq **Processing_System7_0** IP represents the configuration of the ARM core on the Zynq SoC. The ARM core sends the data to the **predict_0** IP and recovers the return data in order to verify it with the result of software version. The ARM core also takes care of measuring the execution time for both software and hardware versions.

6.3 Implementation details and evaluation

6.3.1 Evaluation method

Figure 6.3 shows the implementation design with the NB algorithm hardware version. The `predict_0` IP has been generated by the Vivado HLS tool. The `Processing_System7_0` IP contains the default configuration of the ARM Core on the Zynq SoC. The memory blocks (`means_bram`, `variances_bram`, `priors_bram`, and `vector_bram`) contain the output of training information sent by the ARM core. The output of classification is sent back to the ARM core by the `predict_0` IP using AXILite interface. The `predict_0` IP shown in Figure 6.3 is the non-optimized hardware version. This design is synthesized and implemented on the FPGA. After implementation, a bitstream is generated and used to program the SoC. The ARM core sends inputs to the hardware version of the NB algorithm, recovers outputs and compares them with that of the software version. Xilinx tools 2017.1 are used on a Xilinx Zedboard with a Z-7020 SoC (dual-core Cortex-A9 running at 667 MHz and an Artix-7 FPGA) to implement the architecture shown in Figure 6.3.

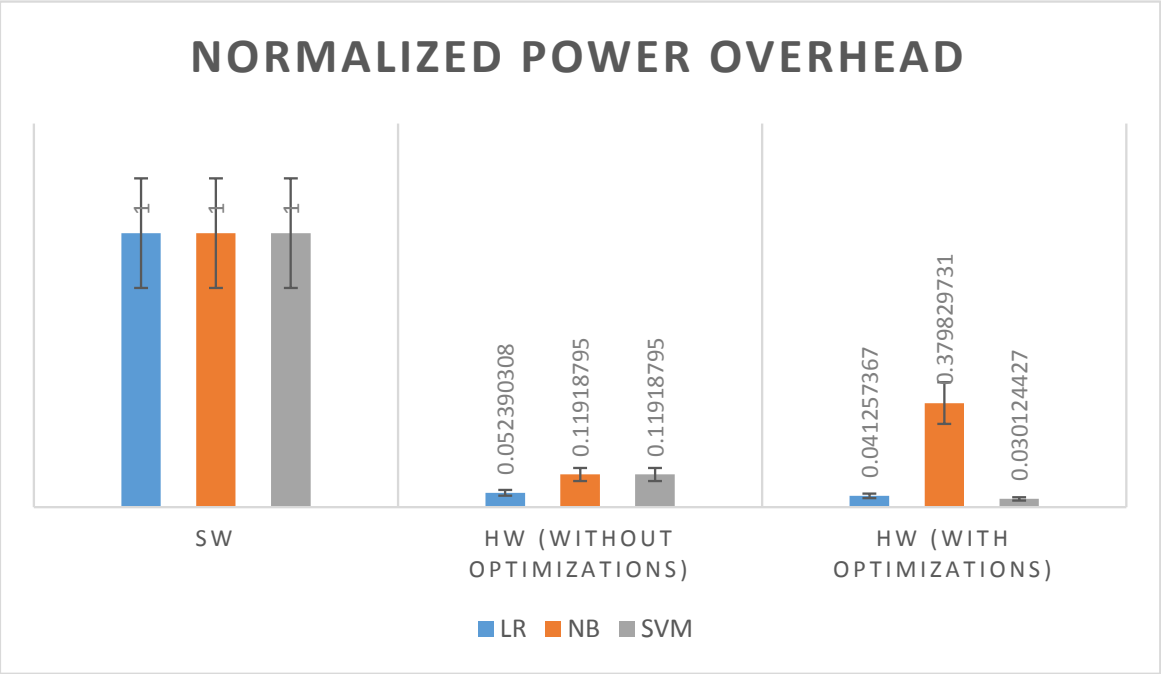


Figure 6.4 – Power overhead of the implemented algorithms

6.3.2 Results

6.3.2.1 Power overhead

Figure 6.4 shows the maximum power overhead of implemented algorithms. The software power consumption value is considered as the base value for comparison. There are two values reported for hardware versions: without and with optimizations. For the LR algorithm, the power overhead is improved by more than 19 times using a non-optimized version when compared to the power overhead incurred by the software solution. If the optimized version of IP is used, then the power overhead is improved by more than 24 times. For the NB algorithm, the power overhead is improved by more than 8.3 times when compared to the software solution. For the optimized version, the power overhead can be improved more than 2.6 times when compared to the software version. The power overhead of the optimized version in the NB algorithm is higher because the area occupied by the optimized algorithm is very high (Figure 6.5). Furthermore, the goal of optimization is to improve both power overhead and the performance of the NB algorithm. For non-optimized hardware version of the SVM algorithm, the power overhead is improved by more than 8.3 times when compared to the software solution while the optimized hardware version improves the power overhead by more than 33 times when compared to the power overhead of the software solution. It shows that the use of FPGA allows reducing the power overhead of classification algorithms,

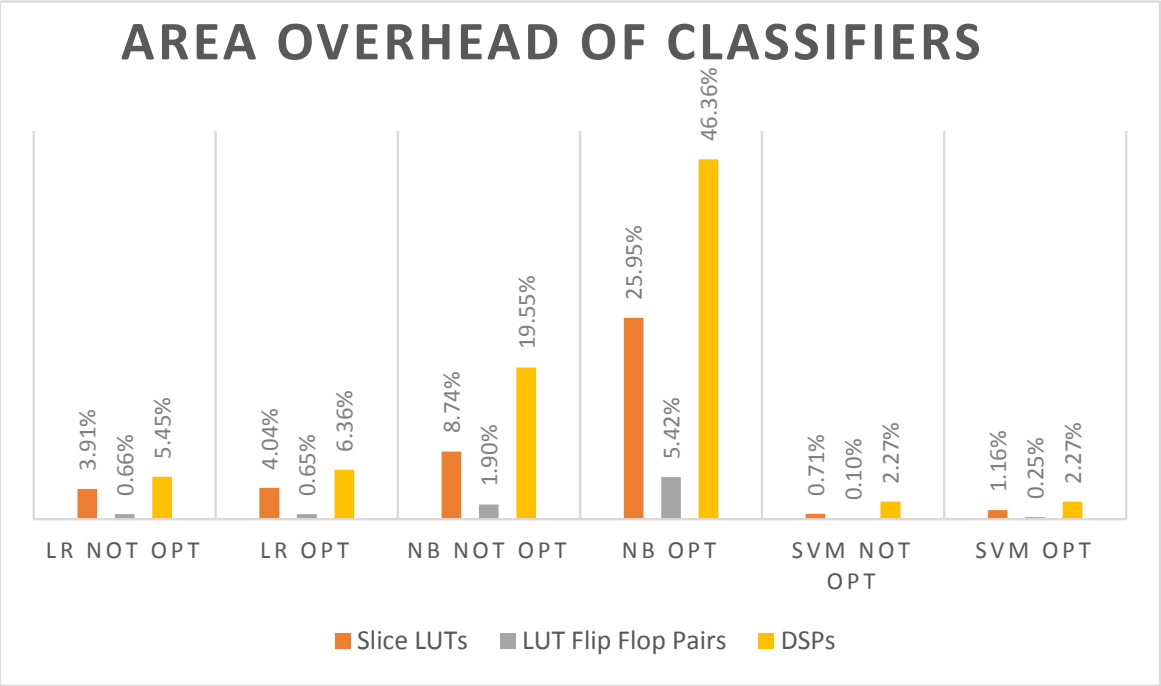


Figure 6.5 – Area overhead of the implemented algorithms using optimized and non-optimized hardware IPs. The non-optimized versions are more area efficient while the optimized versions require more FPGA area. As the goal of the optimization is to improve both power and performance overhead, the area taken by the optimized version is higher than the non-optimized hardware version.

from 2.6 times up to 33 times depending on the algorithm, used in the malware detection scheme proposed in [16].

6.3.2.2 Accuracy

The other important goal is to make sure that the accuracy remains the same for the hardware and software versions. This goal has been achieved by using floating point numbers in hardware in order to have the maximum precision. If the fixed point solution is used in hardware, then the precision is lost resulting in a difference of accuracy between the software and hardware version. Therefore, floating point precision has been used to not lose precision. The accuracy of the LR algorithm is 91.72 %. The accuracy of the NB algorithm is 53.75 % and the accuracy of the SVM algorithm is 89.30 %. The same accuracy values are obtained for the software and both hardware versions. The most accurate results are obtained using the LR algorithm while the SVM algorithm still allows to correctly classify more than 89 % of applications. The NB algorithm is the least accurate among the three implemented algorithms but hardware implementations do not degrade results.

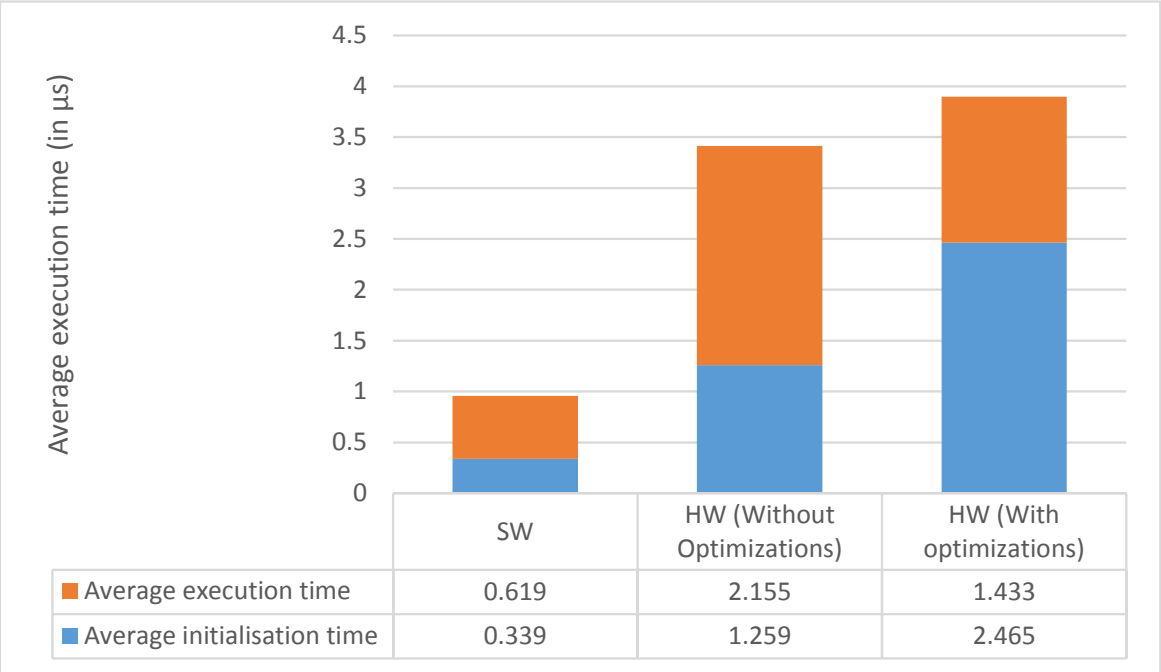


Figure 6.6 – Runtime overhead of LR algorithm

6.3.2.3 Runtime overhead

LR: Figure 6.6 shows the average runtime overhead introduced by the LR algorithm to classify a single application. The non-optimized hardware version requires 3.5 times higher execution time compared to the software version while the optimized version runs 4 times slower than the software version. This result can be explained by the fact that the software can run floating point operations faster than the hardware versions by using the FPU present in the ARM Cortex-A9 core. Furthermore, to improve power, the interface used to transfer data between the ARM core and the hardware version is AXI BRAM interface which takes a low area and low power. The runtime is further divided into two parts: initialization time and execution time. The initialization time is the time required to send the data in memory blocks from the ARM core while the execution time is the time required to run the algorithm and send the result back to the ARM core. The initialization time for hardware version is higher which is expected because the hardware gets data from the external DDR memory while for the software version, the data is cached. The execution time for the hardware version is also higher than the software solution because of the mathematical function which is not optimized in the hardware version generated by Vivado HLS. The software execution time is better thanks to the FPU but requires much more power than the hardware versions.

NB: Figure 6.7 shows the average runtime overhead introduced by the NB algorithm to classify a single application. The non-optimized hardware solution takes 26 % more runtime

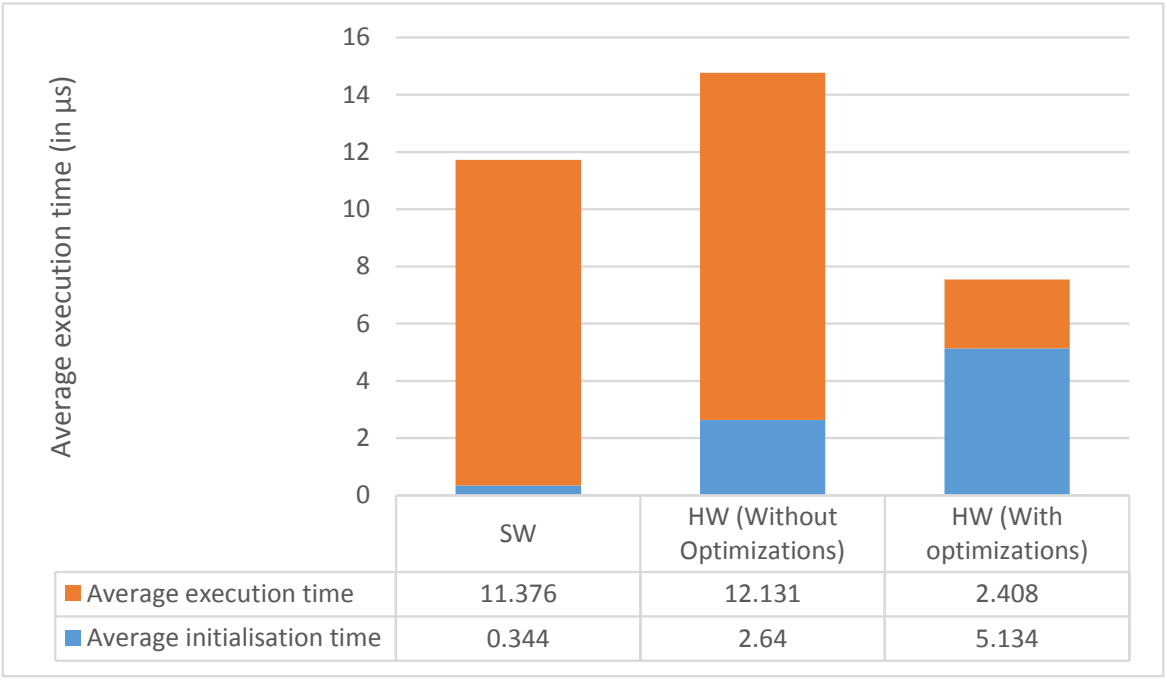


Figure 6.7 – Runtime overhead of NB algorithm

than the software solution. The execution time for the non-optimized hardware version is similar than the software version while the initialization time is much more important than the software version. This is due to the fact that the FPGA needs to fetch data from the DDR memory while the software version takes advantage of the data cache available on the ARM core. The power and performance optimized hardware version of the NB algorithm improves the runtime overhead by more than 55 % while it improves the power overhead considerably (by more than 2.6 times). The results show that the optimized version has a higher initialization time than the non-optimized solution. This is due to the usage of `datapath` directive in Vivado HLS which improves throughput by duplicating the ports. Instead of having a single port for an input, multiple (two in this work) ports are used for a single input. This results in an important initialization time which affects the total runtime overhead.

SVM: Figure 6.8 shows the average runtime overhead introduced by the SVM algorithm to classify a single application. The software version performs the best in terms of runtime. The software performs better than the hardware version due to two major reasons that were explained earlier for the LR algorithm. The initialization time for software version is lower due to the data cache while the execution time of the software version is better than the hardware solution is lower because of the FPU present in the ARM core. Furthermore, the goal of the optimization is to minimize both power and performance overhead. The power overhead of

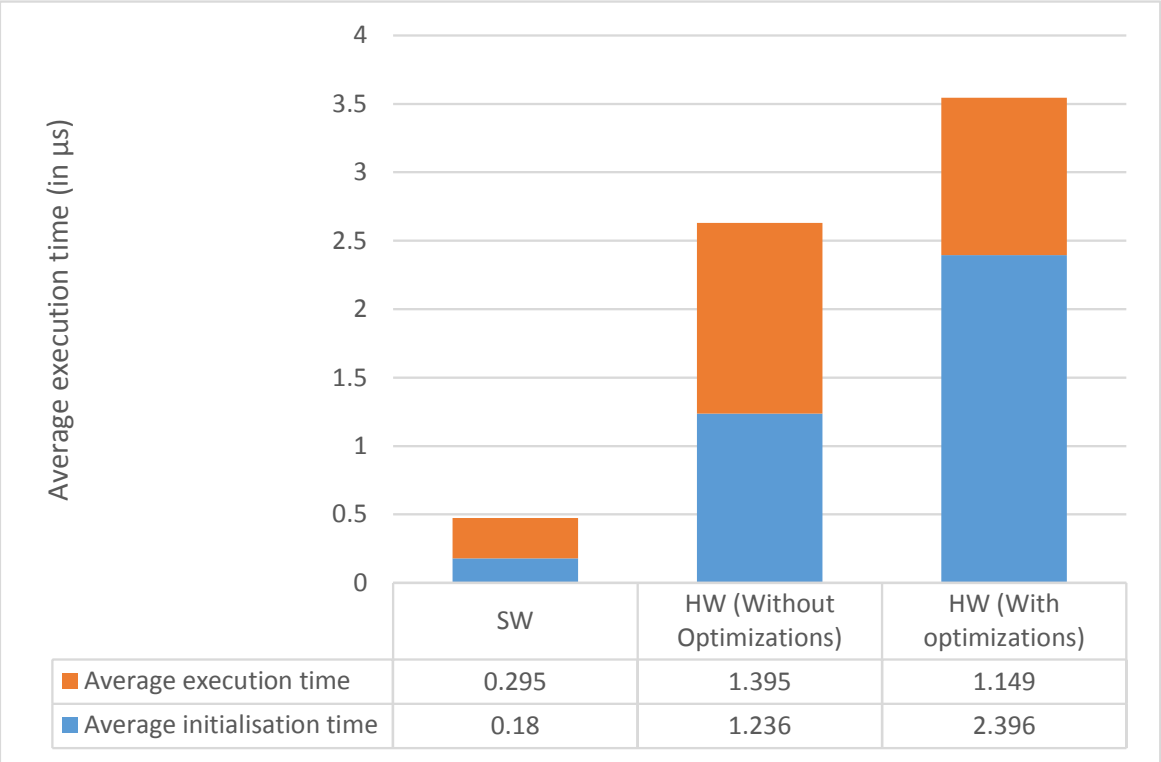


Figure 6.8 – Runtime overhead of SVM algorithm

the hardware version is still lower than the software version while the execution time for hardware versions is higher than the software solution. The BRAM interface uses fewer FPGA resources than other available interfaces to improve power but limits the performance improvement.

6.4 Summary

This chapter presents the approach proposed in order to improve the energy efficiency of the anomaly-based detection scheme detailed in [16]. The main idea is to take advantage of the FPGA in order to run the most power consuming part of the algorithm used in the malware detection scheme. The FPGAs are starting to appear in the mobile devices [73] and can provide a solution to implement energy-efficient algorithms for important services such as security. The hardware versions developed do not decrease the accuracy. The optimized SVM hardware version provides the most energy-efficient solution while the power and performance optimized hardware version of the NB algorithm has the lowest runtime.

Chapter 7

Conclusions and Perspectives

7.1 Conclusions

Recovering information required for DIFT on ARM hardcore CPUs

The first step towards DIFT on a hardcore CPU consists in recovering information required for DIFT on the FPGA part. In this work, a method to recover the information required for DIFT is proposed and implemented. This work takes advantage of CoreSight components in order to determine the program flow of the program on the FPGA side. The PTM component does not generate trace for each instruction executed by the ARM CPU. Therefore, a static analysis is performed that allows recovering missing information. However, the static analysis cannot determine the address of memory instructions such as `ldr`, `str`... The program is instrumented in order to recover these missing memory addresses. Most hardware-assisted existing works do not need more information. Nonetheless, there is still a missing piece of information from the kernel: information about system calls. The FPGA part needs to know the tag of the file being read by the kernel while the kernel needs to know the tag of the file being written. This information is recovered using modified RFBlare. The use of CoreSight components leads to a submission of two patches into the Linux kernel that have been integrated into the kernel.

Dedicated DIFT coprocessor for the ARM architecture

Once the information required for DIFT is recovered, the FPGA needs to propagate and check tag values. An operation needs to be done for almost every instruction of the program. Therefore, a dedicated DIFT coprocessor is required that takes care of propagating and computing tags with respect to a security policy. A few missing features in related works have been addressed in this work: security policy specification at compile-time or runtime, support for floating point instructions and a modular architecture. The dedicated DIFT

coprocessor is divided into two parts: dispatcher and TMC. The dispatcher takes care of reconstructing the program control flow of the application executed on the ARM CPU while the TMC decodes and performs the operation specified by either the compile-time security policy specified in the opcode of the annotation (i.e. instruction executed by the TMC).

Tracking of multiple security policies

The modular architecture of the DIFT coprocessor allows running up to 8 security policies in order to detect multiple software attacks. The dispatcher takes care of dispatching annotations to each of the 8 TMC that can be implemented using the proposed DIFT coprocessor.

Hardware-assisted DIFT for multiple threads

This work shows how the presented architecture can be used to protect multiple threads. The context ID information retrieved from the PTM component can be used in order to detect traces originating from different threads. Besides, trace for each thread respects the program execution order allowing to completely determine the execution order over the FPGA side. The modular architecture of the DIFT coprocessor can be used in order to run multiple threads. The dispatcher takes care of determining the annotations for each TMC and sending it to the specific TMC core.

7.2 Perspectives for future works

In this thesis, a problem of protecting software running on ARM-based hardcore CPUs, that has been overlooked in the existing works, has been investigated. The obtained results have raised some questions that are listed in the following as perspectives for future works.

Multi-threaded software

In this work, the proposed solution works without adaptation if multiple threads are running on the same core (section 5.5.1.2). However, if there are threads running on multiple cores, then the proposed solution needs to be adapted. The most important adaptation effort needs to be done around the configuration of CS components. Multiple PTM components and the TPIU formatter need to be enabled. The PTM decoder needs to be adapted as well in order to decode trace. It must “de-format” trace before decoding it.

TMMU and cache

In this work, a simple version of TMMU has been used in order to ease the management of memory space region in DDR memory containing tags for memory addresses. The TMMU can be improved by adding a tag cache in order to avoid fetching tags from memory for every memory operation. The TMMU and cache can help to improve the latency of memory instructions.

Evaluate different trace source components

There are different types of trace source components. In this work, the ARM PTM component has been considered. However, previous trace source components, still available in some processors (such as ARM Cortex-A53), include ETM trace component that can trace each instruction executed by the CPU rather than tracing only instructions that change program flow as done by the PTM. It means that the static analysis is not required on these processors. Furthermore, if a data trace component is included in CS components, then the trace component can also recover memory addresses. In that case, instrumentation would not be required anymore resulting in fewer software modifications. However, trace source components are being replaced by the recent PTM component which improves trace bandwidth and the processor design.

Find new security policies

All the work done in this thesis can be used to find new security policies. In this thesis, the focus has been put on the development of architectural support in order to implement DIFT. As the architecture is designed and implemented, different tests can allow finding new security policies in order to detect new software attacks.

Protect the kernel code

In this work, the focus has been put on protecting userspace code. However, the PTM component allows tracing only the kernel code as well. Besides, protecting the kernel code is much more challenging as the static analysis is much more complex.

Final Words

The work proposed in this thesis is one of the first works to target the ARM architecture for providing security features. The proposed solution in this work overcomes the visibility issue

faced when implementing DIFT on a hardcore CPU by using CS components, static analysis, instrumentation, and modified RFBlare. Furthermore, this is one of the first works to take advantage of the CS TPIU component to export trace towards the FPGA part and decode it on the fly. The modular architecture proposed can be used to implement multiple security policies and protect multiple threads. The idea of using a general purpose processor to implement the dispatcher is to provide flexibility to users in order to implement other security techniques using the same coprocessor. Besides, the architectural support provided in this work raises some interesting perspectives for future works: improvement of the thread model used, the architecture of some components used (TMMU), the discovery of new security policies and the protection of kernel code.

Appendix A

ZedBoard development board

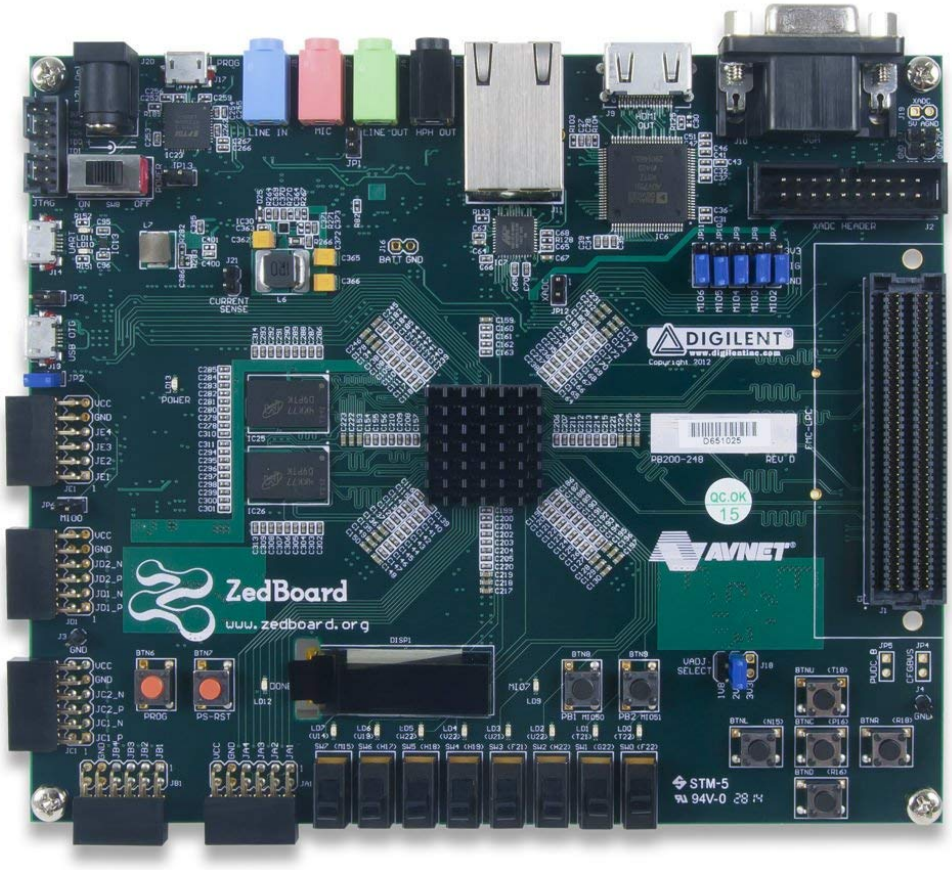


Figure A.1 – Zedboard

All implementations in this work are done on the ZedBoard (Figure A.1). ZedBoard integrates a Zynq SoC (Figure A.2) with a number of hardware peripherals (HDMI, audio, OLED, switches, LEDs...). The Zynq SoC integrates a dual-core ARM Cortex-A9 CPU based PS (*Processing System*) with a Xilinx 7-series FPGA fabric as the PL (*Programmable Logic*).

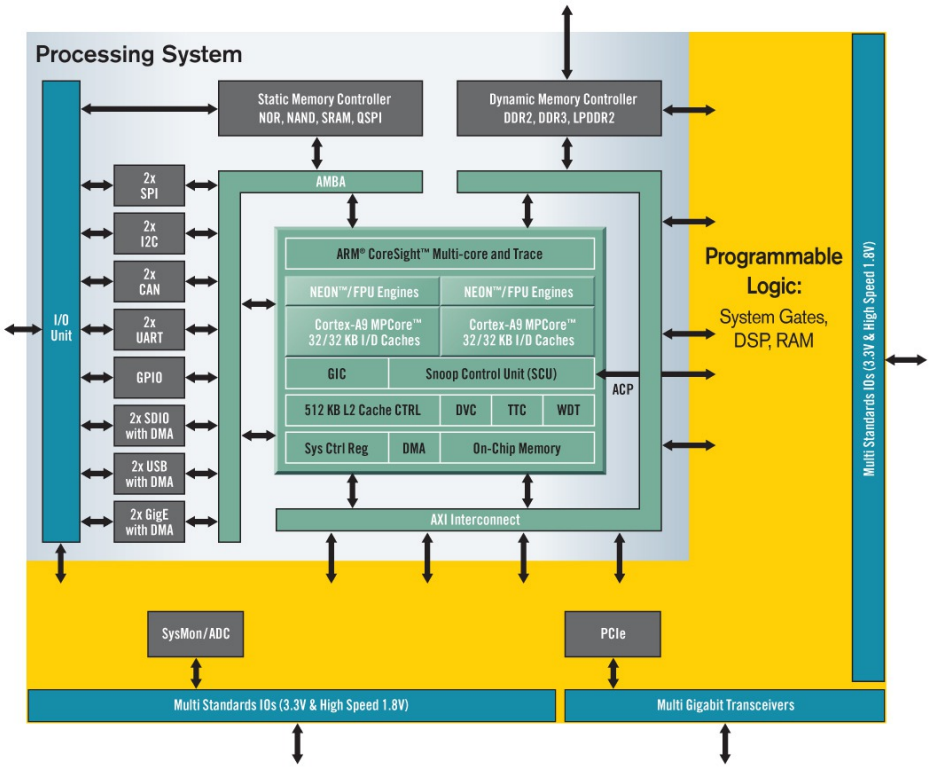


Figure A.2 – Zynq SoC

The most important components of the Zynq SoC used in this work are the following:

- Cortex-A9 CPU used to run the application.
- CoreSight components programmed to trace the application.
- Interfaces between PS and PL used to recover information required for DIFT on the PL.
  - AXI GP (*General Purpose*) ports.
  - AXI HP (*High Performance*) ports.
  - AXI ACP (*Accelerator Coherency ports*).
  - EMIO.

There are other peripherals that can be used by taking advantage of available Linux kernel drivers. The device tree is in charge of enabling the drivers using “compatible” field. The Zynq core runs at the maximum frequency of 667 MHz whereas the PL uses different frequencies varying between 50 to 250 MHz depending on the IP and the design.

There are two types of interface: Master and Slave. The Master interface means that the PS is responsible for initializing communication with the PL while the slave interface allows the PL to access hardware peripherals such as the DDR memory or UART...

Appendix B

Development and Evaluation method

This chapter provides details about how implementations and tests are done. The section B.1 explains how hardware IPs have been developed and tested using Xilinx tools. It also contains information about software modifications in order to test the developed design. Then, the section B.2 shows the software development done in order to take advantage of CoreSight components. Finally, the section B.3 provides details about how the evaluation is performed.

B.1 Hardware development

First, the hardware is designed in HDL (VHDL or Verilog) and simulated using the Vivado simulator. Once the behavioral simulation is valid, the code is packaged as an IP, following the procedure described in [74], to be used with other IP blocks provided by Xilinx.

The design flow used in order to test the developed IPs is shown in Figure B.1. A Vivado design is made containing the tested IP along with Zynq Processing System IP and other IP blocks provided by Xilinx. The design is synthesized and implemented. After implementation, a bitstream is generated. The design is exported towards software in order to program the Zynq PS.

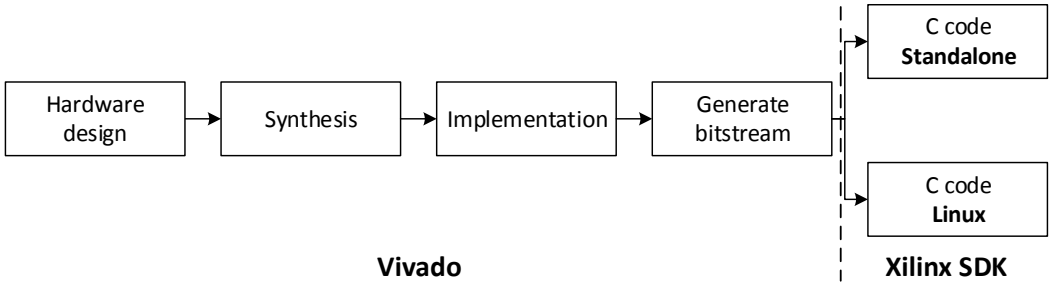


Figure B.1 – Development design flow with Xilinx tools

The exported design from Vivado contains the hardware definition file allowing the Xilinx SDK to build a BSP (*Board Support Package*) for the design. The BSP provides drivers for peripherals used in the design. The C code can be written to communicate with the developed IP block in order to make sure that the design works correctly.

B.1.1 Standalone

The first tests of the design are done in standalone "OS" made available by Xilinx [75] which provides a simple set of drivers for peripherals available on the targeted board. C code is written in order to modify input memory or to send the enable signal to the FPGA part.

B.1.2 Linux Kernel

Tests are made on the Linux kernel by adapting the previously developed program for the standalone OS. For instance, instead of accessing physical addresses, virtual addresses are obtained for physical addresses using the `mmap` system call and used in the program. The program is compiled using the cross compiler obtained using Yocto toolchain [76].

B.2 Software development

The software development, done on the Linux kernel, mainly consists in taking advantage of CoreSight components. There was no support for CoreSight components of the Zynq SoC in the Linux kernel. After adding support for these components, it was noticed that the PTM component did not have the branch broadcast feature available in the driver. Therefore, this feature has been added. Both these patches have been submitted to the Linux kernel and have been integrated into the Linux kernel release 4.11. The trace generated by the PTM needs to be sent towards the FPGA part using the TPIU component. However, the TPIU driver was not functional. Therefore, another patch was developed in order to modify the TPIU driver to be usable. All patches developed in this work are available at the git repository [77].

Once the trace was available on the FPGA, it needs to be decoded. At first, a software decoder has been developed that is made available on the git repository [77]. The decoder has been adapted from the decoder available in the Android kernel [78].

B.3 Evaluation details

B.3.1 Area evaluation

The area is evaluated using Vivado tools. The tools report the area after synthesis and implementation. The provided value is obtained using the post-implementation design except if specified otherwise.

B.3.2 Power evaluation

The power is evaluated using Vivado tools. Vivado tools provide the maximum power value of the design. The power value is an estimation and has an accuracy of +/- 20 %. Once the implementation or synthesis is done, the implemented or synthesized design is opened in order to get the report of power that the tools estimate based on the area occupied by all IPs in the design.

B.3.3 Runtime overhead

The runtime overhead is measured on the Linux kernel using `perf` tool [79].

Appendix C

The DIFT coprocessor instruction set details

C.1 DIFT coprocessor

The DIFT coprocessor is made up of two cores: the dispatcher and the TMC. This chapter provides all instructions available on the DIFT coprocessor: dispatcher instruction set described in C.1.1 and TMC instruction set detailed in C.1.2.

C.1.1 Dispatcher instruction set details

Table C.1 shows all available instructions on the dispatcher including their opcodes. The opcode for each instruction is choosed so that the program compiled using the GCC MIPS compiler [71] can be directly executed on the dispatcher without any software transformations. Table C.2 shows the values for the `FUNC` field used in the `A/L` instruction.

C.1.2 TMC

Table C.3 presents all available instructions, called annotations in order to differentiate from instructions of the dispatcher, on the TMC including their opcodes. The program running on the TMC is obtained using static analysis. However the execution order of annotations depend on the path taken by the ARM core. Therefore, the dispatcher is responsible for ordering annotations by analyzing the decoded trace.

	Opcode
0x0	`A/L`
0x1	Branch
0x2	n.u.
0x3	JAL
0x4	BEQ
0x5	BNE
0x6	BLEZ
0x7	BGTZ
0x8	ADDI
0x9	ADDIU
0xa	SLTI
0xb	SLTIU
0xc	ANDI
0xd	ORI
0xe	XORI
0xf	LUI
0x10	COP
0x11-0x1f	n.u.
0x20	LB
0x21	LH
0x22-0x23	n.u.
0x24	LBU
0x25	LHU
0x26-0x27	n.u.
0x28	SB
0x29	SH
0x2a-0x3f	n.u.

Table C.1 – Dispatcher instruction set

	FUNC
0x0	SLL
0x1	
0x2	SRL
0x3	SRA
0x4	SLLV
0x6	SRLV
0x7	SRAV
0x8	JR
0x9	JALR
0xa	MOVZ
0x10	MFHI
0x11	FTHI
0x12	MFLO
0x13	MTLO
0x18	MULT
0x19	MULTU
0x1a	DIV
0x1b	DIVU
0x20	ADD
0x21	ADDU
0x22	SUB
0x23	SUBU
0x24	AND
0x25	OR
0x26	XOR
0x27	NOR
0x28-0x29	n.u.
0x2a	SLT
0x2b	SLTU
0x2c	n.u.
0x2d	DADDU

Table C.2 – FUNC field of the `A/L` instruction

	Opcode	Instruction Format
0x01	LW	Init
0x02	SW	Init
0x03	LCR (load configuration register)	Init
0x04	Write a TLB entry	Init
0x05	LW_FP	Init
0x06	SW_FP	Init
0x07	tag_rri	Init
0x08	MOV_TRF_2_TRF-FP	Init
0x09	MOV_TRF-FP_2_TRF	Init
0x20	TagRImm	Init
0x21	TagRR	Init
0x22	TagMR	Init
0x23	TagRRR	TR
0x24	Tag_mem_tr	TI
0x25	Tag_tr_mem	TI
0x26	tag_arith_log	TR
0x27	tag_instrumentation_tr	TI
0x28	tag_tr_instrumentation	TI
0x29	Tag_kblare_tr	TI
0x2a	Tag_tr_kblare	TI
0x2b	lui	Init
0x2c	Tag_mem_tr_2	TI
0x2d	Tag_tr_mem_2	TI
0x2e	Tag_check	TR
0x2f	Tag_check_mem	TI
0x30	TagFRImm	Init
0x31	TagFRFR	Init
0x32	TagMFR	Init
0x33	TagFRFRFR	TR
0x34	Tag_mem_tfr	TI
0x35	Tag_tfr_mem	TR
0x36	tag_arith_log_fp	TR
0x37	tag_instrumentation_tfr	TI
0x38	tag_tfr_instrumentation	TI
0x39	Tag_kblare_tr_fp	TI
0x3a	Tag_tr_fp_kblare	TI
0x3b	lui_fp	Init
0x3c	Tag_mem_tfr_2	TI
0x3d	Tag_tfr_mem_2	TI
0x3e	Tag_check_fp	TR
0x3f	Tag_check_mem_fp	TI

Table C.3 – TMC core instructions (called annotations)

	FUNC
0x2	ADD
0x3	SUB
0x4	OR
0x5	AND
0x6	XOR
0x7	NOR
0x8	copy_src_1
0x9	copy_src_2
0x10-0x2d	not used yet

Table C.4 – Func field for the tag_arith_log annotation

Annotations are encoded using three formats described in section 5.4.2.2. All instructions used in the TMC core are detailed in order to understand the operation managed by each one of them.

1. **LW** (init format)
   reg_dst = Mem(reg_src + offset)
   General purpose instruction to load a value from an address value stored in a register.
   Conditions on instruction operands
   $0 \leq$ reg_dst ≤ 23
   $16 \leq$ reg_src ≤ 23

31 26	25 21	20 16	15 0
000001	reg_dst	reg_src	Immediate

2. **SW** (init format)
   Mem(reg_dst + offset) = reg_src
   general purpose instruction to store a value to a memory address.
   Conditions on instruction operands
   $0 \leq$ reg_dst ≤ 23
   $16 \leq$ reg_src ≤ 23

31 26	25 21	20 16	15 0
000010	reg_dst	reg_src	Immediate

3. **LCR** (init format)
   reg_dst_config_register = Mem(reg_src + offset)
   Load configuration register value from an address.
   Conditions on instruction operands
   $0 \leq$ reg_dst ≤ 23
   $16 \leq$ reg_src ≤ 23

31 26	25 21	20 16	15 0
000011	reg_dst	reg_src	Immediate

4. **Write a TLB entry** (init format)
   TLB[auto] = reg_src.
   Initialize a TLB entry corresponding to an address given in reg_src field.
   Conditions on instruction operands
   $0 \leq$ reg_dst ≤ 23.
   $16 \leq$ reg_src ≤ 23.

31 26	25 21	20 16	15 0
000100	n.u.	reg_src	n.u.

5. **tag_rri** (init format)

reg_dst = reg_src + imm_16.

Add a 16-bit value to a register. This is useful in order to update ARM SP register value or ARM PC register value.

Conditions on instruction operands

$16 \leq$ reg_dst ≤ 31.

$16 \leq$ reg_src ≤ 31.

31 26	25 21	20 16	15 0
000111	reg_dst	reg_src	imm_16

6. **Tag_reg_imm** (init format)

   reg_dst = imm

   initialize reg with an immediate value: this instruction can be used to initialize tag values or register values ...

   Conditions on instruction operands

   $0 \leq$ reg_dst ≤ 23

   $16 \leq$ reg_src ≤ 23

31 26	25 21	20 16	15 0
100000	reg_dst	reg_src	Immediate

7. **Tag_reg_reg tag** (init format)

   reg_dst <= reg_src

   initialize reg_dst with a value stored in register

   Move from GPR (of 2nd stage) to tag register file (registers 0 to 16)

   Conditions on instruction operands

   $0 \leq$ reg_dst ≤ 23

   $16 \leq$ reg_src ≤ 23

31 26	25 21	20 16	15 0
100001	reg_dst	reg_src	Immediate

8. **Tag_mem_reg** (init format)

   Mem([reg1_dst + offset]) = reg2

   initialize memory with a value stored in register

   Conditions on instruction operands

   $16 \leq$ reg_dst ≤ 23

   $0 \leq$ reg_src ≤ 23

31 26	25 21	20 16	15 0
100010	reg_dst	reg_src	Immediate

9. **Lui** (init format)

   reg_dst <= (imm_16«16) | reg_src;

To ease initialize of 32 bit values, we can use this instruction to load upper 16 bits into a register with an immediate value

Conditions on instruction operands

$16 \leq$ reg_dst ≤ 23

$16 \leq$ reg_src ≤ 23

31 26	25 21	20 16	15 0
101011	reg_dst	reg_src	Immediate

The tag update instructions allow to update tags of registers or memory addresses.

1. **TRR** (TR format)
   tag(reg_dst) = tag (reg_src1) OPERATION tag(reg_src2)
   update tag of reg_dst_arm with reg_src1_arm and reg_src2_arm
   Conditions on instruction operands
   $0 \leq$ reg_dst ≤ 15
   $0 \leq$ reg_src ≤ 15

31 26	25 22	21 17	16 12	11 7	6 5	4 0
100011	ARM_OPCODE_TYPE	reg_dst	reg_src1	reg_src2	n.u	n.u

2. **Tag_mem_tr** (TI format)
   tag (Mem(reg_dst + offset)) = tag(reg_src1)
   update tag of memory address contained in reg_dst with tag value contained in reg_src1 (ARM STR operation)
   propagate operation specified by security policy
   Conditions on instruction operands
   $16 \leq$ Reg_dst ≤ 23
   $0 \leq$ reg_src1 ≤ 15

31 26	25 22	21 17	16 12	11 0
100100	ARM_OPCODE_TYPE	reg_dst	reg_src1	Immediate

3. **Tag_tr_mem** (TI format)
   tag(reg_dst) = tag (Mem(reg_src + offset))
   update tag of reg_dst with the tag of memory address contained in (val(reg_src) + offset) (this operation is done for ARM LDR operation)
   propagate operation specified by security policy
   Conditions on instruction operands
   $0 \leq$ Reg_dst ≤ 15
   $16 \leq$ reg_src1 ≤ 23

31 26	25 22	21 17	16 12	11 0
100101	ARM_OPCODE_TYPE	reg_dst	reg_src1	Immediate

4. **Tag_mem_tr_2** (TI format)

tag (Mem(reg_dst + offset)) = tag(reg_src1)
update tag of memory address contained in reg_dst with tag value contained in reg_src1 (ARM STR operation)
propagate operation specified by opcode
Conditions on instruction operands
16 ≤ Reg_dst ≤ 23
0 ≤ reg_src1 ≤ 15

31 26	25 22	21 17	16 12	11 0
101100	ARM_OPCODE_TYPE	reg_dst	reg_src1	Immediate

5. **Tag_tr_mem_2** (TI format)
tag(reg_dst) = tag (Mem(reg_src + offset))
update tag of reg_dst with the tag of memory address contained in (val(reg_src) + offset) (this operation is done for ARM LDR operation)
propagate operation specified by opcode
Conditions on instruction operands
0 ≤ Reg_dst ≤ 15
16 ≤ reg_src1 ≤ 23

31 26	25 22	21 17	16 12	11 0
101101	ARM_OPCODE_TYPE	reg_dst	reg_src1	Immediate

6. **Tag_arith_log** (TR format)
General purpose operations on GPR of DIFT coprocessor (2^{nd} stage) (add/or/and/-sub/sll/srl ...)
Conditions on instruction operands
16 ≤ reg_dst ≤ 23
16 ≤ reg_src1 ≤ 23
16 ≤ reg_src2 ≤ 23

31 26	25 22	21 17	16 12	11 7	6 5	4 0
100110	ARM_OPCODE_TYPE	reg_dst	reg_src1	reg_src2	n.u	FUNCT

7. **Tag_instrumentation_tr** (TI format)
tag(Mem(instrumentation)) <= tag (reg_src1) (operation done for ARM STR register relative instructions)
Condition(s) on instruction operands
0 ≤ reg_src1 ≤ 15

31 26	25 22	21 17	16 12	11 0
100111	ARM_OPCODE_TYPE	reg_dst	reg_src1	Immediate

8. **Tag_tr_instrumentation** (TI)

reg_dst <= tag(Mem(instrumentation)) (opreation done for ARM register relative LDR instructions)

Update tag of memory address, obtained by instrumentation IP, with the tag of ARM register (reg_src1)

Condition(s) on instruction operands

$0 \leq$ reg_dst ≤ 23

31 26	25 22	21 17	16 12	11 0
101000	ARM_OPCODE_TYPE	reg_dst	reg_src1	Immediate

9. **Tag_Kblare_tr** (TI format)

   Mem(KBlare) <= tag (reg_src1)

   write tag to Kblare IP PL2PS

   Condition(s) on instruction operands

   $0 \leq$ reg_src1 ≤ 15

31 26	25 22	21 17	16 12	11 0
101001	ARM_OPCODE_TYPE	reg_dst	reg_src1	Immediate

10. **Tag_tr_Kblare** (TI format)

    tag (reg_dest) <= Mem(KBlare)

    read tag from Kblare IP PS2PL

    Condition(s) on instruction operands

    $0 \leq$ reg_src1 ≤ 15

31 26	25 22	21 17	16 12	11 0
101010	ARM_OPCODE_TYPE	reg_dst	reg_src1	Immediate

Appendix D

Publications

International conferences

[C1] Muhammad Abdul Wahab et al. "A MIPS-based coprocessor for information flow tracking in ARM SoCs". *Reconfig*. Accepted. 2018, p. 8.

[C2] Muhammad Abdul Wahab et al. "A novel lightweight hardware-assisted static instrumentation approach for ARM SoC using debug components". *AsianHOST*. accepted. 2018, p. 6.

[C3] Muhammad Abdul Wahab et al. "ARMHEx: A hardware extension for DIFT on ARM-based SoCs". *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*. 2017, pp. 1–7. DOI: 10.23919/FPL.2017.8056767. URL: https://doi.org/10.23919/FPL.2017.8056767.

[C4] Muhammad Abdul Wahab et al. "Towards a hardware-assisted information flow tracking ecosystem for ARM processors". *26th International Conference on Field-Programmable Logic and Applications (FPL 2016)*. Lausanne, Switzerland, Aug. 2016. URL: https://hal.archives-ouvertes.fr/hal-01337579.

Demonstration

[D1] Muhammad Abdul Wahab et al. "ARMHEx: A framework for efficient DIFT in real-world SoCs". *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*. 2017, p. 1. DOI: 10.23919/FPL.2017.8056799. URL: https://doi.org/10.23919/FPL.2017.8056799.

Talks without proceedings

[T1] Pascal Cotret and Muhammad Abdul Wahab. "TrustZone is not enough - Hijacking debug components for embedded security". *Chaos Communication Congress*. Dec. 2017. URL: https://goo.gl/AAmHSZ.

[T2] Muhammad Abdul Wahab and Pascal Cotret. "Pwning ARM Debug Components for Sec-Related Stuff." *HITBSecConf CommSec track 2017 (Hack In The Box Security Conference)*. 2017. URL: https://goo.gl/SSnnLo.

[T3] Muhammad Abdul Wahab, Christophe Moy, and Pascal Cotret. "A hardware coprocessor for Zynq-based Dynamic Information Flow Tracking". *14th International Workshop on Cryptographic architectures embedded in logic devices (CryptArchi)*. La Grande Motte, France, June 2016. URL: https://hal-centralesupelec.archives-ouvertes.fr/hal-01341830.

National conferences

[N1] Muhammad Abdul Wahab et al. "Monitoring information flows in heterogeneous SoCs with a dedicated coprocessor". *13ème Colloque du GDR SoC2*. Paris, France, June 2018.

[N2] Muhammad Abdul Wahab et al. "ARMHEx: a hardware extension for information flow tracking on ARM-based platforms". *12ème Colloque du GDR SoC2*. Bordeaux, France, June 2017.

[N3] Muhammad Abdul Wahab et al. "ARMHEx: a hardware extension for information flow tracking on ARM-based platforms". *Les Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI 2017)*. Grenoble, France, May 2017. URL: https://hal.archives-ouvertes.fr/hal-01558155.

[N4] Muhammad Abdul Wahab et al. "A portable approach for SoC-based Dynamic Information Flow Tracking implementations". *11ème Colloque du GDR SoC/SiP*. Nantes, France, June 2016. URL: https://hal-centralesupelec.archives-ouvertes.fr/hal-01311045.

[N5] Mounir Nasr Allah et al. "HardBlare: a Hardware-Assisted Approach for Dynamic Information Flow Tracking". *Séminaire des doctorantes et doctorants en informatique de la Société Informatique de France*. Poster. Apr. 2016. URL: https://hal-centralesupelec.archives-ouvertes.fr/hal-01311032.

Bibliography

[1] Michael Dalton, Hari Kannan, and Christos Kozyrakis. "Raksha: A Flexible Information Flow Architecture for Software Security". *SIGARCH Comput. Archit. News* 35.2 (June 2007), pp. 482–493. ISSN: 0163-5964.

[2] H. Kannan, M. Dalton, and C. Kozyrakis. "Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor". *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. June 2009. DOI: 10.1109/DSN.2009.5270347.

[3] Daniel Y. Deng et al. "Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric". *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '43. 2010. ISBN: 978-0-7695-4299-7.

[4] Ingoo Heo et al. "Implementing an Application-Specific Instruction-Set Processor for System-Level Dynamic Program Analysis Engines". *ACM Trans. Des. Autom. Electron. Syst.* 20.4 (Sept. 2015), 53:1–53:32. ISSN: 1084-4309.

[5] Jinyong Lee et al. "Efficient Security Monitoring with the Core Debug Interface in an Embedded Processor". *ACM Trans. Des. Autom. Electron. Syst.* 22.1 (May 2016), 8:1–8:29. ISSN: 1084-4309.

[6] Laurent Georget et al. "Information Flow Tracking for Linux Handling Concurrent System Calls and Shared Memory". *SEFM*. Vol. 10469. Lecture Notes in Computer Science. Springer, 2017, pp. 1–16.

[7] Nguyen Anh Quynh. *Capstone - the ultimate disassembler*. [Online; accessed 28-September-2018]. 2013.

[8] MITRE corporation. *CVE Details - The ultimate security vulnerability datasource*. https://www.cvedetails.com. [Online; accessed 08-October-2018]. 2017.

[9] Inc. SPARC Internaltional. *The SPARC Architecture Manual, Version 8*. https://goo.gl/WWc31v. [Online; accessed 26-September-2018]. 2018.

[10] Sarah Murry. *ARM's Reach: 50 Billion Chip Milestone*. https://goo.gl/7K8SAk. [Online; accessed 28-September-2018]. 2018.

[11] HardBlare project Team. *Presentation of the HardBlare project*. https://hardblare.cominlabs.u-bretagneloire.fr/. [Online; accessed 28-September-2018]. 2018.

[12] Laurent George et al. "Blare Tools: A Policy-Based Intrusion Detection System Automatically Set by the Security Policy". *RAID 2009*. 2009. DOI: 10.1007/978-3-642-04342-0_22.

[13] Stephane Geller et al. “Information Flow Control for Intrusion Detection Derived from MAC Policy”. *Proceedings of IEEE International Conference on Communications, ICC 2011, Kyoto, Japan, 5-9 June, 2011*. 2011, pp. 1–6. DOI: 10.1109/icc.2011.5962660. URL: https://doi.org/10.1109/icc.2011.5962660.

[14] Christophe Hauser et al. “A taint marking approach to confidentiality violation detection”. *Tenth Australasian Information Security Conference, AISC 2012, Melbourne, Australia, January 2012*. 2012, pp. 83–90. URL: http://crpit.com/abstracts/CRPITV125Hauser.html.

[15] *Micro-Policies: Formally Verified, Tag-Based Security Monitors*. Zenodo, May 2015. DOI: 10.1109/sp.2015.55. URL: https://doi.org/10.1109/sp.2015.55.

[16] Jelena Milosevic. “Malware Detection at Runtime for Resource-Constrained Mobile Devices: Data-Driven Approach”. Theses. Università della Svizzera Italiana, Oct. 2017.

[17] Muhammad Abdul Wahab. *Device tree patch - Commits links*. https://pastebin.com/JWv7c6P5. [Online; accessed 30-July-2018]. 2018.

[18] W. Hu et al. “^{2} Gate-Level Information-Flow Tracking”. *Computer* 49.8 (Aug. 2016), pp. 44–52. ISSN: 0018-9162. DOI: 10.1109/MC.2016.225.

[19] *Introduction to Perl’s Taint Mode*. [Online; accessed 30-July-2018]. URL: http://webreference.com/programming/perl/taint/index-2.html.

[20] Andrew C. Myers. “JFlow: Practical Mostly-static Information Flow Control”. *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’99. San Antonio, Texas, USA: ACM, 1999, pp. 228–241. ISBN: 1-58113-095-3. DOI: 10.1145/292540.292561. URL: http://doi.acm.org/10.1145/292540.292561.

[21] Andrew C. Myers and Barbara Liskov. “Protecting Privacy Using the Decentralized Label Model”. *ACM Trans. Softw. Eng. Methodol.* 9.4 (Oct. 2000), pp. 410–442. ISSN: 1049-331X. DOI: 10.1145/363516.363526. URL: http://doi.acm.org/10.1145/363516.363526.

[22] Guillaume Hiet. “Détection d’intrusions paramétrée par la politique de sécurité grâce au contrôle collaboratif des flux d’informations au sein du système d’exploitation et des applications : mise en œuvre sous Linux pour les programmes Java”. Version non-définitive - présentée le jour de la soutenance. Theses. Université Rennes 1, Dec. 2008. URL: https://tel.archives-ouvertes.fr/tel-00355089.

[23] William Enck et al. “TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones”. *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 393–407. URL: http://dl.acm.org/citation.cfm?id=1924943.1924971.

[24] Radoniaina Andriatsimandefitra and Valérie Viet Triem Tong. “Capturing Android Malware Behaviour using System Flow Graph”. *NSS 2014 - The 8th International Conference on Network and System Security*. Xi’an, China, Oct. 2014. URL: https://hal.inria.fr/hal-01018611.

[25] Andrey Chudnov and David A. Naumann. "Inlined Information Flow Monitoring for JavaScript". *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 629–643. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813684. URL: http://doi.acm.org/10.1145/2810103.2813684.

[26] Vincent Simonet and Inria Rocquencourt. *Flow Caml in a Nutshell*. 2003.

[27] Nickolai Zeldovich. "Securing Untrustworthy Software Using Information Flow Control". AAI3292438. PhD thesis. Stanford, CA, USA, 2008. ISBN: 978-0-549-35732-2.

[28] Petros Efstathopoulos et al. "Labels and Event Processes in the Asbestos Operating System". *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: ACM, 2005, pp. 17–30. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095813. URL: http://doi.acm.org/10.1145/1095810.1095813.

[29] Maxwell Krohn et al. "Information Flow Control for Standard OS Abstractions". *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 321–334. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294293. URL: http://doi.acm.org/10.1145/1294261.1294293.

[30] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. "Introducing Reference Flow Control for Detecting Intrusion Symptoms at the OS Level". *Recent Advances in Intrusion Detection*. Ed. by Andreas Wespi, Giovanni Vigna, and Luca Deri. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 292–306. ISBN: 978-3-540-36084-1.

[31] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034. URL: http://doi.acm.org/10.1145/1065010.1065034.

[32] Cheng Wang et al. "StarDBT: An Efficient Multi-platform Dynamic Binary Translation System". *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture*. ACSAC'07. Seoul, Korea: Springer-Verlag, 2007, pp. 4–15. ISBN: 3-540-74308-1, 978-3-540-74308-8. URL: http://dl.acm.org/citation.cfm?id=2392163.2392166.

[33] Derek L. Bruening. "Efficient, Transparent, and Comprehensive Runtime Code Manipulation". AAI0807735. PhD thesis. Cambridge, MA, USA, 2004.

[34] Emmanuel Riou et al. "PADRONE: a Platform for Online Profiling, Analysis, and Optimization". *International Workshop on Dynamic Compilation Everywhere*. DCE'14. 2014.

[35] Ryan Whelan, Tim Leek, and David Kaeli. "Architecture-independent dynamic information flow tracking". *International Conference on Compiler Construction*. Springer. 2013, pp. 144–163.

[36] *QEMU, the FAST processor emulator*. https://www.qemu.org/. [Online; accessed 26-September-2018]. 2018.

[37] W. Hu et al. “Theoretical Fundamentals of Gate Level Information Flow Tracking”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.8 (Aug. 2011), pp. 1128–1140. ISSN: 0278-0070. DOI: 10.1109/TCAD.2011.2120970.

[38] André DeHon et al. “Preliminary design of the SAFE platform”. *PLOS@SOSP*. ACM, 2011, 4:1–4:5.

[39] Jonathan Woodruff et al. “The CHERI capability model: Revisiting RISC in an age of risk”. *ISCA*. IEEE Computer Society, 2014, pp. 457–468.

[40] Udit Dhawan et al. “Architectural Support for Software-Defined Metadata Processing”. *SIGARCH Comput. Archit. News* 43.1 (Mar. 2015), pp. 487–502. ISSN: 0163-5964. DOI: 10.1145/2786763.2694383. URL: http://doi.acm.org/10.1145/2786763.2694383.

[41] Shimin Chen et al. “Flexible Hardware Acceleration for Instruction-Grain Program Monitoring”. *SIGARCH Comput. Archit. News* (June 2008). ISSN: 0163-5964. DOI: 10.1145/1394608.1382153.

[42] Olatunji Ruwase et al. “Parallelizing Dynamic Information Flow Tracking”. *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA ’08. Munich, Germany: ACM, 2008, pp. 35–45. ISBN: 978-1-59593-973-9.

[43] Daniel Y. Deng and G. Edward Suh. “High-performance Parallel Accelerator for Flexible and Efficient Run-time Monitoring”. *Proceedings of the 2012 42Nd Annual IEEE/IFIP DSN*. DSN ’12. 2012, pp. 1–12. ISBN: 978-1-4673-1624-8.

[44] Jedidiah R. Crandall and Frederic T. Chong. “Minos: Control Data Attack Prevention Orthogonal to Memory Model”. *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 37. Portland, Oregon: IEEE Computer Society, 2004, pp. 221–232. ISBN: 0-7695-2126-6. DOI: 10.1109/MICRO.2004.26. URL: http://dx.doi.org/10.1109/MICRO.2004.26.

[45] Neil Vachharajani et al. “RIFLE: An Architectural Framework for User-Centric Information-Flow Security”. *37th International Symposium on Microarchitecture (MICRO-37’04)* (2004), pp. 243–254.

[46] Feng Qin et al. “LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks”. *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’06)* (2006), pp. 135–148.

[47] Joël Porquet and Simha Sethumadhavan. “WHISK: An Uncore Architecture for Dynamic Information Flow Tracking in Heterogeneous Embedded SoCs”. *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS ’13. Montreal, Quebec, Canada: IEEE Press, 2013, 4:1–4:9. ISBN: 978-1-4799-1417-3. URL: http://dl.acm.org/citation.cfm?id=2555692.2555696.

[48] Juan Carlos Martinez Santos, Yunsi Fei, and Zhijie Jerry Shi. “PIFT: Efficient Dynamic Information Flow Tracking Using Secure Page Allocation”. *Proceedings of the 4th Workshop on Embedded Systems Security*. WESS ’09. Grenoble, France: ACM, 2009, 6:1–6:8. ISBN: 978-1-60558-700-4. DOI: 10.1145/1631716.1631722. URL: http://doi.acm.org/10.1145/1631716.1631722.

[49] Antonio M. Espinoza et al. "V-DIFT: Vector-Based Dynamic Information Flow Tracking with Application to Locating Cryptographic Keys for Reverse Engineering". *ARES*. IEEE Computer Society, 2016, pp. 266–271.

[50] James Clause, Wanchun Li, and Alessandro Orso. "Dytan: A Generic Dynamic Taint Analysis Framework". *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA '07. London, United Kingdom: ACM, 2007, pp. 196–206. ISBN: 978-1-59593-734-6. DOI: 10.1145/1273463.1273490. URL: http://doi.acm.org/10.1145/1273463.1273490.

[51] Jangseop Shin et al. "A hardware-based technique for efficient implicit information flow tracking". *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2016, pp. 1–7. DOI: 10.1145/2966986.2966991.

[52] N. Roessler and A. DeHon. "Protecting the Stack with Metadata Policies and Tagged Hardware". *2018 IEEE Symposium on Security and Privacy (SP)*. Vol. 00. 2018, pp. 1072–1089. DOI: 10.1109/SP.2018.00066. URL: doi.ieeecomputersociety.org/10.1109/SP.2018.00066.

[53] *ARM Holdings Q1 2017*. URL: https://www.arm.com/company/investors/-/media/arm-com/company/Investors/Quarterly%20Results%20-%20PDFs/Arm_SB_Q1_2017_Roadshow_Slides_Final.pdf?la=en.

[54] L. Davi et al. "HAFIX: Hardware-Assisted Flow Integrity eXtension". *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. DOI: 10.1145/2744769.2744847.

[55] Xilinx. *Zynq-7000 All Programmable SoC - Technical reference manual*. 2015.

[56] ARM. *CoreSight Program Flow Trace - Architecture specifications*. 2011.

[57] ARM. *CoreSight Components - Technical Reference manual*. 2009.

[58] *Spedi disassembler*. [Online; accessed 30-July-2018]. URL: https://www/github.com/abenkhadra/spedi.

[59] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[60] Adrien Vergé. "Hardware-assisted software tracing". *Embedded Linux Conference (ELC)*. Apr. 2014.

[61] Muhammad Abdul Wahab et al. "Towards a hardware-assisted information flow tracking ecosystem for ARM processors". *26th International Conference on Field-Programmable Logic and Applications (FPL 2016)*. Lausanne, Switzerland, Aug. 2016. URL: https://hal.archives-ouvertes.fr/hal-01337579.

[62] Muhammad Abdul Wahab et al. "ARMHEx: A hardware extension for DIFT on ARM-based SoCs". *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*. 2017, pp. 1–7. DOI: 10.23919/FPL.2017.8056767. URL: https://doi.org/10.23919/FPL.2017.8056767.

[63] ARM. *ARM security technology - Building a secure system using TrustZone technology*. 2009. URL: https://www.sbs.ox.ac.uk/cybersecurity-capacity/system/files/ARM%20Security%20Technology.pdf.

[64] Akshay Sudhir Vaidya. "Strengthening Zynq7000 Device Security using ARM TrustZone". MA thesis. Albuquerque, New Mexico: The University of New Mexico, 2015.

[65] Johannes Winter et al. "A Flexible Software Development and Emulation Framework for ARM TrustZone". *Trusted Systems*. Ed. by Liqun Chen, Moti Yung, and Liehuang Zhu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–15. ISBN: 978-3-642-32298-3.

[66] *SafeG dual-OS monitor*. [Online; accessed 30-July-2018]. URL: http://www.toppers.jp/en/safeg.html.

[67] *TOPPERS/FMP Kernel*. [Online; accessed 30-July-2018]. URL: http://www.toppers.jp/en/fmp-kernel.html.

[68] Olle Svanfeldt-Winter, Lafond Sébastien, and Lilius Johan. *Evaluation of the Energy Efficiency of ARM Based Processors for Cloud Infrastructure*. Tech. rep. Turku Centre for Computer Science.

[69] Muhammad Abdul Wahab et al. "ARMHEx: A framework for efficient DIFT in real-world SoCs". *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*. 2017, p. 1. DOI: 10.23919/FPL.2017.8056799. URL: https://doi.org/10.23919/FPL.2017.8056799.

[70] Laurent Georget et al. "Information Flow Tracking for Linux Handling Concurrent System Calls and Shared Memory". *15th International Conference on Software Engineering and Formal Methods (SEFM 2017)*. Ed. by Alessandro Cimatti and Marjan Sirjani. LNCS. Springer International Publishing, Sept. 2017. DOI: 10.1007/978-3-319-66197-1_1. URL: http://hal.upmc.fr/hal-01535949.

[71] GNU. *GCC MIPS cross compiler*. https://goo.gl/E3yThX. [Online; accessed 25-September-2018]. 2018.

[72] ARM. *CoreSight Technology System Design Guide*.

[73] Forbes Aaron Tilley. *This Mysterious Chip In The iPhone 7 Could Be Key To Apple's AI Push*. https://goo.gl/Woxmko. [Online; accessed 25-September-2018]. 2018.

[74] Xilinx. *Creating and Packaging Custom IP (UG1119)*. https://goo.gl/Q2DQA9. [Online; accessed 20-September-2018]. 2018.

[75] Xilinx. *OS and Libraries Documentation Collection*. [Online; accessed 29-September-2018]. 2015.

[76] *Yocto project*. [Online; accessed 30-July-2018]. URL: https://www.yoctoproject.org/.

[77] Muhammad Abdul Wahab. *HardBlare code and patches*. https://abdulparis@bitbucket.org/pcmaw/code-and-patches.git. [Online; accessed 08-October-2018]. 2018.

[78] Android open source project Arve Hjønnevåg. *Decoding the PTM traces*. https://goo.gl/8LFuHL. [Online; accessed 08-October-2018]. 2012.

[79] Arnaldo Carvalho De Melo. "The new linux'perf'tools". 2010.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

CentraleSupélec

Titre : Support matériel pour l'analyse de sécurité du comportement des applications : application au contrôle de flux d'information et à l'analyse des logiciels malveillants

Mots clés : FPGA, sécurité, systèmes embarqués, contrôle de flux d'information, architectures hétérogènes

Résumé : Le contrôle de flux d'informations, Dynamic Information Flow Tracking (DIFT), permet de détecter différents types d'attaques logicielles tels que les dépassements de tampon ou les injections SQL.
Dans cette thèse, une solution ciblant les processeurs hardcore ARM Cortex-A9 est proposée. Notre approche s'appuie sur des composants ARM CoreSight, qui permettent de tracer l'exécution des programmes exécutés par le processeur, afin de réaliser le contrôle de flux d'informations.
Le co-processeur DIFT que nous proposons est réalisé dans la partie FPGA Artix-7 du système sur puce (SoC) Zynq. Il est montré que l'utilisation des composants ARM CoreSight n'ajoute pas de surcoût en terme de temps d'exécution et permet une amélioration du temps de communication entre le processeur ARM et le coprocesseur DIFT.

Title : Hardware support for the security analysis of embedded softwares : applications on information flow control and malware analysis

Keywords : FPGA, security, embedded systems, information flow control, heterogeneous architectures

Abstract : Information flow control (also known as Dynamic Information Flow Tracking, DIFT), allow a user to detect several types of software attacks such as buffer overflow or SQL injections.
In this thesis, a solution based on the ARM Cortex-A9 processor family is proposed. Our approach relies on the use of ARM CoreSight components, which are able to trace softwares as executed by the processor in order to perform the information flow tracking.
The DIFT coprocessor proposed in this thesis is implemented in an Artix-7 FPGA, embedded in a System-on-Chip (SoC) Zynq provided by Xilinx. It is shown that using ARM CoreSight components does not add a latency overhead while giving a better communication time between the ARM processor and the DIFT coprocessor.