Vector Instruction Set Extensions for Efficient and Reliable Computation of Keccak

Hemendra K. Rawat

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Patrick Schaumont, Chair

Leyla Nazhandali

Paul Plassmann

August 10, 2016

Blacksburg, Virginia

Keywords: SIMD, Instruction Set Extensions, SHA-3, Hashing, Authenticated Encryption,

Software Integrity

Copyright 2016, Hemendra K. Rawat

Vector Instruction Set Extensions for Efficient and Reliable Computation of Keccak

Hemendra K. Rawat

(ABSTRACT)

Recent processor architectures such as Intel Westmere (and later) and ARMv8 include instruction-level support for the Advanced Encryption Standard (AES), for the Secure Hashing Standard (SHA-1, SHA2) and for carry-less multiplication. These crypto-instructions are optimized for a single algorithm and provide significant performance improvements over software written using general-purpose instruction set. However, today's secure systems and protocols do not rely on just one, but a suite of many cryptographic applications that are expected to work in a correct and reliable manner. In this work, we propose a new instruction set for supporting efficient and reliable cryptography on modern processors. For efficiency, we propose flexible instruction set extensions for KECCAK, a cryptographic kernel for hashing, authenticated encryption, key-stream generation and random-number generation. KECCAK is the basis of the SHA-3 standard and the newly proposed KEYAK and KETJE authenticated ciphers. For reliability, we propose a set of trusted instructions to verify the integrity of a cryptographic software library. These instructions are aimed at detecting tamper in the software or in the configurable hardware. We develop the instruction extensions for a 128-bit interface, commonly available in the vector processing unit of many modern processors. Simulation results on GEM5 architectural simulator show that the pro-

posed instructions not only improves the performance of Keccak applications by 2 times (over NEON programming) and 6 times (over assembly programming), but also improves the reliability of applications at a performance overhead of just 6%.

This research was supported in part through the National Science Foundation Grant 1441710, and in part through the Semiconductor Research Corporation.

Dedication

In dedication to my family. Mummy, Papa and Didi.

The reason for what I am today.

Acknowledgments

First, I would like to sincerely thank my advisor, Dr. Patrick Schaumont. It has been a privilege to work under his guidance and I am extremely grateful for all his support and motivation during the conception, design, development and publication of this work. Also, I would like to thank Dr. Leyla Nazhandali and Dr. Paul Plassmann for sparing their time to serve on my thesis committee.

I would also like to use this opportunity to express my gratitude towards my family, who have always supported and encouraged me to follow my dreams. Thank you Mummy, Papa and Didi for all the love and support you provided throughout this journey. My friend Sharath, Ameya and Utkarsh, deserve a special note of thanks here. Thank you for all the feedback and motivation that you guys provided, whenever I was in need of it. Life here in Blacksburg would have been boring without you guys.

I sincerely appreciate the help from all my lab mates, Bilgiday, Chinmay, Carol, Harsha, Shravya, Nahid, Conor and Aydin. It was a wonderful experience working with you all.

Contents

1 Introduction **1**

1.1 Introduction 1

1.1.1 Cryptography using Custom Instructions 3

1.2 Contributions 5

1.3 Organization 6

1.4 Related Articles 7

2 Keccak Instruction Set **8**

2.1 Cryptographic Sponge Functions 8

2.1.1 Keccak-f and Keccak-p Permutations 10

2.2 Related Work 12

2.3 Design Exploration for Keccak Instructions 12

2.3.1 Cutting the KECCAK State 14

2.3.2 ARMv7 ISA and NEON SIMD 18

2.4 Proposed Instruction Set Extensions for KECCAK 18

2.4.1 Instruction `rl1x` 19

2.4.2 Instruction `kxorr64` 20

2.4.3 Instruction `xorr` 21

2.4.4 Instruction `chi1` 22

2.4.5 Instruction `chi2` 22

2.4.6 Instruction `chi3` 23

2.5 Results 25

2.5.1 Simulation Setup 25

2.5.2 Performance 26

2.5.3 Hardware Cost 28

2.6 Portability Aspects 29

2.6.1 Intel AVX (128 bit SIMD) 30

2.6.2 64-bit Architectures 31

2.6.3 32-bit Architectures 32

3 Integrity of Cryptographic Implementations **33**

3.1 Introduction 33

3.1.1 Need for Integrity Verification 34

3.1.2 Threat Model and Root of Trust 35

3.2 Proposed Methodology for Integrity Verification 37

3.3 Trusted Instructions for Integrity Verification 39

3.3.1 Instruction `mtrace` 39

3.3.2 Instruction `ctrace` 39

3.3.3 Instruction `mconfig` 40

3.4 High Level Design 44

3.4.1 Monitor Unit 45

3.4.2 Parity Prediction Unit 46

3.5 Protecting Crypto using Trace Monitoring 48

3.5.1 Straight line code 48

3.5.2 Handling Data Dependent Branches 49

3.5.3 Handling Intra Procedural Control Flow 50

3.6 Results 52

3.6.1 Simulation Setup 52
3.6.2 Attack Detection 52
3.6.3 Performance Overhead 53
3.7 Portability and Future work 55

4 Side Channel and Fault Attacks **56**

5 Conclusion **57**

Bibliography **59**

List of Figures

2.1 Constructions with sponge: (a) Hash, (b) Message Authentication Code, (c) Keystream Generation. Construction with duplex: (d) Authenticated Encryption . . . 9

2.2 KECCAK application stack . . . 13

2.3 Data dependencies of θ-effect values . . . 14

2.4 Data dependencies of θ, ρ, π and χ step (one plane) . . . 16

2.5 Aliased NEON registers in ARMv7 . . . 17

2.6 a) Register allocation for KECCAK-$f[1600]$ and KECCAK-$f[800, 400, 200]$ . . . 19

2.7 a) Functionality of `rl1x` instruction, b) Functionality of `kxorr64, xorr` instruction . . . 20

2.8 Functionality of instruction a) `chi1`, b) `chi2` and c) `chi3` . . . 24

3.1 Block Diagram of an extensible RISC Processor . . . 34

3.2 Proposed methodology for protecting cryptographic kernels 37

3.3 Measurements used for hash input 38

3.4 Block diagram of an Extensible RISC CPU with tracing support 44

3.5 High Level Design of the Monitor Unit 45

3.6 High Level Design of the Parity Prediction Unit 47

3.7 Protecting Linear code 49

3.8 Handling data dependent branches 50

3.9 Handling Intra-Procedural Control flow 51

3.10 Performance overhead of the proposed method for SHA3-512 54

List of Tables

2.1 Performance in instructions/byte for various KECCAK modes 27

2.2 Instructions executed per KECCAK round 28

2.3 Area, GE and Transistor count estimates for the proposed custom instructions 29

2.4 Feasibility of the proposed instructions on different platforms 30

3.1 `mtrace` commands 40

3.2 `ctrace` commands 40

3.3 `mconfig` commands 43

3.4 Parity Prediction Equations for KECCAK-f [1600] Custom Instructions . . . 46

3.5 Tamper detection in SHA3-512 application 53

Chapter 1

Introduction

1.1 Introduction

In 2012, NIST announced Keccak as the winner of the SHA-3 cryptographic hashing standard competition [16]. The cryptographic permutation of Keccak can be used in a sponge construction to support hashing, MACing [6], encryption/decryption [3], and random number generation [4]. Keccak can also be used in a duplex construction to support authenticated encryption [5]. Each of these applications are achieved through only minor variations of the Keccak sponge or duplex.

On a standard block cipher, this type of multi-functional behavior can only be achieved through modes of operation (MOO), which can introduce additional complexity in the form of feedback, data dependencies, operations, and intermediate storage. The combination of

an AES-128 block cipher (with only 128 bits of state) with multiple modes of operation can result in a larger and slower design than a multi-purpose KECCAK-f[1600] permutation (with 1600 bits of state). Yalla *et al.* compared multi-function hardware designs based on AES and KECCAK for Virtex-7 FPGA. They conclude that an AES-based multi-functional design is not only 21% larger than a similar KECCAK-based design, but that it is also slower. The throughput of the AES-based design is three times – for hashing – to nine times – for authenticated encryption – slower [29].

Recent processor architectures such as Intel (Westmere,Sandybridge, Ivybridge and Haswell) and ARMv8 offer dedicated instructions to compute AES, SHA-1, SHA-256 and carry-less multiplication. These instructions operate on wide registers: 128 bit multi-media extensions (XMM) in the case of Intel and 128 bit NEON in the case of ARM. Their appearance in mainstream processors is motivated by the increasing proportion of cryptographic processing in the contemporary processor workload. The Crypto-instructions are used to efficiently support secure connections (IPSec, SSL and TLS), bulk encryption and new applications such as blockchains.

In our research, we are motivated by the potential of new instructions to support efficient and reliable cryptography in modern processors. In contrast to the contemporary crypto-instructions (such as AES-NI and SHA-1/SHA-2 for Intel Processors [17], [18]) that are optimized towards accelerating a single task, we try to find an optimal balance between flexibility, performance and reliability. This is important as the secure systems and protocols of today not only require high performance and energy efficiency, but they are also built

on top of multiple cryptographic kernels that support multiple security levels and modes of operation. The reliability of these cryptographic kernels is also a concern these days. As their security properties assume correct execution of the kernel, but in practice software has bugs, firmware can be maliciously updated and hardware may occasionally have faults. As a first step towards this goal, we propose a flexible instruction set for Keccak that can accelerate Keccak kernels of several state sizes (of 1600 bit, 800 bit, and smaller). This enables these custom instructions to implement multiple cryptographic algorithms based on Keccak, such as hash, stream ciphers, authenticated encryption and many more. Next, we built an integrity check mechanism based on trusted custom instructions to protect cryptographic applications against software and hardware tampering.

The results show that cryptographic algorithms developed using proposed custom instruction are not only 2 times (over NEON programming) to 6 times (over Assembly programming) faster compared to the latest, hand-optimized implementations, but they are also able to detect tampering in the software or custom hardware that affects their control and data flow integrity.

1.1.1 Cryptography using Custom Instructions

In this section, we justify the strategy of integrating cryptographic hardware as vector instruction-extensions in a processor. It is commonly assumed that the addition of new instructions to the instruction-set of a processor is a risky and complex undertaking. Not only

does it require a thorough understanding of the microprocessor micro-architecture, it also impacts the processor's software tool-chain, including the compiler and assembler. However, there are several strong arguments in favor of the proposed custom-instruction methodology, and they are clarified as follows.

- First, the proposed instruction extensions are specialized, and we do not require extensive compiler support for them such as low-level code-generation out of a high-level language. Instead, support at the assembly level is adequate. This limits the complexity of the required software infrastructure.

- Second, there is a trend towards open processor architectures, which are more amenable to tweaking. In this spirit, the academic community is developing the RISC-V processor [31]. Even commercial micro-architectures are no longer completely closed. Using a proper licensing scheme, ARM processors can be modified before integration into a System-on-chip, and silicon system houses are using this capability as a feature differentiator. Although we advocate for our vector instruction set as a standard feature rather than a feature differentiator, it is clear that contemporary processor design flows support such customization.

- Third, for highly complex system-on-chip design, the use of instruction extensions is preferable compared to the use of memory-mapped acceleration units. Memory-mapped acceleration units face increasing latency because of complex on-chip communication structures, such as multi-level buses and share communication pathways.

Moreover, the management of sensitive cryptographic state over distributed, memory-mapped acceleration units in an SoC is an additional risk, when we consider that the memory space is a shared resource.

- Fourth, vector-instructions operate at the apex of the memory-hierarchy. Provided that the cryptographic state of an algorithm can be kept inside of the processor registers, vector-instructions will eliminate the effects of memory latency. When crypto-kernels iterate over the cryptographic state for several rounds, this performance benefit becomes significant because the full processor speed is available for the entire duration of the kernel execution.

1.2 Contributions

The contributions of this work are as follows:

- We present an implementation of the KECCAK-*f/p* permutation as custom instructions for a vector processing unit. The proposed instructions support sponge and duplex constructions with KECCAK in four different state sizes (200, 400, 800 and 1600 bits). The design is designed for a (128-bit x 128-bit → 128-bit) ARM NEON interface. We discuss portability to other architectures and other bit-widths in a separate section.

- We demonstrate the proposed instructions by implementing a collection of five cryptographic algorithms based on KECCAK, including the SHA3-512 hash, the authenti-

cated ciphers Lake Keyak and River Keyak, and the authenticated ciphers Ketje SR and Ketje JR.

- We evaluate the performance of the resulting designs in the context of an ARMv7 micro-architecture. We use the GEM5 architectural simulator, which provides binary compatibility with the ARM ABI, but not the same cycle-accurate behavior. To avoid inaccurate comparisons, we adopt the performance metric instructions/byte.

- We propose a methodology to verify the integrity of software written using the proposed instruction-set extensions. The methodology provides the cryptographic assurance that the Keccak based or any cryptographic algorithms running on a user's machine are identical to the algorithms as executed on the library developer's machine.

1.3 Organization

The thesis is structured as follows. Chapter 2 explains the proposed instruction set for Keccak. It discusses the Keccak sponge construction, design space exploration, proposed custom instructions and finally it offers performance and hardware cost analysis. Chapter 3 explains the integrity check methodology and emphasizes on the need for integrity verification for cryptographic applications. It discusses the threat model, proposed solution and also offers analysis of attack detection rate and performance overhead. We discuss side channel and fault attacks in chapter 4. Chapter 5 concludes the thesis.

1.4 Related Articles

A part of this work is described in the following papers:

- H. Rawat, P. Schaumont, "SIMD Instruction Set Extensions for KECCAK with Applications to SHA-3, Keyak and Ketje," ACM Hardware and Architectural Support for Security and Privacy (HASP) 2016, Seoul, Korea, June 2016.
- H. Rawat, P. Schaumont, "Vector Instruction Set Extensions for Efficient Computation of KECCAK," IEEE Transactions on Computers [**To be submitted**]

Chapter 2

Keccak Instruction Set

In this chapter we describe a novel mapping of Keccak *f/p* permutation as six new custom instructions that can be utilized to compute a Keccak-based sponge or duplex of four different sizes (1600, 800, 400 and 200 bits wide). But before offering the design space exploration and instruction specifications, we will briefly discuss the algorithmic details of Keccak family of sponge functions.

2.1 Cryptographic Sponge Functions

Cryptographic sponge functions are a class of algorithms that operate on input of variable length and produce variable length output based on a fixed length permutation. This ability to generate variable length output with tunable security level allows them to perform quasi all symmetric cryptographic operations. The sponge functions have three components: a

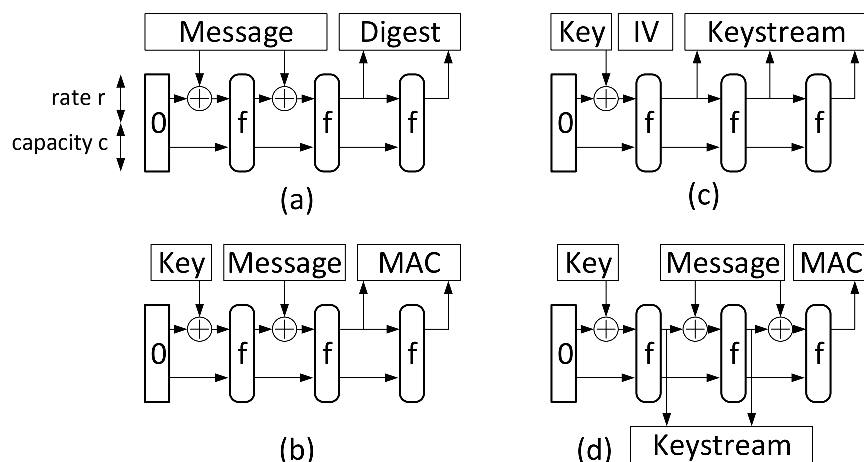

Figure 2.1: Constructions with sponge: (a) Hash, (b) Message Authentication Code, (c) Keystream Generation. Construction with duplex: (d) Authenticated Encryption

b bit state, a state permutation function f and a padding rule to adjust the input stream length to a multiple of the sponge bitrate r. The sponge capacity c is defined by $b - r$, and defines the security level. A sponge operates in two phases: Absorbing and Squeezing. The absorbing phase integrates r bits of padded input at a time, each time permuting the state. The squeezing phase extracts r bits of output at a time, each time further permuting the state. An alternate operation of sponge, called the duplex construction [5], interleaves absorbing and squeezing phases. Figure 2.1 demonstrates hashing, MACing and keystream generation using the sponge mode, and authenticated encryption (AE) using the duplex mode.

This work is based on a specific family of sponge functions based on Keccak cryptographic permutation. We specifically chose Keccak family of sponge functions as it is the fundamental building block for SHA3 [16], and for two authenticated encryption candidates in

the CAESAR competition, namely KEYAK [9] and KETJE [8]. Additionally, other KECCAK based symmetric cryptographic applications like PRNG [4] have been proposed in the literature. A detailed description of these applications is out of scope of this thesis and can be found in the cited specifications. In this thesis, we will briefly describe the working of KECCAK-f and KECCAK-p permutations which form the main computational core of KECCAK applications, and they are the main focus of our work.

2.1.1 KECCAK-f and KECCAK-p Permutations

KECCAK-$f[b]$ is a set of set of seven iterated permutations, consisting of a sequence of n_r rounds on a finite state of b bits. The value of b is defined as 25×2^l where l ranges from 0 to 6. KECCAK-$f[b]$ organizes the b bit state as a 3D matrix of dimension $5 \times 5 \times w$, where w is defined as 2^l. The number of rounds n_r is determined by the width of the permutation, $n_r = 12 + 2l$. In each round of KECCAK-f the state undergoes a set of 5 steps (transformations).

$$R = \theta \ o \ \rho \ o \ \pi \ o \ \chi \ o \ \iota$$

The $5 \times 5 \times w$ state matrix (Figure 2.3) can be split into slices as well as lanes. A slice is defined as a 5×5 matrix in the state with a constant z coordinate. A lane is an array of w bits of state with constant x and y coordinate. Depending upon the KECCAK-f permutation 1600, 800, 400 or 200, the lane size changes to 64, 32, 16, 8 bits respectively. A slice is always 25 bits irrespective of the type of KECCAK-f permutation

In brief, the 5 KECCAK steps can be defined as follows:

$\boldsymbol{\theta}$: For each column i, calculate the column parity C_{i-1} and C_{i+1} of columns $(i-1)mod5$ and $(i+1)mod5$ respectively. Left rotate C_{i+1} by one and XOR with C_{i-1}. XOR the resulting value D_i into each lane of column i. θ step requires support for XOR and ROL (rotate left) CPU instruction.

$\boldsymbol{\rho}$: Left rotate all lanes in the state by a fixed offset. For efficient implementation of ρ step, the CPU should support ROL instruction for word size equal to lane size.

$\boldsymbol{\pi}$: All lanes in the state are transposed in a fixed pattern. This step can be done using only MOV instruction.

$\boldsymbol{\chi}$: Each bit of the lane is non linearly combined with the bits of nearby lanes using AND, XOR and NOT operations.

$\boldsymbol{\iota}$: A w bit constant is XORed to a single lane.

The KECCAK-$p[b, nr]$ permutation is a generalized version of KECCAK-f permutation with tunable number of rounds $\mathrm{n_r}$. They form the basis of KEYAK and KETJE. An in-depth explanation of the KECCAK permutation can be found in the KECCAK reference [7].

2.2 Related Work

So far, there have been only a few efforts to design instruction sets for Keccak; most of the effort has gone into optimized implementations in hardware or in software. Constantin *et al.* propose custom-instruction designs for a 16 bit micro-controller [12]. They describe a set of three instructions that offer a 30% cycle count reduction for SHA3, and a 30% reduction in memory footprint (`text`). Wang *et al* describe the integration of a 64 bit Keccak datapath into a 32 bit LEON3 processor for accelerating SHA3[27]. They report 87% reduction in cycle count and a 10 % reduction in memory footprint. In comparison to these designs, our work targets a multi-purpose instruction set for Keccak applications based on a 128 bit SIMD unit.

2.3 Design Exploration for Keccak Instructions

Figure 2.2 illustrates how the Keccak permutation is utilized as a universal cryptographic kernel. All the Keccak modes are implemented using either the Sponge or else the Duplex construction. These constructions use one of the four relevant Keccak-f or Keccak-p primitives (1600, 800, 400, 200) depending upon the security goal and throughput requirement. The design approach for Keccak applications simplifies the software development process by abstracting the implementation details of the lower layers (primitives) from the generic top application layer (modes). It also localizes the optimization scope of Keccak based applications to the primitives. Our profiling results for hashing (SHA3)

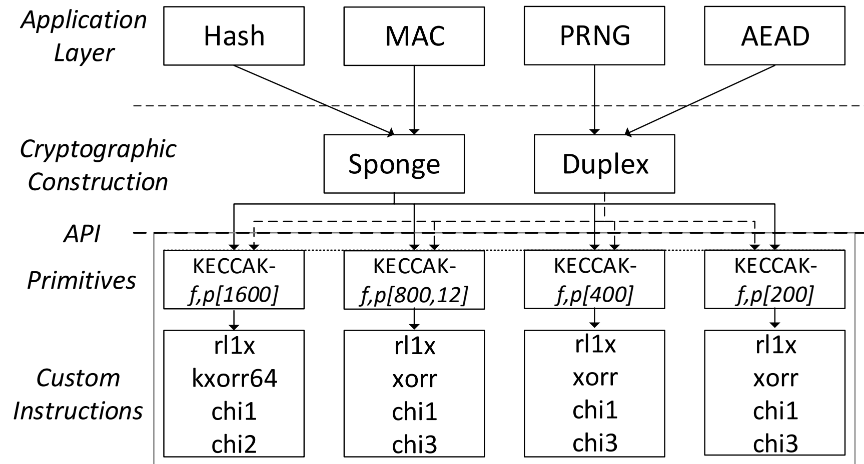

Figure 2.2: KECCAK application stack

show that 99% of CPU cycles were used for KECCAK-f [1600]. For LAKE KEYAK, 65% of the total CPU cycles were spent in doing KECCAK-$p[1600, 12]$

Our work takes advantage of the layered design of KECCAK based applications. We propose a set of six custom instructions to accelerate four KECCAK primitives 1600, 800, 400 and 200 and show the flexibility of our design by accelerating five different KECCAK applications namely SHA3, LAKE KEYAK, RIVER KEYAK, KETJE SR and KETJE JR. Our instructions are stateless, which means that we keep the entire KECCAK state into processor registers. Therefore, we first explain how to partition a relatively large KECCAK state (1600 bits) into smaller pieces that can be processed as instruction operands of 128 bits each. Later we will briefly discuss some features of the ARMv7 ISA and NEON SIMD that we used for designing the proposed instructions.

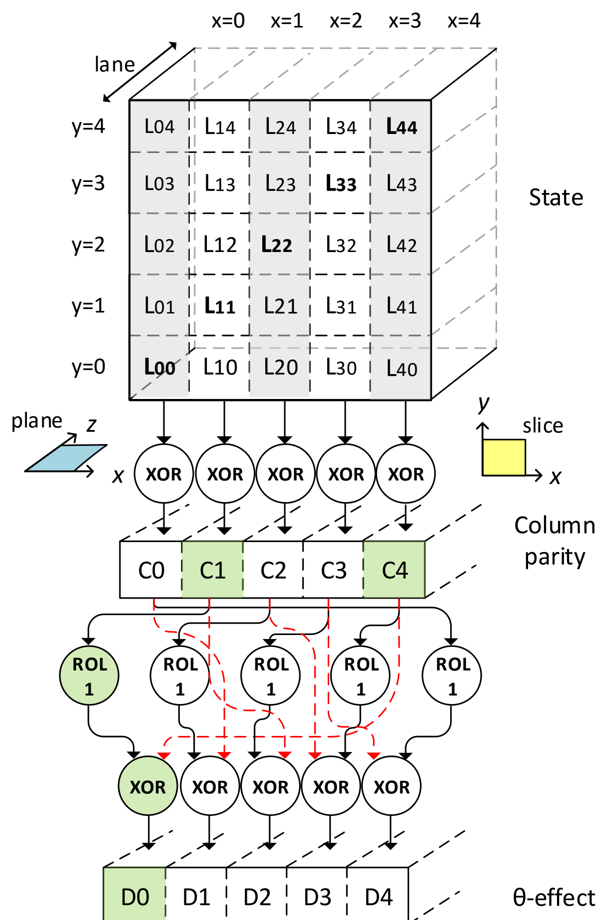

Figure 2.3: Data dependencies of θ-effect values

2.3.1 Cutting the Keccak State

A $5 \times 5 \times w$ bit Keccak state can be either viewed as w slices of 25 bits each or 5 planes with 5 lanes of w bits each. B. Jungk *et al.* propose a slice-oriented Keccak hardware [20] which is based on the observation that all Keccak steps except ρ can be done efficiently with slice-wise processing. They stored the Keccak state in 25 8×8 distributed RAMs and rescheduled the Keccak round function to perform π, χ, ι and θ steps together. Such an

approach is good for custom hardware where the state can be stored in distributed RAMs and the hardware can read the state in lane-wise or slice-wise fashion efficiently depending upon the Keccak step. From the point of view of software executing on a processor, slice-wise design may not be very efficient. First, input messages for absorption normally arrive in lane-oriented fashion, making slice-wise storage expensive in terms of data movement. Second, after the $\pi \, o \, \chi \, o \, \iota \, o \, \theta$ steps are done on the slices, the state needs to be transformed back into lane-wise orientation for ρ step. Therefore, all optimized software implementations [2] whether SIMD or 64/32 bit use plane wise processing.

The design of custom instructions requires the partitioning of the dataflow graph of the target algorithm into instruction patterns, such that the schedule length of the graph becomes as short as possible [24]. In the case of SIMD-like instructions, we are using instruction patterns of 2×128 bit input and a 128 bit output.

Keccak-f can have have a state size of up to 1600 bits (25×64 bits). For a processor with 32 64 bit registers, just holding a complete Keccak-f[1600] state in CPU register consumes 25 registers, leaving just 7 free registers to hold intermediate values during the round computation. Our custom instructions have been chosen such that register spills to main memory (LOAD/STORE) during the Keccak round are avoided, and such that that register reordering operations (MOV and VEXT) are minimized.

Figure 2.3 shows the data dependency graph for calculation of θ-effect (D_i). The column parity for each column is denoted by C_i. The lane size depends upon the Keccak-f permutation, but here we will assume Keccak-f[1600] with lane size and intermediate round

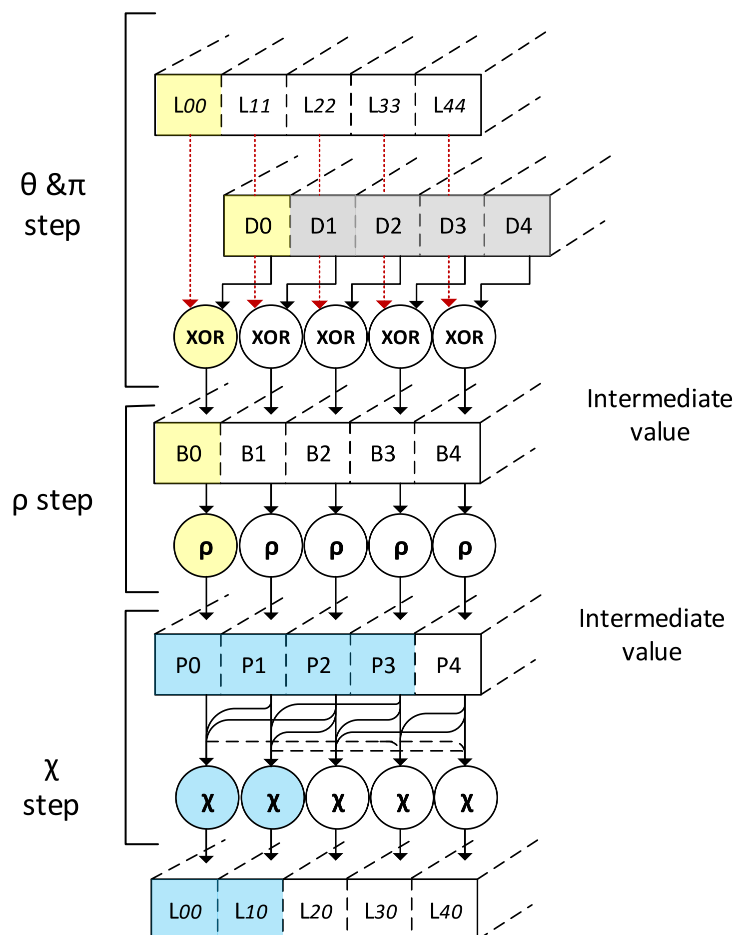

Figure 2.4: Data dependencies of θ, ρ, π and χ step (one plane)

values of 64 bits. ARM NEON already supports XOR instructions on 128 bit registers, so the column parities can be very efficiently calculated. To support calculation of θ-effect from column parities, we add an instruction which combines XOR with ROL(1) (rotate left by 1).

Figure 2.4 shows the data dependencies of θ, ρ, π and χ step for one KECCAK plane. ι step is not shown for the sake of brevity. The lanes are denoted by L_{xy} and are selected based on π step. B_i denotes the intermediate values after θ and π step. P_i denotes the intermediate values after ρ step. In a round, a single processed KECCAK lane after ι step depends on 3

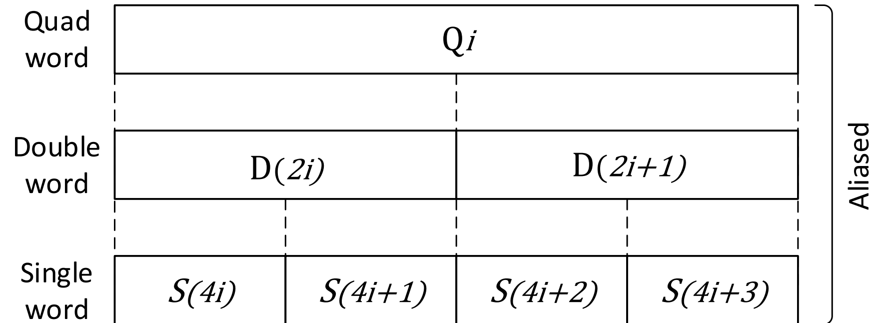

Figure 2.5: Aliased NEON registers in ARMv7

lanes, 3 θ-effect values, ρ offsets (not shown) and ι constant. Two 128 bit input registers of an instruction cannot hold all these values. One solution is to split the 64 bit lane into two independent 32 bit lanes using the bit-interleaving technique [2], so that up to eight 32 bit lanes can be packed in two 128 bit registers. We do not take this approach because of two reasons. First, breaking a 25 lane (64 bit) state into a 50 lane (32 bit) state doubles the number of register transpositions required in π step.

Second, doing the ρ step on multiple lanes of a plane in parallel requires multiple barrel shifter units in hardware which will be expensive in terms of gates. So we take an alternative approach and combine θ and ρ steps with π in a single instruction which requires one variable shifter unit in hardware. To accelerate the χ step we propose custom instructions that apply NOT, AND, XOR operation to nearby lanes packed in two 128 bit registers.

2.3.2 ARMv7 ISA and NEON SIMD

NEON instructions in ARM execute in a separate pipeline with its own register file [1]. The ARMv7 NEON unit has 16 quad-word (128 bit) registers, which are aliased with 32 double-word (64 bit) registers (Figure 2.5). In addition, the first 16 double-word registers are aliased with 32 single-word (32 bit) registers. NEON instructions have a fixed encoding size of 32 bits and operate on Quad(Q), Double(D) and Single(S) word registers depending upon the instruction definition. NEON instructions use a three-register format (2 source operands and 1 destination operand) or a format with 3 registers and an immediate value (VEXT instruction). Instructions also specify the vector alias format. For example, the `I32` specifier in `VADD.I32 q1, q2, q3` signifies that quad registers `q1`, `q2` and `q3` have 4x32-bit integer data.

2.4 Proposed Instruction Set Extensions for Keccak

We propose a set of six custom instructions for Keccak-$\{f, p\}[1600, 800, 400, 200]$ primitives. Similar to other crypto-instructions (e.g. Intel AES-NI and SHA), our instructions take advantage of the wide SIMD registers. Not all of our instructions are SIMD in nature. They operate on quad, double or single word (scalar) registers depending upon the step and Keccak permutation. Their shape and functionality is highly customized. In general, we optimized our instructions for the 1600 and 800 primitives, and we reuse the same instructions for implementing 400 and 200 primitives. The proposed instructions are of the

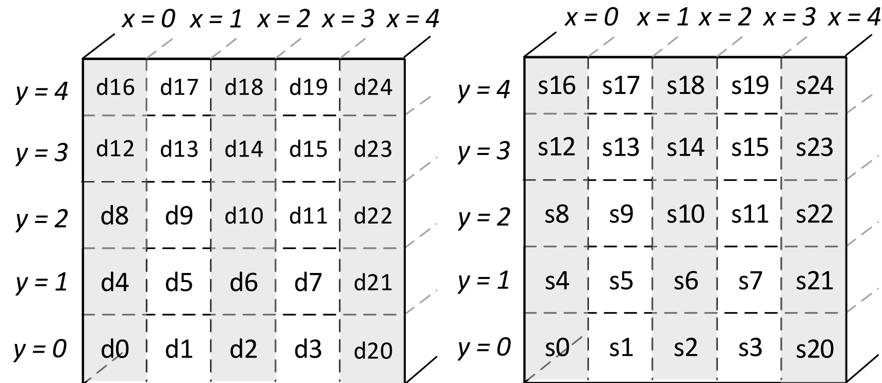

Figure 2.6: a) Register allocation for KECCAK-*f*[1600] and KECCAK-*f*[800, 400, 200]

register-to-register format, with two source operands and one destination operand. Operands are registers, and one of the source operands can also take an immediate argument. The proposed instructions are compatible with other NEON instructions and do not require special architectural features such as non-standard register files or lookup tables. Figure 2.6 shows the mapping of the three dimensional KECCAK state to the NEON registers. For the 1600 bit primitive, we map each lane to a double-word register `D0-D24` and for the 800, 400, 200 bit primitives, we map each lane to single-word register `S0-S24`. The following sections are a discussion of each of the six proposed instructions. Unless specified, the examples assume the KECCAK-*f*[1600] permutation with 64 bit lanes.

2.4.1 Instruction `rl1x`

Instruction `rl1x` (*rotate left by 1 and XOR*) takes 2 registers as input (Figure 2.7 a), left rotates the value in source register 2 by one, and XORs the resulting value to source register

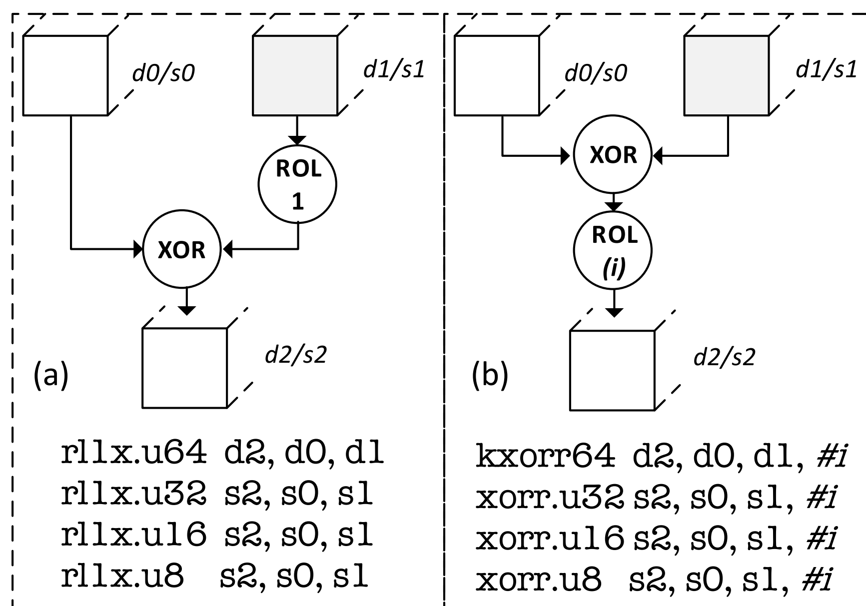

Figure 2.7: a) Functionality of `rl1x` instruction, b) Functionality of `kxorr64`, `xorr` instruction

1. The `rl1x` instruction accelerates the calculation of θ-effect value that is needed for the θ step. Once the parity of all the five columns is calculated and stored in five registers, the `rl1x` instruction operates on any two parity values and computes one θ-effect value per instruction. The instruction supports double and single-word NEON registers with vector sizes of 64, 32, 16 and 8 bits for supporting θ-effect calculation for KECCAK variants 1600, 800, 400, 200.

2.4.2 Instruction `kxorr64`

Instruction `kxorr64` (KECCAK *XOR and rotate*) uses two registers and an immediate value as operands, XORs the source registers, left-rotates the result by an immediate offset and returns the result in a destination register. `kxorr64` combines θ, ρ and π steps in a single

instruction. Figure 2.7 b shows the functionality of `kxorr64` instruction. Source operand 1 contains a lane that needs to be transposed to a new location for π step, source operand 2 contains the corresponding θ-effect value, and the immediate field contains the required ρ offset value. `kxorr64` applies θ and ρ steps and assigns the result to a new destination register (π step).

Accommodating five bits as an immediate value in the instruction encoding is challenging. VEXT supports a four-bit immediate field but for supporting 25 different rotation offsets for 25 Keccak lanes, at least 5 bits are needed. Since our instructions only support double registers, we used bit 6 of the instruction (used for defining Quad or Double register type) for encoding one extra bit of the immediate value. `kxorr64` is only used for implementing Keccak-$f, p[1600]$ permutations. For permutations of other sizes we provide a separate instruction `xorr`.

2.4.3 Instruction `xorr`

The functionality of `xorr` instruction is similar to `kxorr64` in Figure 2.7b. The only difference is that it operates on single-word registers Similar to `kxorr64`, `xorr` combines θ, ρ and π steps in a single instruction, but supports 32, 16 and 8 bit rotations for Keccak-$f[800, 400, 200]$. Depending upon the vector size specified, `xorr` treats the values in the registers as 1×32, 2×16 or 4×8 bit vectors and applies the same operation on all the vectors. Since, all 32 rotations for a 32 bit word can be encoded in 5 bit values, unlike `kxorr64`, `xorr` supports

full range of rotation.

2.4.4 Instruction `chi1`

`chi1` instruction aids χ step of Keccak. `chi1` instruction accepts 4 lanes (in 2 quad registers) containing the intermediate values after π step, and applies the χ step on them to finalize two Keccak lanes. Like other NEON instructions, `chi1` instruction also support multiple register views. A quad register can be viewed as 2×64 bit lanes (used in Keccak-f[1600]) or 4×32 bit lanes (for Keccak-f[800,400, 200]). The functionality of both the forms is shown in Figure 2.8 a.

2.4.5 Instruction `chi2`

Since a Keccak plane has an odd number of lanes, the `chi1` instruction can finalize only the first four lanes of the plane. Finalizing the last lane for every plane requires register rearrangement followed by `VBIC` and `VEOR` NEON instructions. To aid this step, we provide `chi2` instruction which can save these extra computations. Figure 2.8 b shows the functionality of `chi2`. `chi2` instruction accepts 2 quadword registers and produces a double-word register. The `chi2` instruction fits in the category of narrow instructions - the destination register is smaller then the source registers. In general, two `chi1` instructions paired with a MOV and a `chi2` instruction can apply the χ step to a complete Keccakplane. The `chi2` instruction applies only to Keccak-f, p[1600] permutations. For processing other variants

of KECCAK, we have designed a `chi3` instruction.

2.4.6 Instruction `chi3`

Just like the `chi2` instruction, the `chi3` (Figure 2.8 c) instruction is also an auxiliary instruction that helps fixing the last lane of a plane without register rearrangement and `VEOR` and `VBIC` instructions. It takes two double-word registers as input source operands, producing a 32 bit single-word as result. A `chi1` instruction followed by a `chi3` instruction can apply χ step on a complete KECCAK plane. `chi3` instruction is used for KECCAK-$f[800, 400, 200]$ permutations. Since the χ step only contains XOR, NOT and AND operations we support only U32 vector size for `chi3`. Lanes of sizes 16 and 8 bits can also be stored in 32 bit single-word registers and `chi1` followed by `chi3` can perform χ step for KECCAK-$f[400]$ and KECCAK-$f[200]$.

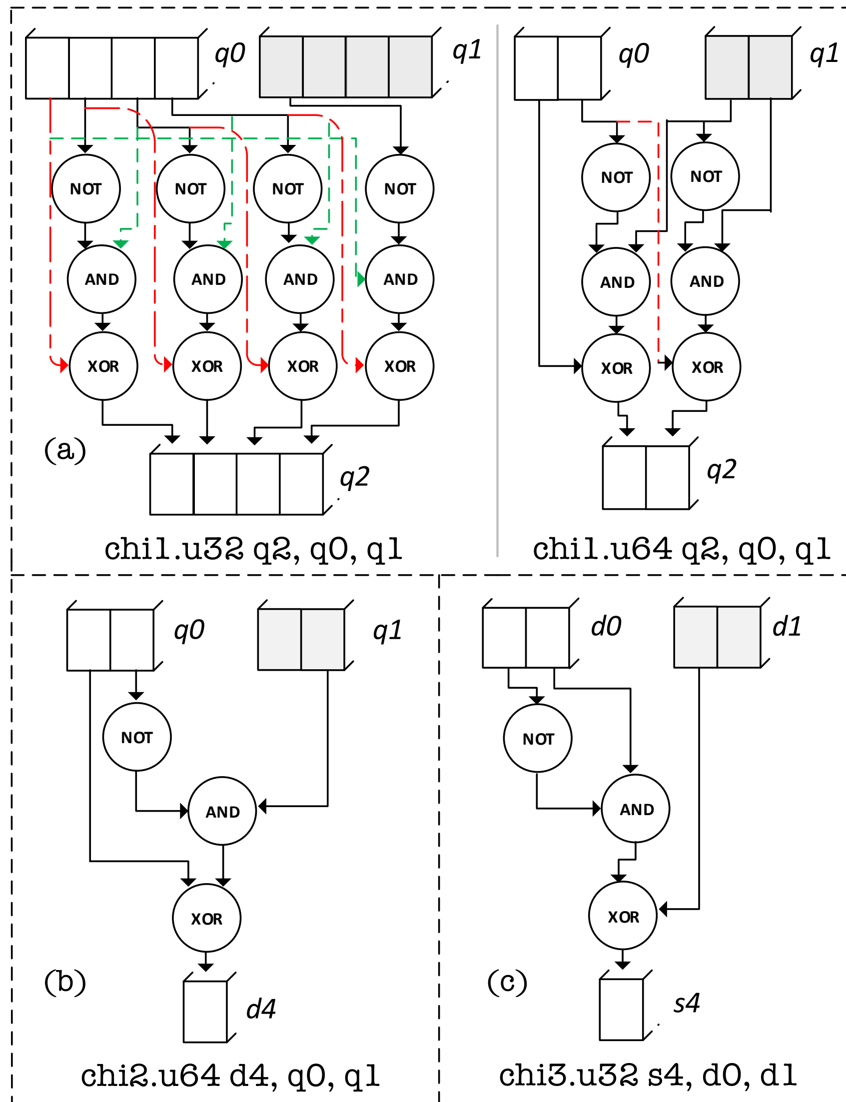

Figure 2.8: Functionality of instruction a) `chi1`, b) `chi2` and c) `chi3`

2.5 Results

We used the open-source GEM5 simulator for implementation and benchmarking of the proposed custom instructions. We added our instructions to the ARMv7 ISA part of GEM5 and described their functionality using GEM5's ISA description language. The NEON SIMD unit operates using a separate register file and pipeline. Instructions are sent to NEON pipeline from the ARM side via an instruction queue. Both the ARM pipeline and NEON pipeline can work independently as long as the queue is not full. The ARM core as well as the NEON SIMD unit also support multi-issue. Factors like instruction latency, instruction queue length, issue width of the CPU are all implementation specific and difficult to model with a high degree of accuracy on a simulator. To get a reasonable estimate of the performance improvements we chose a simple timing CPU model available in GEM5. The timing simple CPU model executes instructions at a rate of 1 CPI (cycles per instruction) but takes the latency of memory system into account by adding stalls on cache accesses. We present our results in terms of dynamic instructions committed by the CPU normalized to input message size of the crypto algorithm.

2.5.1 Simulation Setup

We simulated a single core ARM CPU at 1 GHz with an L1 I and D cache of 32KB and L2 cache of 2MB. For generating the executable binaries we used cross-compiled GCC-4.9.2 and added the instruction encodings to ARM specific part of the GNU assembler. We followed

the encoding format for NEON instructions and made sure that the instruction codes do not conflict with the existing NEON and ARM instructions. For comparison purposes, we used the optimized implementations of SHA3 (c = 1024), Keyak and Ketje available in Keccak code package [10]. We used Known Answer Tests to verify the functional correctness of SHA3(c=1024). For Keyak and Ketje we used CAESAR testbench in the Keccak code package.

2.5.2 Performance

Table 2.1 shows the results of our optimized assembly implementations using proposed custom instructions (CI) compared to the optimized C, hand optimized 32-bit assembly and NEON assembly implementations in the Keccak code package. All the measurements were taken on the GEM5 with the setup discussed above. The simulation statistics were taken for the complete crypto application with an input block size of 10, 100 and 1000 blocks. The averaged results are shown in instructions committed by CPU per byte of input data. The expected speedup is calculated against the optimized NEON or 32 bit assembly implementations based on the availability of implementation in the Keccak code package. Table 2.2 gives the instructions committed for processing a single round of Keccak-f for different software implementations. Our implementations using CI reduce the instructions committed by CPU by a factor of 1.4 to 2.6× depending upon the application. The expected speedup on a real hardware should also be very similar for three reasons. First, our implementations of Keccak primitives do not incur any branches during the round computations and offer ILP

Table 2.1: Performance in instructions/byte for various Keccak modes

Mode	**C**	**32-bit ASM**	**NEON**	**CI†**	**Speed -up**
SHA3(c=1024)	243.5	143.9	48.1	**21.9**	2.2
Lake Keyak(E)	61.0	NA	13.4	**7.7**	1.7
Lake Keyak(D)	61.7	NA	14.9	**9.2**	1.6
River Keyak(E)	55.2	39.3	NA	**14.8**	2.6
River Keyak(D)	57.2	40.6	NA	**16.1**	2.5
Ketje SR (E)	166.1	87.9	NA	**55.0**	1.6
Ketje SR (D)	166.1	87.9	NA	**55.0**	1.6
Ketje JR (E)	309.1	146.6	NA	**106.5**	1.4
Ketje JR (D)	309.1	146.6	NA	**106.5**	1.4

† This work. E = Encryption, D = Decryption, NA = Not Available

(instruction-level parallelism) ranging from degree 2 to 4. Second, our instructions use simple operations like XOR, AND, NOT and rotations, which can be completed in single clock cycle in execution stage of processor's pipeline. Third, our implementations of Keccak round do not cause any register spills that might affect throughput. Hence, Keccak applications implemented using custom instructions should be able to achieve a CPI very close to 1 on a superscalar processor with low memory latency.

Table 2.2: Instructions executed per Keccak round

Primitive	C	32-bit ASM	NEON	CI
Keccak-f[1600]	713	414	145	**66**
Keccak-f[800]	271	194	NA	**56**
Keccak-f[400]	370	217	NA	**55**
Keccak-f[200]	361	168	NA	**57**

2.5.3 Hardware Cost

To get a reasonable estimate of the hardware overhead of our proposed design, we created RTL designs for all the six custom instructions. We assumed that the functional units will get upto 2×128 bit inputs and a 5 bit immediate field and generate a 128 bit output. For synthesis, we used Synopsys Design Compiler that generates a netlist using UMC′s 90nm Process. The area estimates for each instruction have been provided in Table 2.3 with corresponding gate equivalent (GE) and transistor equivalent counts. An ARMv7-A based quad-core processor[25] uses an area of $3.8mm^2$ in 28nm technology. On converting the area into GE [14], a single ARM core has around 3.8 million gates. Adding Keccak instruction extensions will cost an extra 4658 gates, which adds an overhead of 0.1% while significantly improving the performance of hashing, MACing and a range of other cryptographic applications based on Keccak sponge and duplex construction. Apart from the hardware overhead of functional units, instruction decode logic will also have some hardware overhead of additional custom instructions. But since our instructions follow similar encoding format as that of other NEON instructions, the decoding overhead should be negligible.

Table 2.3: Area, GE and Transistor count estimates for the proposed custom instructions

Instruction	μ^2	**Gate equiv.**	**Transistor equiv.**
`rl1x`	1238	310	1238
`kxorr64`	7474	1869	7474
`xorr`	4884	1221	4884
`chi1`	3576	894	3576
`chi2`	966	242	966
`chi3`	486	122	486
Total	18624	4658	18624

2.6 Portability Aspects

The instruction design and results presented in this work are for ARMv7 ISA with NEON SIMD. We will next describe how these instructions can be ported to other platforms. While we do not claim that the speedup and code footprint reduction will be same as shown in Section 5 for these other targets, we will explain how our instructions can improve KECCAK's performance on those platforms.

Table 2.4 shows the feasibility of instructions with respect to three different architectures, namely Intel AVX (128 bit SIMD), generic 64 bit platforms and 32 bit embedded platforms. This feasibility analysis is based on the number of operands supported by the ISA, the register width and the instruction encoding width.

Table 2.4: Feasibility of the proposed instructions on different platforms

Instruction	Intel AVX	64-bit Arch	32-bit Arch
`rl1x`	✓	✓*	✓*
`kxorr64`	✓	✓	✗
`xorr`	✓	✓	✓
`chi1`	✓	✗	✗
`chi2`	✓	✗	✗
`chi3`	✓	✓	✗

* Not all variants supported

2.6.1 Intel AVX (128 bit SIMD)

AVX[13] is the SIMD instruction set supported by Intel processors. It supports 128 bit wide registers and instructions with three operand, nondestructive format `(dest = src1 + src2)` like ARM NEON. AVX supports sixteen 128 bit XMM registers, which were later extended to 256 and 512 bits in AVX2 and AVX-512, with backward compatibility to 128 bit XMM registers. Since our design is targeted for 128 bit wide registers, we will only discuss the portability aspects of our instructions on 128 bit SIMD. The instruction design space will change with AVX2 and AVX-512 as a chunk of the Keccak state can be stored in wider registers. Unlike NEON, AVX does not support aliasing of 128 bit registers to two 64 bit or four 32 bit registers . Since, some of the instructions like (`rl1x`, `xorr`) need access to an independent 64 bit or 32 bit scalar value inside a 128 bit register, we propose to encode

scalar index in the instruction encoding. Intel Architectures support instruction encodings of up to 120 bits, so a 2 bit scalar index, for each register in the instruction can be encoded in the instruction encoding to apply the operation only to a particular scalar. Thus, all of the six instructions can be implemented on an Intel platform supporting AVX without any major architectural changes.

2.6.2 64-bit Architectures

Some of the instructions can also be implemented on 64 bit architectures like ARMv8, x64. All proposed instructions except `chi1` and `chi2` operate on 64/32 bit operands and produce a 64/32 bit result. Implementing `kxorr64` and `rl1x` instructions on 64 bit platforms can give a performance boost to Keccak as they can accelerate the θ, ρ and π step. The `chi1 & chi2` instructions require 128 bit wide registers and cannot be supported on a 64 bit architecture, but alternatively χ step can be accelerated by supporting ternary instruction like `ternchi(dest = dest⊕(∼src1 & src2))` or `andn (dest = ∼src1 & src2)`. Smaller versions of Keccak can also benefit from these instructions, if a w bit lane is stored in a 64 bit register and the word size for rotation 64, 32, 16 and 8 is passed to hardware as instruction encoding.

2.6.3 32-bit Architectures

For 32 bit embedded platforms, `rl1x` and `xorr` instructions can be very useful. All optimized 32 bit software implementations use bit-interleaving technique to break a bigger 64 bit lane to smaller 32 bit lane. These 32 bit lanes are used independently for rotations and other Keccak steps. A 32 bit version of `rl1x` and `xorr` instruction can help accelerate the θ, ρ and π step on smaller embedded platforms. Like 64 bit, for 32 bit architectures also, `andn` or `ternchi` can be useful for accelerating χ step.

Chapter 3

Integrity of Cryptographic Implementations

3.1 Introduction

In this work we investigate techniques to check the correct operation of a cryptographic software library, in a manner that is easy to support by the library developer, and easy to verify by the end user. The idea is to instrument the library to measure the data and control flow as it executes the cryptographic APIs on a trusted system, and convert this measured control and data flow into cryptographic digests. A library developer can then embed security checks throughout his source code that measure cryptographic digests on the user system and verify them with the values collected by the developer. Any mismatch between the

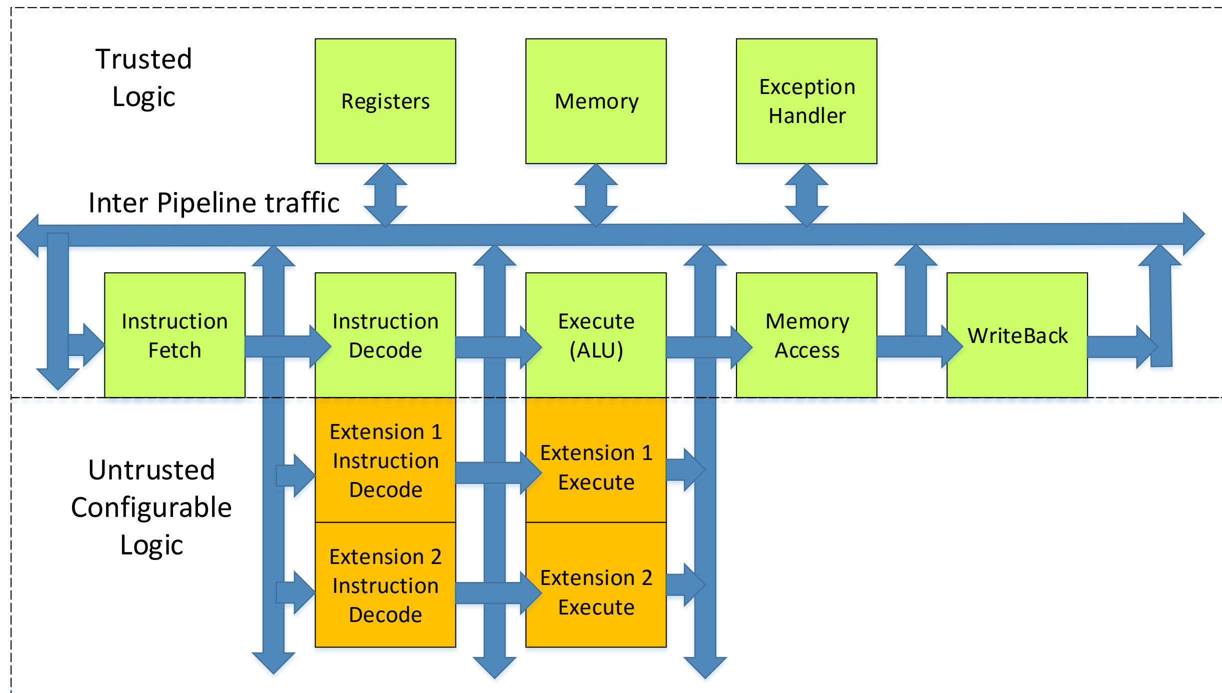

Figure 3.1: Block Diagram of an extensible RISC Processor

actual and expected digest values is treated as an error. The verification mechanism is thus transparent to the user and is fully controllable by the developer of the cryptographic library.

3.1.1 Need for Integrity Verification

Consider the deployment of a new cryptographic primitive (such as a hash based signature or a new mode of operation) as a cryptographic library. The user of such a library might use its APIs for protecting sensitive information and relies on the library developer for correctness of the cryptographic primitive. This desired claim of correctness should cover the software binary code as well as the hardware platform executing this code. The latter is

relevant because modern reconfigurable hardware platforms contain upgradeable firmware. Intel Processors use microcodes to break complex CISC instructions into simpler RISC type instructions. These microcodes can be updated using firmware updates. Current research in computer architecture proposes extensible processors [28], [23] (Figure 3.1) which allow addition of specialized instructions, either through modifications during design stage or at runtime through configurable logic. Hence, even an identical software binary which uses these specialized instructions may behave differently depending on the hardware platform.

Another concern with the design of cryptographic software and firmware is that bugs in the design may result in additional security leaks. These bugs may be accidental programming mistakes, malicious attacks or transient faults induced due to the physical environment in which the system is operating. Since, majority of application softwares and protocols rely on cryptography for security, one should be concerned about the reliability of cryptographic application as well as the hardware platform executing it.

3.1.2 Threat Model and Root of Trust

Swierczynski *et al.*, describes an attack on a commercial secure USB stick that succeeds by weakening the functionality of the AES cipher by inserting a hardware Trojan in the configurable logic [26]. Similar security concerns arise from any untrusted custom instruction in the hardware that is used for accelerating cryptography. Dinur *et al.*, [15] demonstrated successful hash collisions on a round reduced version of Keccak . Traditionally, a "reduced round

attack" is not considered practical, because cryptographers make a black-box assumption of the cryptographic algorithm. But, these assumptions change when an adversary obtains control of the execution of software. The integrity of a cryptographic application can be easily compromised through software attacks that exploits vulnerability either in the crytographic library, trusted code (Operating System, hypervisor), or the hardware platform. A number of software exploits like code injection, stack overflow have been reported in the literature [19] that exploits weakness in the software.

Hence, we consider a threat model that covers the application software as well as the custom instruction implementations invoked by the application. Our proposed strategy uses additional trusted instructions which are part of the base instruction set of the processor. These instructions can be used by a trusted library developer to embed integrity checks in the cryptographic library. High performance optimized cryptographic libraries are written by experts and we believe that the right person to write security checks for the crypto should be the software developer himself. Our instructions are aimed at detecting tamper in the software or in the untrusted part of the hardware. They are built on earlier ideas in Control Flow Integrity (CFI) and Data Flow Integrity [21], [11] and try to capture control flow, data flow as well as functional correctness of the cryptographic implementation.

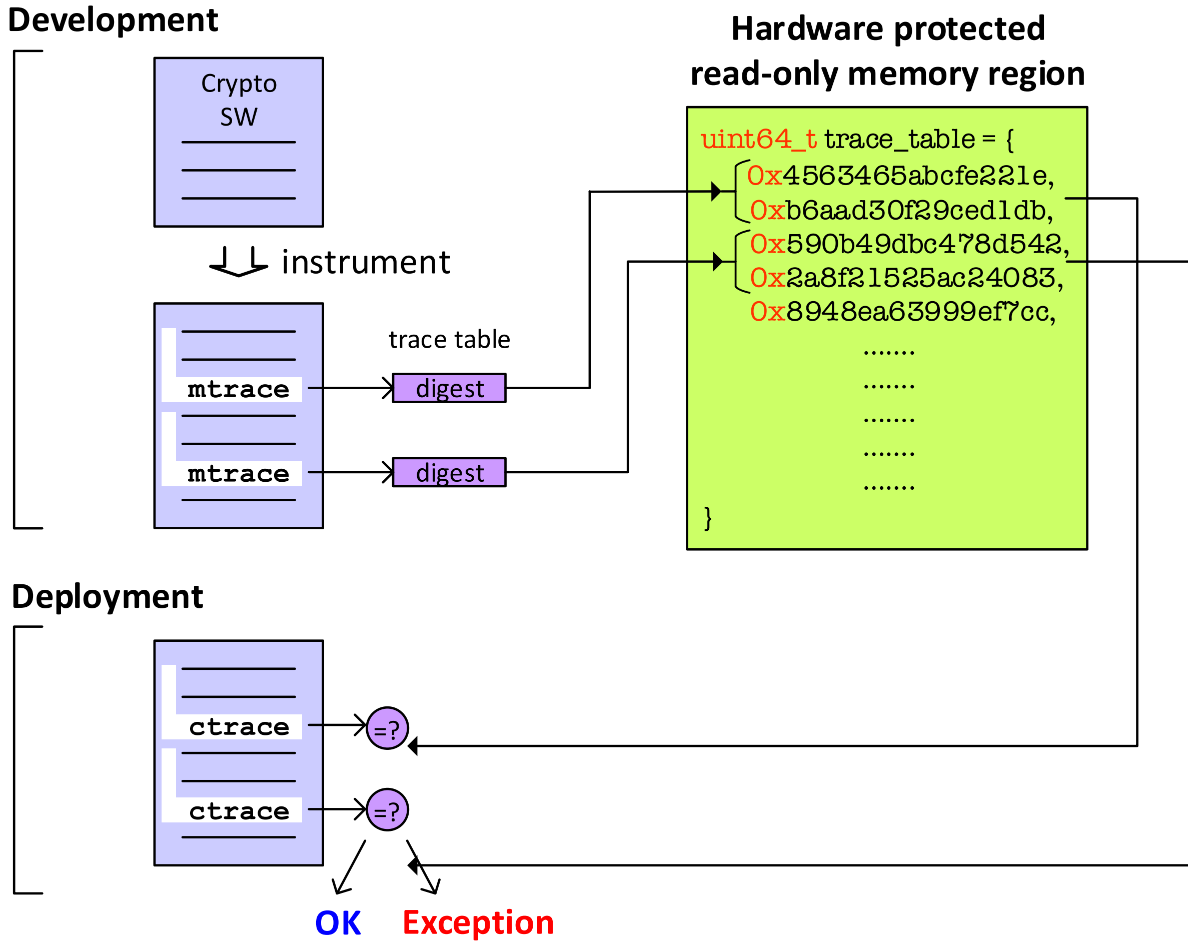

Figure 3.2: Proposed methodology for protecting cryptographic kernels

3.2 Proposed Methodology for Integrity Verification

We propose three custom instructions `mtrace`, `ctrace` and `mconfig` which are part of root of trust of the microprocessor. These instructions are sufficiently generic to protect any cryptographic kernel built using untrusted custom instructions. Figure 3.2 illustrates the principle of operation. During the development, the library developer writes code and its associated KAT (known answer test) which is instrumented with measurement primitives

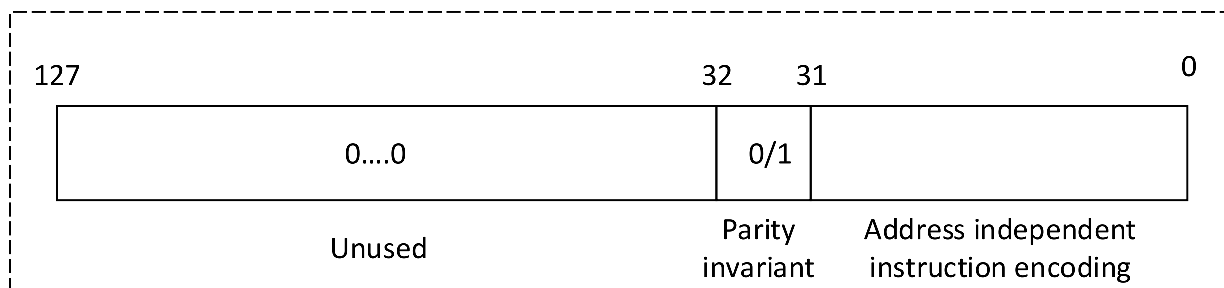

Figure 3.3: Measurements used for hash input

(`mtrace`), which compute a cryptographic hash over the runtime properties of the software and custom logic. The `mtrace` primitive emits the digest in a trace table. Before distributing the library, the measurement primitives are replaced by check primitives (`ctrace`) and the trace table containing cryptographic digests is added to the software source. Additionally, the developer also embeds security and configuration parameters through `mconfig` instructions. These parameters, protect the trace table against tampering and enforce a lower limit on the frequency of `ctrace` instructions. During deployment, the enhanced KAT and APIs execute on the end-user's system. Each check primitive re-evaluates the digest and compares it to the corresponding trace table entry. A discrepancy is then flagged by means of a security exception.

Figure 3.3 specifies the measurements used by `mtrace` instruction for capturing expected program behavior. The digest of the hash is computed over the address independent part of the instruction encodings and a 1-bit value indicating a match of the expected and actual parity for the custom instructions. The digests of instructions from different functions of the program can be chained together into a single trace. `mtrace` measurement primitive is

flexible enough to enable the programmer to capture program properties like Inter Procedural control flow, Intra Procedural control flow, instruction stream integrity, computational integrity of the custom instruction and data flow integrity of the program at the level of CPU registers. This enables the library developer to write strict integrity checks without significantly compromising on the performance. The complete specifications of `mtrace` and `ctrace` and `mconfig` instructions is provided in the next section, the section after that provide examples that demonstrate the use of these instructions in a cryptographic code.

3.3 Trusted Instructions for Integrity Verification

3.3.1 Instruction `mtrace`

mtrace reg, #imm

Instruction `mtrace` is the measurement primitive that returns the trace value accumulated at the current instant in the hardware trace register. This value is written to the specified destination register. Additionally, the developer can skip the previous or next instruction from the trace computation. A complete list of supported commands accepted by `mtrace` instruction are provided in table 3.1.

3.3.2 Instruction `ctrace`

ctrace reg, #imm

Table 3.1: `mtrace` commands

Command	**Mnemonic**	**#imm**
Measure trace	#mtr	0
Ignore Previous Instruction and Measure	#igpm	1
Ignore Next Instruction and Measure	#ignm	2

Table 3.2: `ctrace` commands

Command	**Mnemonic**	**#imm**
Compare trace	#cmp	0
Ignore Previous Instruction and Compare	#igpc	1
Ignore Next Instruction and Compare	#ignc	2

Instruction `ctrace` is the primitive for comparison and flagging error. It accepts a 128-bit source register and matches its value with the current value of the internal trace register. If both the value matches, it means that the program is following a permissible behavior. On a mismatch, `ctrace` triggers an exception. The library developer can catch this exception to print an error message before terminating the program. `ctrace` has the same format as `mtrace` and the tracing unit uses same opcode for `mtrace` and `ctrace` for calculating digest. Hence, both the instructions are interchangeable. A complete list of supported states of the `ctrace` instruction are provided in table 3.2

3.3.3 Instruction `mconfig`

mconfig reg, #imm

Instruction `mconfig` provides necessary features to control the behavior of hardware tracing unit. The hash value in the internal trace registers are read-only and the software developer can only reset them to a hard-coded fixed value using `mconfig`. The hardware provides up to three 128-bit secure registers to collect trace values of various sections of code. This helps in writing integrity checks that can handle data dependent and inter-procedural control flow in the program. A complete list of supported states of the `mconfig` instruction are provided in table 3.3. Additionally, `mconfig` allows the developer to configure the trace table parameters, to configure the security level and to add checkpoints:

Trace Table: As shown in figure 3.2, trace table is a contiguous area of memory that stores the trusted trace values stored by the developer. Protection of the trace table from any kind of tampering during runtime is an important requirement of our approach and is supported by `mconfig` instruction. `mconfig` instruction can accept a 32-bit memory address using #ldaddr command and its size using #ldsize command. The specified region of memory will be made as read-only till the entire duration of program. Also, #ldaddr and #ldsize commands can only be used once during the entire program execution. This is to protect against any attack that tries to change the trace table parameters by executing malicious code containing `mconfig` instructions.

Security Level: `mconfig` also supports three security levels: 0 (highest), 1 and 2 (lowest). Our scheme is based on machine level instructions and the trace values are checked only when `ctrace` is executed. If somehow, an attacker is able to takeover program control and start executing malicious code without any `ctrace` instructions, no exceptions will be

raised. To prevent such a scenario, a security level can be initialized which forces use of `ctrace` after every 10, 25 and 50 assembly instructions based on specified security level 0, 1 or 2. Additionally, once `mconfig` is initialized, the tracing unit monitors all load/stores to memory and make sure that the trace values used for `ctrace` are taken only from memory region dedicated for trace table. It also makes sure that init, start and stop commands are issued in proper order and all the configuration registers are initialized before using `mtrace` and `ctrace`. For any violation, an exception is raised.

Checkpoints: Checkpoints are unique values supported by `mconfig` that can be added to the hash input using `mconfig` instruction. They are used in complex control flows to mark the execution of a function.

Table 3.3: `mconfig` commands

Command	**Mnemonic**	**#imm**
Reset trace register 0	#init0	0
Reset trace register 1	#init1	1
Reset trace register 2	#init2	2
Reset all trace register	#initall	3
Security Level 0	#sec0	4
Security Level 1	#sec1	5
Security Level 2	#sec2	6
Start tracing	#start	7
Stop tracing	#stop	8
Select trace register 0	#treg0	9
Select trace register 1	#treg1	10
Select trace register 2	#treg2	11
Select reg 0 and start	#st0	12
Select reg 1 and start	#st1	13
Select reg 2 and start	#st2	14
Load trace table address	#ldaddr	15
Load trace table size	#ldsize	16
Insert a checkpoint	NA	17-255

3.4 High Level Design

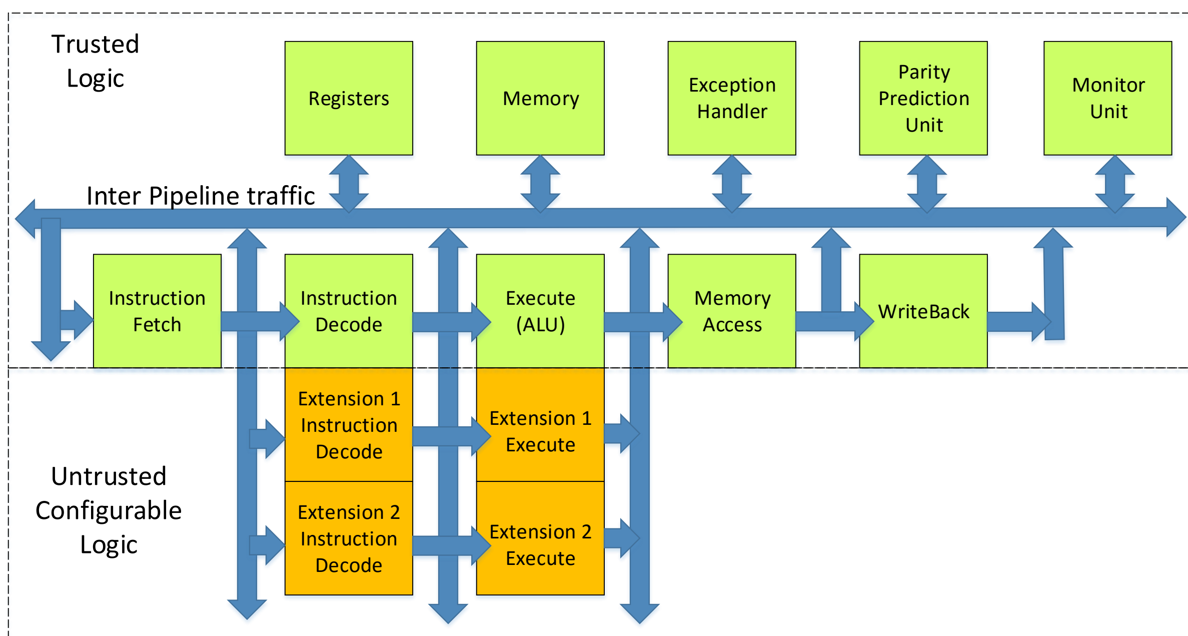

Figure 3.4: Block diagram of an Extensible RISC CPU with tracing support

Figure 3.4 shows a high level view of the modified CPU with the proposed integrity check mechanism. We demonstrate our methodology on classic 5-stage In-order RISC CPU. However, our method should be easily ported to complex OoO processors with in-order commit stage. Our design introduces two additional units in the pipeline, a parity prediction unit and a monitor unit. The instructions committed by CPU in WB stage are pushed into a 128-bit FIFO queue in the monitor unit. The monitor unit works asynchronously with the main pipeline and only stall the main pipeline in the event of a trace mismatch, instruction queue full or when the monitor unit is not ready with the trace value. Based on the throughput, hardware and security goals, our design uses GHASH [22] and a single bit parity

prediction logic. GHASH is a hashing algorithm used in the AES-GCM mode of authenticated encryption and is very suitable for hardware designs with low area and latency. It is based on Galois multiplication over a finite field of GF(2^{128}) with digest size of 128. We will now discuss each unit in detail.

3.4.1 Monitor Unit

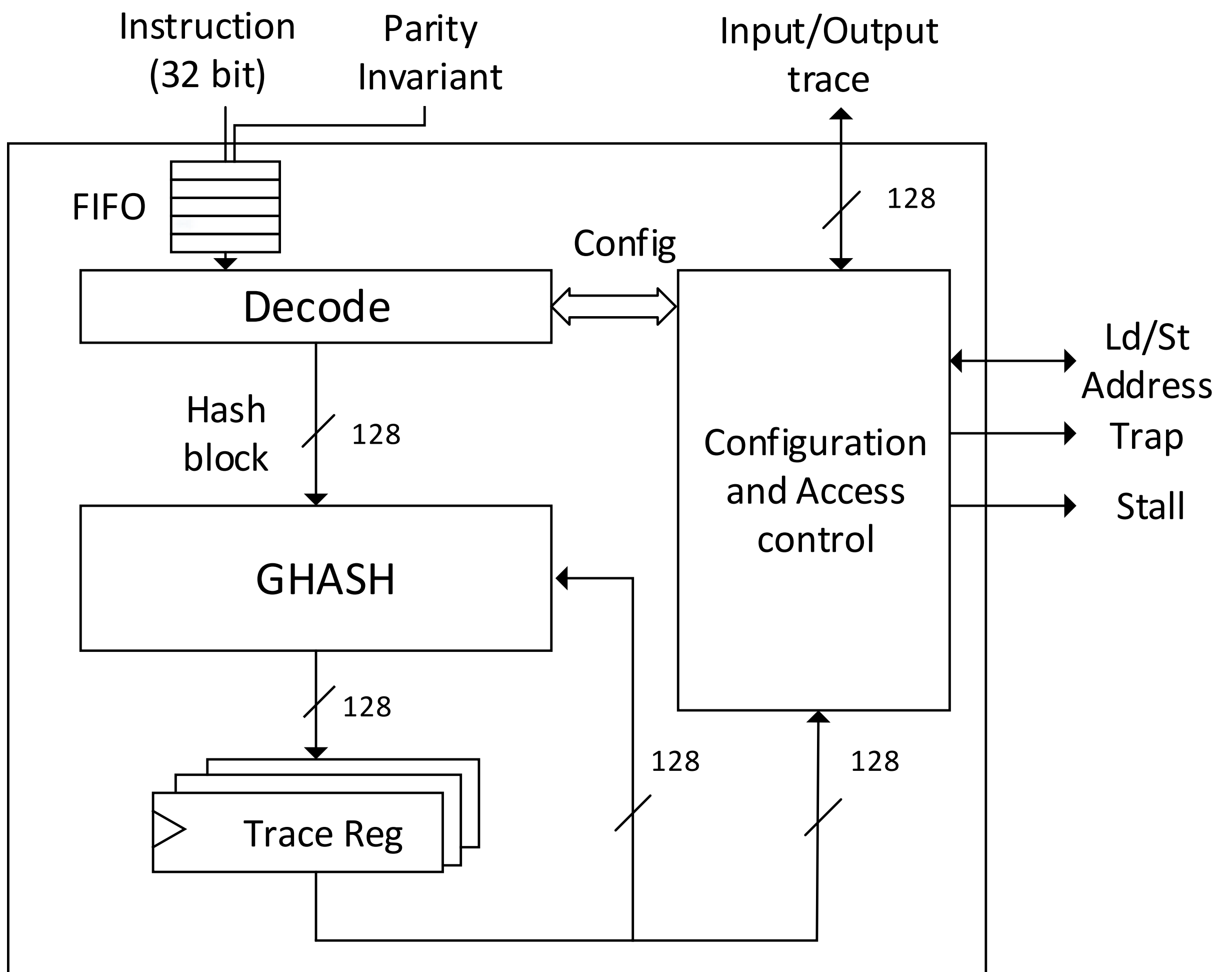

Figure 3.5: High Level Design of the Monitor Unit

The monitor unit is the main hardware unit responsible for calculating GHASH, raising exceptions and synchronization with the main pipeline. As shown in figure 3.5, the unit receives committed instructions and parity invariant in a 128-bit wide FIFO. The monitor unit continuously polls this FIFO for data and sends them to decode logic. From here, the instructions are sent to GHASH block after their address related immediate parts (for B, BX, BLX) are truncated. The unit has three read-only 128-bit trace registers for accumulating digests. Based on the commands provided by `mconfig`, `ctrace` and `mtrace`, monitor unit continuously calculates hash from the input hash block. These digests are stored in the read-only trace registers and sent to the pipeline when `mtrace` is executed. Additional registers, in the configuration and access control block store the trace table size, starting address, and the security level. The unit also keeps a track of the Load/Store operations in the MEM stage to make sure the trace table is read-only and all the security properties specified using the security level are met.

3.4.2 Parity Prediction Unit

Table 3.4: Parity Prediction Equations for Keccak-*f* [1600] Custom Instructions

Instruction	Predicted Parity
`rl1x`	$P(Result) = P(Op1) \oplus P(Op2)$
`kxorr64`	$P(Result) = P(Op1) \oplus P(Op2)$
`chi1.64`	$P(Result0) = P(Op1[0]) \oplus (P(Op1[1]) \oplus P(Op2[0]) \oplus P(\neg\, Op1[1] \cup Op2[0]))$
`chi2.64`	$P(Result) = P(Op1[0]) \oplus (P(Op1[1]) \oplus P(Op2[1]) \oplus P(\neg\, Op1[1] \cup Op2[1]))$

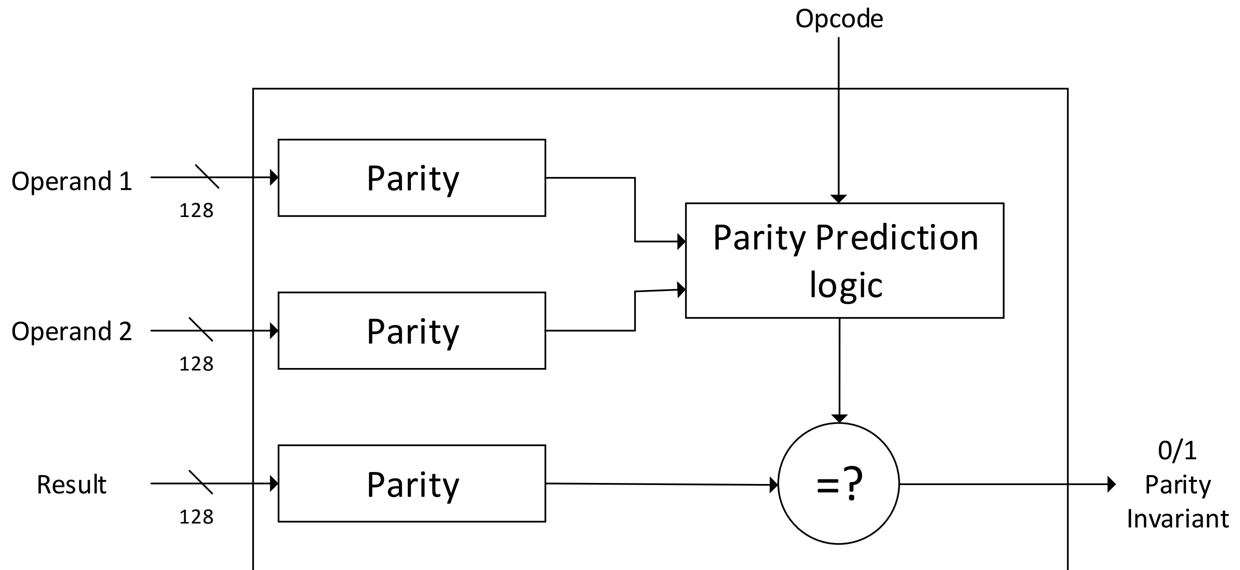

Figure 3.6: High Level Design of the Parity Prediction Unit

As shown in figure 3.6, the parity prediction unit receives instruction opcode, input and the output operand values from the currently active functional unit. It is aimed for a lightweight runtime functional verification and checks, that the untrusted functional units are operating according to their specification. Table 3.4 lists the parity prediction equations for custom instructions required for computing KECCAK-f [1600] kernel. They can be easily derived from the specifications provided in section 2.4 of Chapter 2 "KECCAK Instruction Set". The parity prediction logic can be included in the design stage of the processor for all the 3rd party custom instruction IP blocks. For extensible processors with soft fabric, the parity prediction logic can be configurable and should come from a trusted source. The unit predicts the parity of the result of the instruction and if the predicted parity does not match the actual parity of the result, the parity invariant is set to 1. The monitor unit reflects this mismatch

in the hash value.

3.5 Protecting Crypto using Trace Monitoring

In contrast to generic software applications which may require complex data structures and pointer arithmetic, cryptographic algorithms are mostly a set of mathematical transformations achieved using simple boolean operations like XOR, AND and SHIFT. For performance requirements, the core kernel of the algorithm is often hand-coded in assembly as a straight line code (with unrolled loops) using a set of specialized instructions like `AES-NI` or Keccak CI. In this section, we show, how a library developer can capture execution traces of hand optimized cryptographic kernels with linear control flow and also handle common programming situations like data dependent branch statements and intra-procedural control flow.

3.5.1 Straight line code

As shown in figure 3.7, the first line of the program should be the initialization of the trace table parameters and resetting the trace registers. As the program executes the monitor unit will compute the execution trace in parallel. On encountering an `mtrace` instruction, the current value of the trace will be returned in the destination register. This value should be stored into a trace table using a store operation. During deployment, `mtrace` instructions are replaced by `ctrace`, and stores to trace table are replaced by load operations. Since,

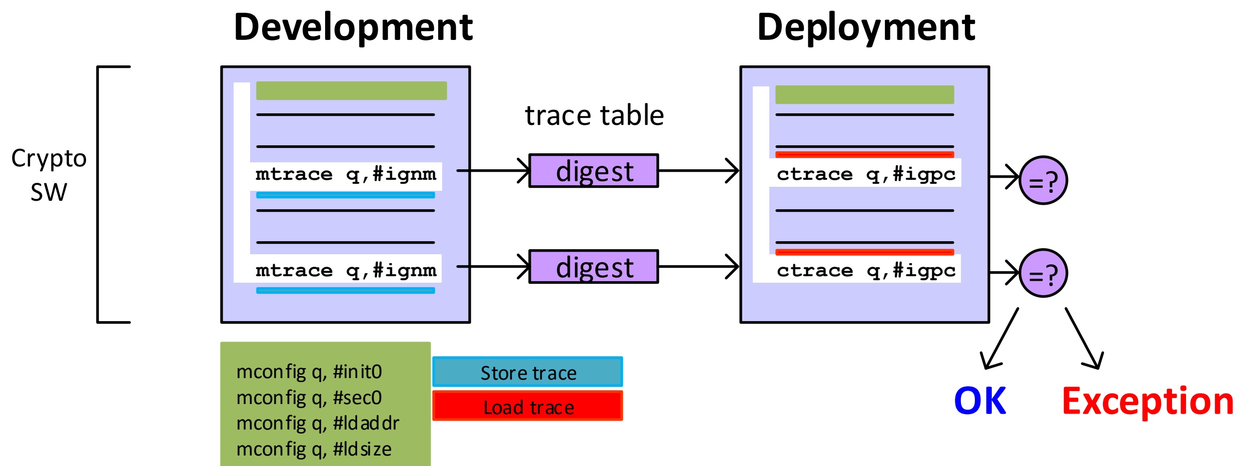

Figure 3.7: Protecting Linear code

the trace registers are not reset to default value, the hash values computed on a `mtrace` also depends on the previous `mtrace` block. This chaining of hash values improves security, as an attacker has to skip all the `ctrace` instructions to skip error detection.

3.5.2 Handling Data Dependent Branches

The tracing unit has three trace registers that can be used interchangeably for collecting instruction stream traces. As shown in figure 3.8 `if` block can be handled separately without breaking the hash chain, by collecting its trace in a separate trace register. The same methodology can be applied for data dependent loops and `if-else-if` blocks. Nested if statements should be avoided, by replacing them with separate if blocks. As clear from the figure 3.8, data dependent branches in the code are source of vulnerability in the program, so it's recommended that they should also be protected using other techniques like

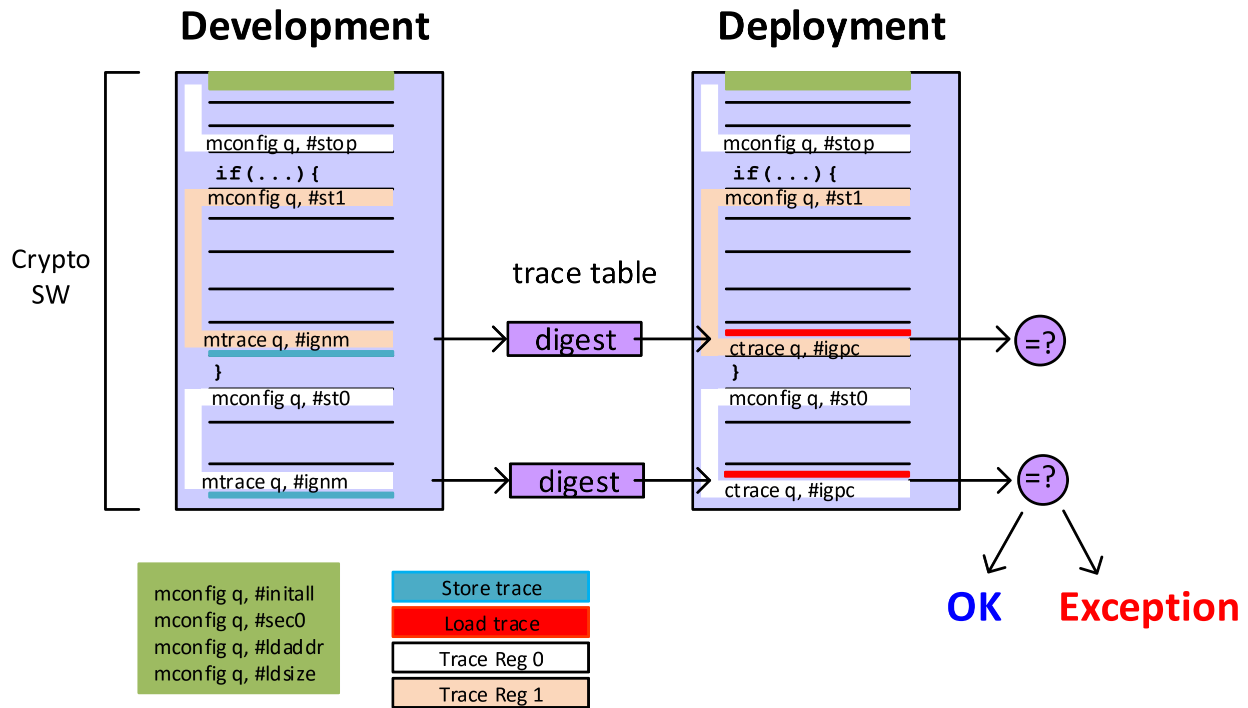

Figure 3.8: Handling data dependent branches

assert statements to check if data input is within a valid range or other safe programming techniques.

3.5.3 Handling Intra Procedural Control Flow

Control flow of a program, in principle is independent of the actual opcodes. Control flow integrity deals with verifying the addresses of the jump locations across function boundaries. It tries to capture the shape of the Control flow graph of the program. The digest of each function in a program, derived from its instruction stream is a unique identity of the function. If these digests are chained together in an order that represents the valid control flow of a

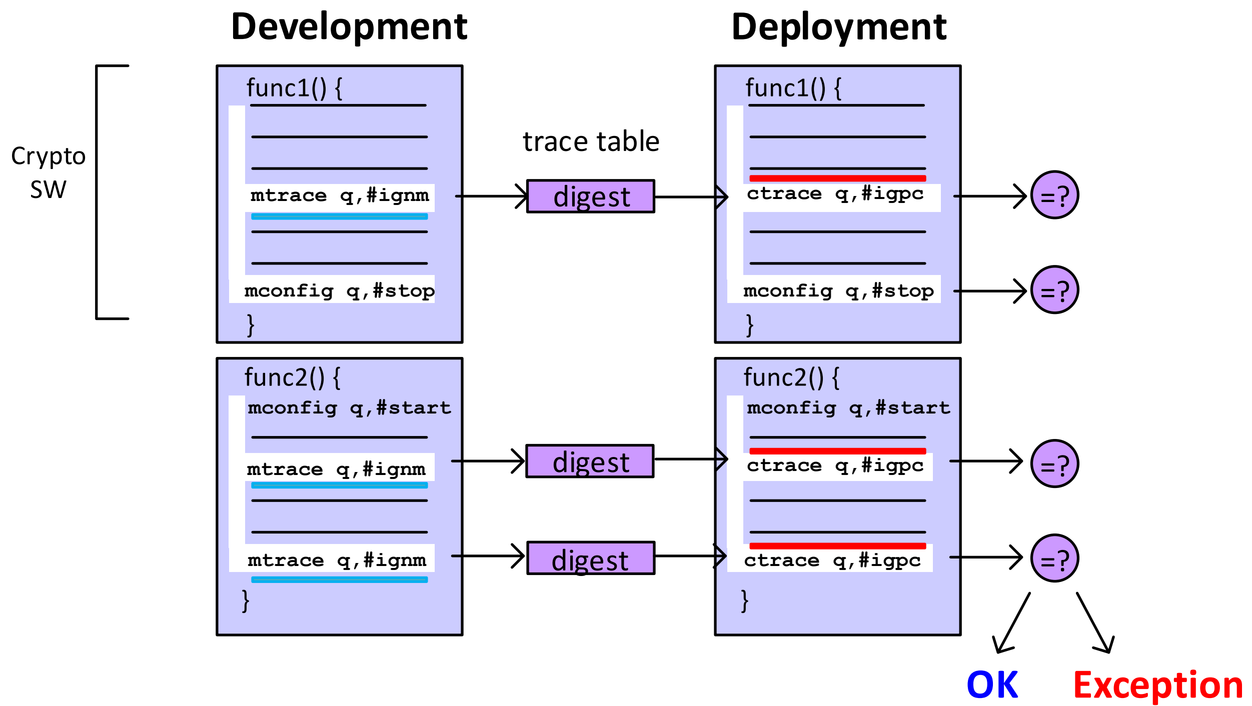

Figure 3.9: Handling Intra-Procedural Control flow

program, then the shape of the control flow graph can be captured in the final digest. Also, the intermediate digests will represent the valid control flow at that particular instant in the program. Figure 3.9 shows, how the digests can be chained using `mtrace/ctrace` instructions across function boundaries. `mconfig` can also create unique checkpoints inside function blocks, that can mark that functions are executed in a particular order.

3.6 Results

3.6.1 Simulation Setup

We have designed the proposed integrity check methodology on GEM5 architectural simulator using Simple CPU model. We simulated a single core ARM CPU at 1 GHz with an L1 I and D cache of 32KB and L2 cache of 2MB. For generating the executable binaries we used cross-compiled GCC-4.9.2 and added the instruction encodings to ARM specific part of the GNU assembler. While choosing the instruction encodings, we followed the encoding format for NEON instructions and made sure that the instruction codes do not conflict with the existing NEON and ARM instructions. To show the effectiveness of our integrity check mechanism towards attacks, we protected a simple SHA3-512 application using `mtrace/ctrace` instructions. We modified GEM5 framework for injecting data and control flow errors and modified the performance measurement unit (PMU) and added counters for measuring injected and detected number of errors.

3.6.2 Attack Detection

Table 3.5 shows the number of integrity violations detected on a protected SHA3-512 application. The results are taken for 12,000 randomly injected errors that disrupts the control and data flow of the algorithm. Our proposed method was able to detect 79% of the total errors and 21% of the undetected errors were custom instruction malfunction errors, which

Table 3.5: Tamper detection in SHA3-512 application

Integrity Violation	Injected	Detected	Detection Rate(%)
Instruction Replacement	4000	4000	100
Instruction Skip	4000	4000	100
Custom Instruction malfunction	4000	1429	36
Total	12000	9429	79

could not be detected by single bit parity prediction scheme. However, it's not a limitation of the design. Better error detection schemes like multi-bit parity and Berger codes can be used to improve the detection probability of such errors.

3.6.3 Performance Overhead

Figure 3.10 show the performance overhead of the proposed approach on a SHA3-512 application implemented using Keccak custom instructions and NEON instruction set. A total of 49 `ctrace` and 9 `mconfig` instructions were embedded and the hash was chained across all the functions of the application. This resulted in a performance overhead of 6% over an unprotected implementation based on Keccak instructions. However, the protected kernel still outperformed a NEON based hand-optimized implementation. We expect a very similar performance overhead of the instructions on a real hardware platform as well. From performance point of view, only `mconfig` and `ctrace` instructions are important as they will be used in the deployed code. To achieve low performance single cycle overhead, both `mconfig` and `ctrace` instructions can operate asynchronously with the actual pipeline using the FIFO

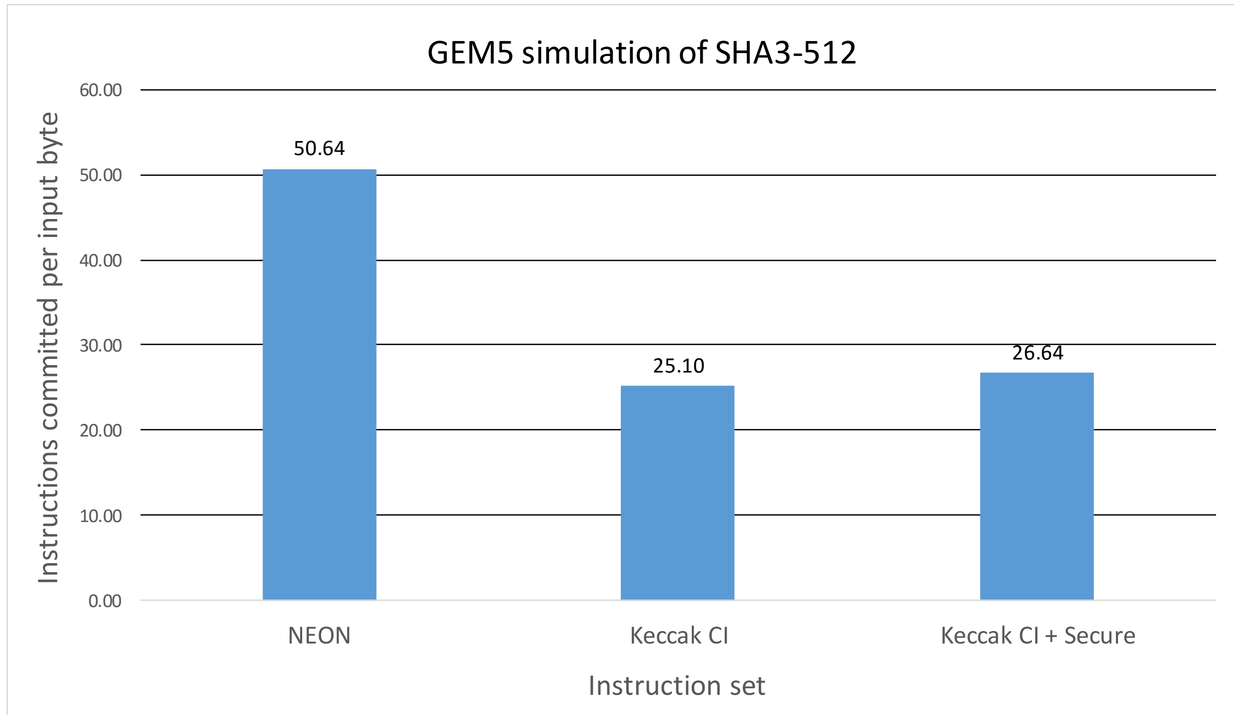

Figure 3.10: Performance overhead of the proposed method for SHA3-512

queue. The configuration settings of `mconfig` is applicable only to the instructions after it in the FIFO queue and, `ctrace` comparison and exceptions can also take place with a delay without halting the main pipeline. Thus, our integrity check mechanism can provide good level of tamper-proof resistance to a cryptographic application at a nominal performance overhead.

3.7 Portability and Future work

Our current design is built on a 128-bit NEON SIMD instruction set supported on ARM processors. These instructions adhere to specifications in the ARM architectural manual and use 32-bit encoding very similar to immediate versions of existing `VBIC` and `VORR` instructions. The instructions use 128-bit wide quad registers and can be easily ported to any processor that support 128-bit registers and in-order commit. For 32-bit and 64-bit machines, the `mtrace/ctrace` instructions can just use the first 32-bit or 64-bit of GHASH.

We did not investigate the hardware overhead of the proposed methodology in this study but we plan to do so in the future. Our present framework measures the flow of instruction stream and custom instruction functional integrity, and tries to capture the control flow and register level data flow of the program using these measurements. In the area of software verification, data flow integrity also captures the read and write operations on memory locations which is a limitation of our current design. In general, data flow integrity can be achieved by tagging the memory location on write operation and verifying the integrity of the tag during read as shown in [11]. We plan to add support for memory tagging in the monitor unit to improve the robustness of our integrity check mechanism. In the future, we also aim to implement better ways to detect functional correctness of the custom logic and detection of hardware Trojans to mitigate the security risks associated with the extensible processors.

Chapter 4

Side Channel and Fault Attacks

Our current work did not investigate the vulnerability of custom instruction based KECCAK implementations against side channel and fault attacks. In general, KECCAK can be implemented in a way such that its execution time is independent of input message. KECCAK also do not use large table lookups, so it's not vulnerable to timing and cache based side channel attacks [2]. Since, our implementations process data similarly as the processor with base instruction set, and do not use any large table lookups, they should not be vulnerable to timing and cache-based attacks. However, KECCAK modes that use secret keys are vulnerable to power and EM analysis based attacks. Possible side-channel countermeasure techniques like masking as discussed in [2] can be used to protect KECCAK software implementations against these attacks. However, in our current work we did not employ any masking techniques to any KECCAK implementation. Similarly, our proposed integrity check mechanism was designed for a threat model that do not include engineered hardware faults.

Numerous solutions can be found in the literature that protect the integrity of hardware [30] against engineered fault attacks.

Chapter 5

Conclusion

In this thesis we present a thorough analysis of the instruction set design space for KECCAK primitives. We have proposed a set of six custom instructions based on NEON instruction set in ARMv7 ISA, which can accelerate KECCAK-f and KECCAK-p primitives of size 1600, 800, 400 and 200 bits. To demonstrate the flexibility of our extensions we implemented five different KECCAK applications: SHA3, LAKE KEYAK, RIVER KEYAK, KETJE SR and KETJE JR and we analyzed the portability of the proposed instructions on Intel AVX and generic 64-bit and 32-bit platforms. To protect these cryptographic applications against tampering in the software or configurable hardware, we propose a methodology based on trusted instructions that enables a library developer to check the integrity of his cryptographic library on an end user's system in a transparent and efficient way. To support our claims, we show performance improvements in instructions committed per byte of input data, using cycle accurate GEM5 simulator. We show that with the proposed extensions, KECCAK ap-

plications can achieve a performance gain between 1.4 - 2.6× over hand optimized assembly. And they can be protected against tampering at a nominal performance overhead of 6%. We also provide the hardware cost of the proposed extensions for accelerating Keccak in GE using UMC′s 90nm technology and show they can be integrated in a Cortex-A9 processor at a nominal overhead of just 4658 GEs.

Bibliography

[1] ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition Issue C. online at `infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html`, 2014.

[2] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, , and R. V. Keer. KECCAK implementation overview. online at `http://keccak.noekeon.org/Keccak-implementation-3.2.pdf`, May 2012.

[3] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Sponge functions. Ecrypt Hash Workshop, May 2007.

[4] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Sponge-Based Pseudo-Random Number Generators. In S. Mangard and F.-X. Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2010.

[5] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography*

(SAC), 2011.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On the security of the keyed sponge construction. Symmetric Key Encryption Workshop (SKEW), February 2011.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. The KECCAK reference. online at `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`, January 2011.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. CAESAR submission: Ketje v2. online at `http://ketje.noekeon.org/Ketje-1.1.pdf`, March 2014.

[9] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. CAESAR submission: Keyak v2. online at `http://keyak.noekeon.org/Keyak-2.1.pdf`, December 2015.

[10] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer. The Keccak Code Package. online at `https://github.com/gvanas/KeccakCodePackage`, 2016.

[11] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160. USENIX Association, 2006.

[12] J. Constantin, A. Burg, and F. K. Gürkaynak. Investigating the potential of custom instruction set extensions for SHA-3 candidates on a 16-bit microcontroller architecture. *IACR Cryptology ePrint Archive*, 2012:50, 2012.

[13] Intel. Corporation. Intel 64 and IA-32 Architectures Software Developers Manual. online at `http://download.intel.com/design/processor/manuals/253665.pdf`, May 2011.

[14] Samsung. Corporation. Samsung Foundry 32/28nm Low-Power High-K Metal Gate Logic Process and Design Ecosystem. online at `http://www.samsung.com/us/business/oem-solutions/pdfs/Foundry_32-28nm_Final_0311.pdf`, March 2011.

[15] I. Dinur, O. Dunkelman, and A. Shamir. New attacks on keccak-224 and keccak-256. In *Fast Software Encryption*, pages 442–461. Springer, 2012.

[16] M. J. Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Federal Inf. Process. Stds. (NIST FIPS) - 202, August 2015.

[17] S. Gueron. Intel advanced encryption standard (aes) new instruction set), howpublished = Intel White Paper, year = 2010, month = May,.

[18] S. Gulley, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich. Intel sha extensions – new instructions supporting the secure hash algorithm on intel architecture processor. Intel White Paper, July 2013.

[19] G. Hoglund and G. Mcgraw. Exploiting software: How to break code. 2004.

[20] B. Jungk and J. Apfelbeck. Area-efficient fpga implementations of the sha-3 finalists. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 235–241, Nov 2011.

[21] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. Cryptographically enforced control flow integrity. *arXiv preprint arXiv:1408.1451*, 2014.

[22] D. McGrew and J. Viega. The galois/counter mode of operation (gcm). *Submission to NIST. http://csrc. nist. gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec. pdf*, 2004.

[23] R. N. Pittman, N. L. Lynch, R. Forin, R. N. Pittman, N. L. Lynch, and R. Forin. emips, a dynamically extensible processor. Technical report, 2006.

[24] K. Seto and M. Fujita. Custom instruction generation with high-level synthesis. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 14–19, June 2008.

[25] Y. Shin, K. Shin, P. Kenkare, R. Kashyap, H. J. Lee, et al. 28nm high- metal-gate heterogeneous quad-core cpus for high-performance and energy-efficient mobile application processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*, pages 154–155, Feb 2013.

[26] P. Swierczynski, M. Fyrbiak, P. Koppe, A. Moradi, and C. Paar. Interdiction in practice hardware trojan against a high-security usb flash drive. Cryptology ePrint Archive, Report 2015/768, 2015. http://eprint.iacr.org/2015/768.

[27] Y. Wang, Y. Shi, C. Wang, and Y. Ha. Fpga-based sha-3 acceleration on a 32-bit processor via instruction set extension. In *Electron Devices and Solid-State Circuits (EDSSC), 2015 IEEE International Conference on*, pages 305–308, June 2015.

[28] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi. The risc-v instruction set manual. volume 1: User-level isa, version 2.0. Technical report, DTIC Document, 2014.

[29] P. Yalla, E. Homsirikamol, and J. Kaps. Comparison of multi-purpose cores of keccak and AES. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 585–588, 2015.

[30] B. Yuce, N. F. Ghalaty, C. Deshpande, C. Patrick, L. Nazhandali, and P. Schaumont. Fame: Fault-attack aware microprocessor extensions for hardware fault detection and software fault response. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, pages 8:1–8:8, New York, NY, USA, 2016. ACM.

[31] B. Zimmer, Y. Lee, A. Puggelli, J. Kwak, R. Jevtic, B. Keller, S. Bailey, M. Blagojevic, P. Chiu, H. Le, P. Chen, N. Sutardja, R. Avizienis, A. Waterman, B. C. Richards, P. Flatresse, E. Alon, K. Asanovic, and B. Nikolic. A RISC-V vector processor with simultaneous-switching switched-capacitor DC-DC converters in 28 nm FDSOI. *J. Solid-State Circuits*, 51(4):930–942, 2016.