RESOURCE-EFFICIENT CRYPTOGRAPHY FOR UBIQUITOUS COMPUTING

Lightweight Cryptographic Primitives from a Hardware & Software Perspective

DISSERTATION

for the degree of Doktor-Ingenieur
of the Faculty of Electrical Engineering and Information Technology
at the Ruhr University Bochum, Germany

by Elif Bilge Kavun
Bochum, December 2014
Anneme ve babama...
Elif Bilge Kavun
Place of birth: İzmir, Turkey
Author’s contact information:
elif.kavun@rub.de
www.emsec.rub.de/chair/staff/elif_bilge_kavun

Thesis Advisor: Prof. Dr.-Ing. Christof Paar
Ruhr-Universität Bochum, Germany

Secondary Referee: Prof. Christian Rechberger
Danmarks Tekniske Universitet, Denmark

Thesis submitted: December 19, 2014
Thesis defense: February 6, 2015
Abstract

Technological advancements in the semiconductor industry over the last few decades made the mass production of very small-scale computing devices possible. Thanks to the compactness and mobility of these devices, they can be deployed “pervasively”, in other words, everywhere and anywhere – such as in smart homes, logistics, e-commerce, and medical technology. Embedding the small-scale devices into everyday objects pervasively also indicates the realization of the foreseen “ubiquitous computing” concept. However, ubiquitous computing and the mass deployment of the pervasive devices in turn brought some concerns – especially, security and privacy.

Many people criticize the security and privacy management in the ubiquitous context. It is even believed that an inadequate level of security may be the greatest barrier to the long-term success of ubiquitous computing. For ubiquitous computing, the adversary model and the security level is not the same as in traditional applications due to limited resources in pervasive devices – area, power, and energy are actually harsh constraints for such devices. Unfortunately, the existing cryptographic solutions are generally quite heavy for these ubiquitous applications. In order to address the security problem of the resource-constrained devices, “lightweight cryptography” has been defined over a decade ago and many different lightweight cryptographic primitives have already been proposed. The published work so far mostly deals with hardware cost reduction. However, this is not the only important metric for such devices. Depending on the application, resource-constrained devices may need lightweight ciphers to be executed in one clock cycle, which still achieve a certain security level and a small footprint. Furthermore, as most of the pervasive computing applications are implemented in software on embedded microcontrollers, there is also a need for lightweight ciphers that result in efficient code size and execution time.

In this thesis, we understand lightweight cryptography also as “resource-efficient cryptography” and we aim to provide new “resource-efficient” solutions for resource-constrained devices, which address the mentioned gaps in lightweight cryptography. We start with initial investigations on existing lightweight primitives, where we present efficient implementations on different platforms, their applications, and comparisons. In the light of our initial investigations, we first propose a new low-latency and low-area lightweight block cipher PRINCE. Following PRINCE, we change our direction to the software side – targeting the software implementations on microcontrollers. As a first step, we come up with a hardware/software co-design approach, the Non-linear/Linear Unit (NLU) Instruction Set Extension (ISE), which targets the 8-bit AVR instruction set of widely-used Atmel microcontrollers. After that, we extend our approach more on the primitive design side, where we define another new lightweight cipher, the “software-oriented” lightweight cipher PRIDE.

In addition to our contributions on efficient lightweight primitive implementations presented in the first part of this thesis, the two novel lightweight block cipher designs achieve the targets and present the best academic results published so far. In the ISE design, our good results
encourage further block cipher extensions on different microcontrollers in order to get a better code size and execution time. However, it is of course not easy to overcome all the gaps in lightweight cryptography in one work. Therefore, other designs and solutions addressing different metrics still remain as an open research problem left for future works.

**Keywords.**
Lightweight cryptography, resource-efficient cryptography, design, ubiquitous computing, symmetric-key cryptography, block cipher, hash function, hardware implementation, software implementation, Application Specific Integrated Circuit (ASIC), Field Programmable Gate Array (FPGA), microcontroller, Atmel AVR, IT Security
Kurzfassung

Ressourcen-effiziente Kryptographie für Ubiquitous Computing: Kryptographische Primitive aus einer Hardware & Software Perspektive


In dieser Arbeit bezeichnen wir die Hocheffiziente Kryptographie auch als “Ressourcen-effiziente Kryptographie” und wir wollen neue “Ressourcen-effiziente” Lösungen für ressourcenbeschränkte Geräte bieten, die die genannten Lücken adressieren. Wir beginnen mit Untersuchungen von bestehenden Primitiven, deren Charakteristika wir auf verschiedenen Plattformen präsentieren. Angesichts unserer ersten Untersuchungen schlagen wir zunächst eine neue Blockchiffre mit dem Namen PRINCE vor, die besonders wenig Chipfläche und Ausführungszeit benötigt. Danach zielen wir auf Softwareimplementierungen auf Mikrocontrollern. Der erste Schritt in diese Richtung ist ein Hardware/Software Codesign genannt NLUISE, das sich an


Schlagworte.

Hocheffiziente Kryptographie, Ressourcen-effiziente Kryptographie, Entwurf, Ubiquitous Computing, symmetrische Kryptographie, Blockchiffre, Hashfunktion, Hardware Implementierung, Software Implementierung, [ASIC] [FPGA] Mikrocontroller, Atmel AVR, IT-Sicherheit
This thesis is the outcome of wonderful three and a half years at the Embedded Security (EmSec) Group of the Horst Görtz Institute for IT-Security (HGI), Ruhr University Bochum (RUB). EmSec was always like (sometimes literally) a second home to me. The warm, friendly, and inspiring environment of EmSec (and of course, SHA\(^1\)) group was always motivating, thanks a lot to you all!

There are many people that I would like to express my gratitude, without whom the results of this thesis would not be possible. First of all, I would like to thank my supervisor Christof Paar for giving me this Ph.D. opportunity. I have always felt very lucky for being his student, he is open to new ideas and provides the research freedom a student needs. He always encouraged networking, which gave me the opportunity to meet and work with the leading researchers in IT Security field – I learned a lot from them! Secondly, I am very grateful to Tolga Yalçın and Gregor Leander for their friendly and cooperative guidance during my Ph.D. – they have been great teachers and mentors. Special thanks to Christian Rechberger, for being the secondary referee of my thesis and the fruitful research projects we had. My co-authors (in alphabetical order) – Martin. R. Albrecht, Lejla Batina, Andrey Bogdanov, Julia Borghoff, Anne Canteaut, Amitabh Das, Benedikt Diriessen, Barış Ege, Susanne Engels, Tim Güneysu, Miroslav Knežević, Lars R. Knudsen, Nele Mentens, Hristina Mihajloska, Oliver Mischke, Ventislav Nikov, Thomas Pöppelmann, Peter Rombouts, Søren S. Thomsen, Elmar Tischhauser, Ingrid Verbauwhede; it was an honor working with you, thanks a lot!

Many thanks to our project partner NXP for pointing us to the exciting low-latency cipher design idea, which is now a very important part of this thesis. Furthermore, I would like to thank the UbiCrypt (Ubiquitous Cryptography) Research Training Group (DFG Graduiertenkolleg) for funding my research, and supporting me throughout my Ph.D. with many personal and career development seminars and workshops. All UbiCrypt members, it was great to work with you, thanks a lot for all the comments and productive discussions. Thanks to all the people I met during my internship at Qualcomm, it was a great experience working with you!

Special thanks to our team assistant Irmgard Kühn and our technical assistant Horst Edelmann, for their kind help with all administrative and technical stuff. Also, thanks a lot to our UbiCrypt coordinator Dominik Baumgarten for answering all my UbiCrypt-related questions and especially for his help during CrossFyre workshop organization.

My office-mate Christian Zenger – it was awesome to share an office with you, keep the awesomeness! Vielen Dank an Benedikt, especially for helping me with the translation of the thesis abstract into German.

All the loved ones; my family and friends – you provided me the support I need all the way along... This work would not be possible without you, thank you for being there!

Last but not the least, to all other people I forgot to mention here: You know I owe you, thanks!

\(^1\)Sichere Hardware Arbeitsgruppe
Table of Contents

Imprint ................................................................. V
Abstract ............................................................... vii
Kurzfassung .......................................................... ix
Acknowledgements ..................................................... xi

I Preliminaries ......................................................... 1

1 Introduction ......................................................... 3
  1.1 Ubiquitous Computing and Security Concerns .................. 3
  1.2 Resource-constrained (Lightweight) Cryptography ......... 4
  1.3 Motivation ...................................................... 5
  1.4 Contributions and Structure of the Thesis ................. 6

2 Overview ............................................................ 9
  2.1 Cryptological Background .................................... 9
  2.2 Block Ciphers ................................................. 9
    2.2.1 Substitution-Permutation Network (SPN) ............. 10
    2.2.2 Notation and Mathematical Background ............. 11
  2.3 Hash Functions ............................................... 13

3 Tools ................................................................. 15
  3.1 Hardware Platforms .......................................... 15
    3.1.1 ASIC .................................................. 15
    3.1.2 Reconfigurable Hardware – FPGA ................... 16
  3.2 Software Platforms .......................................... 17
    3.2.1 ATmega8A ............................................. 17
    3.2.2 STM32F4DISCOVERY ................................. 17
    3.2.3 EnergyMicro EF32GG-STK3700 ....................... 17

II Resource-efficient Implementations of Existing Ciphers .... 19

4 An Analysis of Recently Developed Lightweight Block Ciphers 21
  4.1 Introduction .................................................. 21
  4.2 Background .................................................. 22
    4.2.1 Fully-parallel Implementations ..................... 22
Table of Contents

4.2.2 Implementation of KATAN ............................................. 24
4.3 Analysis Strategy ......................................................... 24
   4.3.1 Architectural Decisions ........................................... 24
   4.3.2 Evaluation of Design Parameters .................................. 25
4.4 Results ........................................................................... 25
4.5 Conclusions and Future Work ............................................ 26

5 Random Access Memory (RAM)-Based Ultra-Lightweight FPGA Implementation of PRESENT 29
   5.1 Introduction .................................................................. 29
   5.2 Previous Work ............................................................. 30
   5.3 The Proposed PRESENT Implementations on FPGA ............ 30
       5.3.1 State Processing ................................................... 31
       5.3.2 Key Schedule ...................................................... 32
       5.3.3 Final Round ....................................................... 33
       5.3.4 Overall Design .................................................... 34
   5.4 Results and Discussion ................................................ 37
   5.5 Conclusion and Future Work ......................................... 37

6 On the Suitability of SHA-3 Finalists for Lightweight Applications 39
   6.1 Introduction .................................................................. 39
   6.2 Implementations of the Finalist Hash Functions ................. 40
       6.2.1 BLAKE ............................................................... 40
       6.2.2 Grøstl ............................................................... 44
       6.2.3 JH .................................................................. 46
       6.2.4 Keccak ............................................................... 52
       6.2.5 Skein ............................................................... 55
   6.3 Interface ....................................................................... 59
   6.4 Results and Discussion ................................................ 60
   6.5 Conclusion ................................................................. 61

7 IPSecco: A Lightweight and Reconfigurable Internet Protocol Security (IPSec) Core 63
   7.1 Introduction .................................................................. 63
   7.2 Related Work ............................................................. 64
   7.3 Implementation ........................................................... 65
       7.3.1 Message Authentication ........................................... 66
       7.3.2 Authenticated Encryption .......................................... 68
       7.3.3 Key Exchange ...................................................... 71
   7.4 Results and Discussion ................................................ 72
   7.5 Conclusion and Future Work ......................................... 74

III New Resource-efficient Constructions for the Ubiquitous Computing Era 77
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Targeting Low-latency: A Lightweight Block Cipher <strong>PRINCE</strong></td>
<td>79</td>
</tr>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>79</td>
</tr>
<tr>
<td>8.2</td>
<td><strong>PRINCE</strong> Background</td>
<td>81</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Highlights of <strong>PRINCE</strong></td>
<td>81</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Cipher Description</td>
<td>82</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Design Decisions</td>
<td>83</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Security Analysis</td>
<td>87</td>
</tr>
<tr>
<td>8.3</td>
<td>Hardware Investigations</td>
<td>88</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Searching for the Optimal Sbox</td>
<td>88</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Low-cost Linear Layer</td>
<td>89</td>
</tr>
<tr>
<td>8.4</td>
<td>Hardware Implementations</td>
<td>89</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Unrolled Architecture</td>
<td>90</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Round-based Architecture</td>
<td>91</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Performance of Both Architectures on FPGA</td>
<td>92</td>
</tr>
<tr>
<td>8.5</td>
<td>Software Implementations</td>
<td>93</td>
</tr>
<tr>
<td>8.6</td>
<td>Investigations on Countermeasures Against Potential Side-Channel Attacks (SCA) on <strong>PRINCE</strong></td>
<td>94</td>
</tr>
<tr>
<td>8.6.1</td>
<td>Adversary Model</td>
<td>94</td>
</tr>
<tr>
<td>8.6.2</td>
<td>Potential Countermeasures for <strong>PRINCE</strong> Against SCA</td>
<td>95</td>
</tr>
<tr>
<td>8.7</td>
<td>Concluding Remarks</td>
<td>96</td>
</tr>
<tr>
<td>9</td>
<td>Towards Software-efficiency: <strong>NLU</strong> Instruction Set Extension</td>
<td>97</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>97</td>
</tr>
<tr>
<td>9.2</td>
<td>Related Work and the Proposed <strong>ISE</strong> Model</td>
<td>98</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Realization of Substitution</td>
<td>99</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Realization of Permutation</td>
<td>100</td>
</tr>
<tr>
<td>9.2.3</td>
<td>Loading the Substitution and Permutation Coefficients</td>
<td>101</td>
</tr>
<tr>
<td>9.2.4</td>
<td><strong>NLU</strong> Instructions</td>
<td>101</td>
</tr>
<tr>
<td>9.3</td>
<td>Unified Non-linear/Linear Hardware Unit</td>
<td>101</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Non-linear Unit</td>
<td>103</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Linear Unit</td>
<td>103</td>
</tr>
<tr>
<td>9.3.3</td>
<td>Area and Power Consumption of <strong>NLU</strong> Module</td>
<td>103</td>
</tr>
<tr>
<td>9.4</td>
<td>Applications</td>
<td>105</td>
</tr>
<tr>
<td>9.4.1</td>
<td><strong>PRESENT</strong> Software Implementation</td>
<td>105</td>
</tr>
<tr>
<td>9.4.2</td>
<td><strong>CLEFIA</strong> Software Implementation</td>
<td>109</td>
</tr>
<tr>
<td>9.4.3</td>
<td><strong>Serpent</strong> Software Implementation</td>
<td>110</td>
</tr>
<tr>
<td>9.4.4</td>
<td><strong>AES</strong> Software Implementation</td>
<td>110</td>
</tr>
<tr>
<td>9.5</td>
<td>Results and Discussion</td>
<td>114</td>
</tr>
<tr>
<td>9.6</td>
<td>Conclusion and Future Directions</td>
<td>115</td>
</tr>
<tr>
<td>10</td>
<td>Software-efficiency – Going Further: A Lightweight Block Cipher <strong>PRIDE</strong></td>
<td>117</td>
</tr>
<tr>
<td>10.1</td>
<td>Introduction</td>
<td>117</td>
</tr>
<tr>
<td>10.1.1</td>
<td>Our Contribution</td>
<td>118</td>
</tr>
<tr>
<td>10.2</td>
<td>The Interleaving Construction</td>
<td>119</td>
</tr>
<tr>
<td>10.2.1</td>
<td>The General Construction</td>
<td>119</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>List of Tables</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>About the Author</td>
<td>189</td>
<td></td>
</tr>
<tr>
<td>Author’s Publications</td>
<td>191</td>
<td></td>
</tr>
</tbody>
</table>
Part I

Preliminaries
Chapter 1

Introduction

This chapter provides some background information about ubiquitous computing and “resource-constrained” (lightweight) cryptography, gives a brief overview of the available lightweight constructions and implementations, and discusses the unaddressed points in lightweight solutions. Following that, the research motivation of this thesis is explained. Finally, the structure of the thesis is outlined and the research contributions are summarized.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Ubiquitous Computing and Security Concerns</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Resource-constrained (Lightweight) Cryptography</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Motivation</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Contributions and Structure of the Thesis</td>
<td>6</td>
</tr>
</tbody>
</table>

1.1 Ubiquitous Computing and Security Concerns

Modern computing devices have been in our lives now for more than 50 years. At the beginning, they used to cover the size of a large room, and they were consuming as much power as several hundred modern PCs. Over the last few decades, the semiconductor industry has shown great technological advancements, which made the mass production of very small-scale computing devices possible. The compactness, power-efficiency, and mobility of these devices let them to be deployed pervasively, in other words, everywhere and anywhere. For instance, pervasive devices such as embedded microcontrollers, sensors, and terminals can now be found in many different daily life applications – smart homes, logistics, e-commerce, and medical technology are to name but a few of these. In Figure 1.1 chart (b) demonstrates the development of computing devices (a) visually. Embedding the small-scale devices into our everyday objects pervasively also indicates the realization of the foreseen “ubiquitous computing” [Wei99] concept, which is becoming a more important part of our lives day by day.

However, ubiquitous computing and the mass deployment of pervasive devices in turn brought some concerns – especially, security and privacy. Despite the security and privacy attributes mentioned in the ubiquitous computing definition of Massachusetts Institute of Technology (MIT)’s Project Oxygen [MIT], many people criticize the security and privacy management in the ubiquitous context. It is even believed that an inadequate level of security may be the
greatest barrier to the long-term success of ubiquitous computing. However, for ubiquitous computing, the adversary model and the security level is not the same as for traditional applications due to the resource-constrained nature of the small-scale devices. Area, power, and energy are harsh constraints as a result of the limited resources. Unfortunately, many existing “traditional” cryptographic solutions are generally quite heavy for such applications. “Lightweight cryptography” has been defined to provide solutions for the security problems of these embedded devices. As lightweight cryptography is specifically tailored for resource-constrained environments, we will also call it “resource-efficient cryptography” in this thesis.

1.2 Resource-constrained (Lightweight) Cryptography

Lightweight cryptography has been attracting researchers despite being a relatively new topic. The main question of this field of cryptography is achieving high or sufficient levels of security by consuming small area and less computing power, which means being “resource-efficient”. The
focus of lightweight cryptography lies in new lightweight primitive designs and adaptations and efficient implementations of existing cryptographic primitives.

In the area of lightweight cryptography, several primitives for symmetric and asymmetric cryptography, and hash functions have been proposed within the last decade. Among the best-studied symmetric algorithms are the block ciphers CLEFIA [SSA+07], HEIGHT [HS+06], KATAN, KTANTAN [CDK09], KLEIN [GNL11], mCrypton [LK06], LED [GPR11], Piccolo [SH+11], and PRESENT [BK+07], as well as the stream ciphers Grain [HJM08], Mickey [BD08], and Trivium [CP08]. While NTRU being the only lightweight example for asymmetric cryptography (note that Elliptic Curve Cryptography (ECC) [Kob87, Mil86] performs well compared to RSA [RSA78] and public-key cryptosystems based on discrete logarithm problem [PP10a] and is hence used in lightweight applications, but it is not developed specifically for lightweight); PHOTON [GP011], QUARK [AHNP13], and SPONGENT [BK+11] have been proposed as lightweight hash functions. There have also been many implementation/adaptation attempts – different hardware and software implementations of traditional and lightweight proposals have been extensively studied in many versions – speed-optimized, area-optimized, serial, parallel, etc. – by the researchers.

Out of the primitives that we mentioned above, two prominent lightweight block cipher examples, the PRESENT lightweight block cipher and Sony’s CLEFIA, have already been accepted as lightweight cryptography standards by International Organization for Standardization (ISO) (as ISO/IEC Standard 29192-2:2012) in 2012 [ISO]. This standardization also points out industry demand for lightweight ciphers.

Somewhat surprisingly, the focus in the lightweight crypto literature has almost been entirely on hardware-efficiency. However, this is not the only important metric for pervasive devices. Depending on the application, resource-constrained devices may need lightweight crypto primitives that are designed according to other metrics.

1.3 Motivation

So far, metrics other than hardware-efficiency have barely been addressed in lightweight cryptography solutions. While hardware-efficiency is certainly a valid optimization objective, its relevance to real-world applications is limited. Nowadays, there are several interesting and strong proposals available that feature a very small area but simultaneously neglect other, important real-world constraints. For example, depending on the application, resource-constrained devices may need lightweight ciphers to be executed in one clock cycle (e.g., instant authentication and read/write access to memory devices), while still guaranteeing a certain security level and a small footprint. Unfortunately, recent proposals achieve the goal of a small chip area by sacrificing execution speed to such an extent that even in applications where speed is supposedly uncritical, the ciphers are getting too slow. Furthermore, as the vast majority of pervasive devices is equipped with (small) embedded CPUs, as opposed to dedicated ASICs, and the pervasive computing applications – including security applications – are implemented in software on these embedded microcontrollers, there is also a need for lightweight ciphers that result in efficient code size and execution time. Interestingly, lightweight solutions with respect to software-efficiency have barely been addressed so far. Figure 1.2 visualizes this situation in

2See also [KNR12], which asks “Is lightweight = light + wait?”.
lightweight cryptography: There have been numerous block cipher proposals on the hardware area side; however, the other three parts are nearly left unaddressed. The software column depicts the attempts to come up with ciphers partially-tailored for low-cost processors; but except SPECK \(\text{BSS}^+\text{13}\), these proposals generally result in high execution time and/or use a large amount of program memory storage\(^3\).

![Figure 1.2: The gaps in lightweight cryptography](image)

In this thesis, we aim to address the mentioned gaps in lightweight cryptography. We start with initial investigations of existing lightweight primitives and solutions, where we present efficient implementations of them together with comparisons. We also present efficient implementation of ciphers specific to different platforms. In light of our initial investigations, we propose the new low-latency and low-area lightweight block cipher PRINCE, which fills the first column of the table in Figure 1.2. After PRINCE, we change our direction to software-oriented solutions – targeting software implementations on microcontrollers. We first propose a hardware/software co-design approach, the NLU ISE, which targets the 8-bit AVR instruction set of widely-used Atmel microcontrollers. Following that, we extend our approach to the primitive design side, where we try to find solutions for the second column of the table in Figure 1.2. We propose another new lightweight crypto primitive, the “software-oriented” lightweight block cipher PRIDE.

Note that, in this thesis, we will focus on the lightweight block cipher and lightweight hash function parts of lightweight cryptography.

### 1.4 Contributions and Structure of the Thesis

This thesis consists of four parts: Preliminaries (Part I) – together with this introduction, we present a cryptographic overview and a description of our tools in this part; Appetizers (Part II) – here we explain our initial investigations and findings in lightweight cryptography; Resource-efficient Solutions for Ubiquitous Computing (Part III) – we introduce our novel designs in this part; Conclusion (Part IV) – we finally conclude the thesis and outline future directions.

\(^3\)See Section 10.1 for more information.
1.4. Contributions and Structure of the Thesis

Note that, due to the interdisciplinary nature of the thesis topic, we had collaborations with different research groups during the course of this thesis. However, the thesis presents only the contributions of the thesis author. We refer to the original publications where necessary.

In the following, we list the four parts and their chapters together with the respective contributions and the regarding publications.

■ Part I

□ Chapter 2 A general overview of block ciphers and hash functions is presented here, in order to provide the basic knowledge needed for the following chapters.

□ Chapter 3 Here we present the hardware and software tools used during the course of this thesis.

■ Part II

□ Chapter 4 This chapter presents an area, power, and energy analysis of some of the recently developed lightweight block ciphers and compares them to different implementations of the standard AES [NIST] algorithm. This work is a guide for the selection of algorithms based on application requirements. The results of this collaborative work were later published at the International Workshop on RFID Security and Privacy (RFIDSec) in 2013 [BDE+13].

□ Chapter 5 Here we propose two different FPGA implementations of the lightweight cipher PRESENT. Our main design strategy in this chapter is the utilization of existing RAM blocks in FPGAs for the storage of internal states, in order to reduce the slice count. Our results were published at the International Conference on Reconfigurable Computing and FPGAs (ReConFig) in 2011 [KY11].

□ Chapter 6 This chapter investigates the suitability of SHA-3 finalists for lightweight applications. For each finalist, we try to achieve the lowest reported gate count while maintaining a respectable throughput. We mainly favor a word-serial approach in our designs to achieve a low gate count. This work was presented at the 3rd SHA-3 Conference in 2012 [KY].

□ Chapter 7 In this chapter, we propose a reconfigurable lightweight IPSec hardware core, which is implemented on FPGA. We evaluate efficient implementations of standardized and/or well-known lightweight and hardware-friendly algorithms. Here we make use of some of the designs presented in 5 and 6. The results of this collaborative work were published at ReConFig in 2012 [DGK+12].

■ Part III

□ Chapter 8 The design of a new lightweight block cipher PRINCE, which specifically targets low-latency encryption, is investigated in this chapter. The main design target is to keep the gate count below the existing ciphers in the literature, and the latency at exactly one clock cycle. Within the scope of this thesis, we only focus on the specific parts of this design process, on which the author mainly worked. This work was published at ASIACRYPT in 2012 [BCG+12a].

□ Chapter 9 This chapter is a step forward towards the software-efficiency of the cryptographic algorithms. We introduce a non-linear/linear ISE namely NLU which
is capable of implementing non-linear operations, namely Sboxes, expressed in their Algebraic Normal Form (ANF) and linear operations expressed as binary matrix multiplication, “multiply-and-add”, form. We base our NLU design only on the widely-used 8-bit AVR instruction set. Our results were published at IEEE Symposium on Computer Arithmetic (ARITH) in 2013 [EKM+13].

Chapter 10: This chapter focuses on “software-efficient” algorithm design. In order to get an efficient cipher, we have a look at the software-costly SPN component linear layer. On a hardware-accelerated reconfigurable search platform, we search for efficient linear layers regarding software performance. Our software-oriented block cipher proposal PRIDE uses the resulting linear layer. PRIDE is optimized for 8-bit microcontrollers and outperforms all academic solutions both in terms of code size and cycle count. The results of this collaborative work were published at ReConFig in 2013 [KLY13] and at CRYPTO in 2014 [ADK+14a].

Part IV

Chapter 11: Here we summarize our results and discuss the contributions of the thesis.

Chapter 12: In this chapter, we point out the parts that might need further investigation and present some ideas for future research.

Note that the author of this thesis has contributed to other topics, such as hardware acceleration for cryptanalysis and authenticated encryption; however, they are not included in this thesis due to its focus on lightweight cryptography. A detailed list of all publications can be found at the end of this thesis.
Chapter 2
Overview

Lightweight cryptographic primitives possess the properties of traditional crypto primitives but are optimized for resource-constrained platforms. Therefore, here we present a general overview of block ciphers and hash functions, in order to provide the basic knowledge needed for the following chapters.

Contents of this Chapter

2.1 Cryptological Background ................................. 9
2.2 Block Ciphers .............................................. 9
2.3 Hash Functions ............................................ 13

2.1 Cryptological Background

Cryptology consists of two main parts, one being the effort of building primitives – cryptography, the other being the effort of analyzing these primitives – cryptanalysis. Actually, in the ideal case, these two have a mutual relationship: Cryptanalysis investigates the security of a primitive and provides feedback; then cryptography comes up with a more secure primitive design, which gives cryptanalysis the chance to apply better analysis techniques to break the primitive. In this thesis, our focus is on cryptographic primitives. As depicted in Figure 2.1 it has three main branches; symmetric (secret-key) crypto, asymmetric (public-key) crypto, and hash functions. Within the scope of this thesis, we are interested in the symmetric-key primitives and hash functions.

Block ciphers and stream ciphers are the basic building blocks of symmetric cryptography. Out of these, we will give further information on block ciphers in the following.

2.2 Block Ciphers

Block ciphers are one of the most prominently used cryptographic primitives and account for the largest portion of data encrypted today. This was facilitated by the introduction of Rijndael [DR98] as the AES, which was a major step forward in the field of block cipher design. Not only does AES offer strong security, but its structure also inspired many cipher designs ever since.

There are two main design strategies that can be identified for block ciphers: Sbox-based constructions and constructions without Sboxes. Sbox-less constructions are most prominently
those using addition, rotation, and Exclusive OR (XOR) (Addition-Rotation-XOR (ARX) designs). Furthermore, Sbox-based designs can be split into Feistel-ciphers and SPN. Both concepts have been successfully used in practice, the most prominent example of an SPN cipher being AES and the most prominent Feistel-cipher being the former DES [NIS99]. In the scope of this thesis, we will focus on SPN constructions.

### 2.2.1 SPN

An SPN is an iterated cipher structure where the round function is built from invertible function layers such as substitution and permutation, as can be seen in Figure 2.2.

![Figure 2.2: Illustration of SPN](image)

Besides the most prominent example AES, it is also worth mentioning that the concept of SPN has not only been used in the design of block ciphers but also for designing crypto-
2.2. Block Ciphers

graphic permutations, most prominently for the design of several sponge-based hash functions including SHA-3 [NIS14]. In Substitution-Permutation (SP) networks, the round function consists of a non-linear layer composed of small Sboxes working in parallel on small chunks of the state, and a linear layer that mixes those chunks. Thus, designing an SPN block cipher essentially reduces to choosing one (or several) Sboxes and a linear layer.

Substitution (Non-linear) Layer

Substitution is the non-linear part of a cipher and introduces confusion. It generally consists of the parallel application of functions $S: \{0,1\}^m \rightarrow \{0,1\}^m$ that operate on small chunks of data. The $S$ functions are called Sboxes (Substitution boxes) and can be implemented as a lookup table with $2^m$ entries. Note that the input and output size of Sboxes may also be different. However, in such cases the Sbox is not bijective.

A lot of research has been devoted to the study of Sboxes. All Sboxes of size up to 4 bits have been classified (indeed, more than once – cf. [BCBP03, LP07, Saa11]). Moreover, Sboxes with optimal resistance against differential and linear attacks have been classified up to dimension 5 [BL08]. In general, several constructions are known for good and optimal Sboxes in arbitrary dimensions. Starting with the work of Nyberg [Nyb93], this has evolved into its own field of research in which those functions are studied in great detail. A nice survey of the main results of this line of study is provided by Carlet [Car10].

Permutation (Linear) Layer

The permutation layer is an affine transformation that operates on the complete block and introduces diffusion. A well-chosen linear layer is not only crucial for the security and efficiency of a block cipher, but also allows to argue in a simple and thereby convincing way about its security. AES and its predecessor SQUARE [DKR97] are good examples of that.

Research efforts have been focused on Sboxes as mentioned in the previous section; however, the situation for the linear layer is less clear. For the design of the linear layer, two general approaches can be identified. One still widely-used method is to design the linear layer in a rather ad-hoc fashion, without following general design guidelines. While this might lead to very secure and efficient algorithms (cf. Serpent [ABK98] and SHA-3 as prominent examples), it is not very satisfactory from a scientific point-of-view. The second general design strategy is the wide-trail strategy introduced by Daemen in [Dae95] (see also [DR01]). Especially for the security against linear [Mat93] and differential [BS90] attacks, the wide-trail strategy usually results in simple and strong security arguments. It is therefore not surprising that this concept has found its way in many recent designs (e.g., Khazad [BR00a], Anubis [BR00a], Grøstl [GKM+11], PHOTON, LED, PRINCE, mCrypton to name but a few). In a nutshell, the main idea of the wide-trail strategy is to link the number of active Sboxes for linear and differential cryptanalysis to the minimal distance of a certain linear code associated with the linear layer of the cipher. In turn, choosing a good code (with some additional constraints) results in a large number of active Sboxes.

2.2.2 Notation and Mathematical Background

In this part, we fix some basic notation and furthermore recall the ideas of the wide-trail strategy.
Chapter 2. Overview

We deal with SPN block ciphers where the Sbox layer consists of \( n \) Sboxes of size \( b \) each. Thus the block size of the cipher is \( n \times b \). The linear layer will be implemented by applying \( k \) binary matrices in parallel.

We denote by \( \mathbb{F}_2 \) the field with two elements and by \( \mathbb{F}_2^n \) the \( n \)-dimensional vector space over \( \mathbb{F}_2 \). Note that any finite extension field \( \mathbb{F}_{2^b} \) over \( \mathbb{F}_2 \) can be viewed as the vector space \( \mathbb{F}_2^b \) of dimension \( b \). Along these lines, the vector space \( (\mathbb{F}_2^b)^n \) can be viewed as the (nested) vector space \( (\mathbb{F}_2^b)^n \).

Given a vector \( \mathbf{x} = (x_1, \ldots, x_n) \in (\mathbb{F}_2^b)^n \), where each \( x_i \in \mathbb{F}_2^b \), we define its weight\(^1\) as

\[
\text{wt}_b(\mathbf{x}) = |\{1 \leq i \leq n \mid x_i \neq 0\}|
\]

Following [DR01], given a linear mapping \( L : (\mathbb{F}_2^b)^n \rightarrow (\mathbb{F}_2^b)^n \), its differential branch number is defined as

\[
B_d(L) := \min \{\text{wt}_b(\mathbf{x}) + \text{wt}_b(L(\mathbf{x})) \mid \mathbf{x} \in (\mathbb{F}_2^b)^n, \mathbf{x} \neq 0\}
\]

The cryptographic significance of the branch number is that the branch number corresponds to the minimal number of active Sboxes in any two consecutive rounds. Here an Sbox is called active Sbox if it gets a non-zero input difference in its input.

Given an upper bound \( p \) on the differential probability for a single Sbox along with a lower bound of active Sboxes immediately allows to deduce an upper bound for any differential characteristic\(^2\) using average probability for any non-trivial characteristic \( \leq p \# \text{active Sboxes} \).

For linear cryptanalysis, the linear branch number is defined as

\[
B_l(L) := \min \{\text{wt}_b(\mathbf{x}) + \text{wt}_b(L^*(\mathbf{x})) \mid \mathbf{x} \in (\mathbb{F}_2^b)^n, \mathbf{x} \neq 0\}
\]

where \( L^* \) is the adjoint linear mapping. That is, with respect to the standard inner product, \( L^* \) corresponds to the transposed matrix of \( L \).

In terms of correlation (cf., for example, [Dae95]), an upper bound \( c \) on the absolute value of the correlation for a single Sbox results in a bound for any linear trail (or linear characteristic, linear path) via

\[
\text{absolute correlation for a trail} \leq c \# \text{active Sboxes}.
\]

The differential branch number corresponds to the minimal distance of the \( \mathbb{F}_2 \)-linear code \( C \) over \( \mathbb{F}_2^b \) with generator matrix

\[
G = [I \mid L^T]
\]

where \( I \) is the \( n \times n \) identity matrix. The length of the code is \( 2n \) and its dimension is \( n \) (here dimension corresponds to \( \log_2(|C|) \) as it is not necessarily a linear code). Thus, \( C \) is a \((2n, 2^n)\) additive code over \( \mathbb{F}_2^b \) with minimal distance \( d = B_d(L) \).

The linear branch number corresponds in the same way to the minimal distance of the \( \mathbb{F}_2 \)-linear code \( C^\perp \) with generator matrix

\[
G^* = [L \mid I].
\]

---

\(^1\)Of course \((\mathbb{F}_2^b)^n\) is isomorphic to \(\mathbb{F}_2^{nb}\), but the weight is defined differently on each.

\(^2\)Averaging over all keys, assuming independent round keys.
2.3 Hash Functions

Note that $C^\perp$ is the dual code of $C$ and in general the minimal distances of $C^\perp$ and $C$ do not need to be identical.

Finally, given linear maps $L_1$ and $L_2$, we denote by $L_1 \times L_2$ the direct sum of the mappings, i.e.

$$(L_1 \times L_2)(x, y) := (L_1(x), L_2(y)).$$

2.3 Hash Functions

Hash functions are vital cryptographic primitives and they are used in many protocols. A hash function computes a digest of a message, which is a short and fixed-length bit-string. Unlike other cryptographic algorithms, hash functions do not use key. In order to be secure, hash functions need to possess three main properties, namely pre-image resistance (one-wayness), second pre-image resistance (weak collision resistance), and collision resistance (strong collision resistance) [PP10b]. Hash functions are used in applications such as digital signatures, Message Authentication Code (MAC)s, key derivation, etc. The primary application of hash functions in cryptography is message integrity. The hash value provides a digital fingerprint of message contents, ensuring that the message is not changed by an adversary. Hash algorithms are effective because of the extremely low probability that two different plaintext messages will yield the same hash value. There are several well-known hash functions in use today, e.g., MD5 [MD5], National Institute of Standards and Technology (NIST) standards SHA-1, SHA-256 [NIST] and new SHA-3, PHOTON, and SPONGENT to name but a few of these.
Chapter 3

Tools

In this chapter, we present the hardware and software tools used during the course of this thesis. We use the hardware and software tools mainly for two reasons – as implementation platform, and as search platform. Here, we explain the use case of each of these tools, and briefly give their properties.

Contents of this Chapter

<table>
<thead>
<tr>
<th>3.1 Hardware Platforms</th>
<th>3.2 Software Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>17</td>
</tr>
</tbody>
</table>

3.1 Hardware Platforms

In our work, we have used hardware platforms for two basic purposes; as implementation platform, or as hardware accelerator in order to realize our search algorithms. In the former case, we have used both ASIC and FPGA platforms. However, for the latter case, we made use of only FPGA platforms, due to the reconfigurability they offer.

In the following, we give some information on the hardware tools we have used in our research.

3.1.1 ASIC

In our work, we implemented many different hardware architectures of the existing and new lightweight cryptographic primitives. In the implementation process, Mentor Graphics Modelsim [Gra] and Cadence NCVerilog 06.20-p001 [Cad] are used for functional verification/simulation. The RTL description is then synthesized with Cadence Encounter RTL Compiler v10.1 [Cad13], which was also used to generate the area, timing, and power estimation reports.

Since the gate count and delay parameters are heavily technology dependent, the implementations have been synthesized for different standard-cell technology libraries. Three different technology libraries are used throughout this thesis: 130 nm and 90 nm low-leakage Faraday Libraries from UMC [UMCa, UMCb], and 45 nm generic NANGATE [Nan] Open Cell Library. All ciphers are implemented in Verilog-Hardware Description Language (HDL) and typical operating conditions were assumed in all syntheses.

Area results are depicted in $\mu m^2$ in area reports; however, this value is highly-dependent on the fabrication technology and the standard-cell library used in the synthesis. Therefore, in order to have fair comparisons, chip area is typically measured in Gate Equivalence (GE), i.e., the cipher area normalized to the area of a two-input NAND gate in a given standard-cell
library. The area in \( GE \) is derived by dividing the silicon area of the implementation by the silicon area of the two-input NAND gate in the same technology library.

### 3.1.2 Reconfigurable Hardware – FPGA

Although ASICs still have lower costs in large volumes, for customers willing to produce small amounts of sensor nodes or Radio Frequency IDentification (RFID)s, low-cost and low-power FPGA\textsuperscript{s} seem to be the ideal solution. With the latest advances in the FPGA technologies and architectures, they are expected to be preferable for battery-powered applications like wireless sensor nodes, presenting a popular platform for lightweight applications. Furthermore, the reconfigurable nature of FPGA\textsuperscript{s} makes them even more attractive in applications that may require regular hardware updates and/or modifications. Therefore, some of our implementations are also synthesized on FPGA\textsuperscript{s}.

In addition to the regular implementations, we also implemented our reconfigurable search architecture (see Chapter 10) on the FPGA\textsuperscript{platform}. Using an FPGA\textsuperscript{was indispensable due to the reconfigurability and flexibility it provided, unlike an ASIC\textsuperscript{platform. This was important as we needed to modify the search parameters and implement architectural modifications easily.}

As the search platform, we used Xilinx Virtex-6 ML605 Evaluation Board (with XC6VLX240T FPGA\textsuperscript{on it}) \textsuperscript{Xil12} (Figure 3.1). We also made use of Xilinx Spartan-3 FPGA\textsuperscript{Family including Spartan-3A XC3S50 FPGA\textsuperscript{Xil11a} and Xilinx Spartan-6 FPGA\textsuperscript{Family Xil11c} together with ML605 Evaluation Board for implementations purposes. All the cores were synthesized, mapped, and place & routed for the corresponding FPGA\textsuperscript{using Xilinx ISE Design Tool Xil}.  

![Figure 3.1: Xilinx Virtex-6 ML605 Evaluation Board](image-url)
3.2 Software Platforms

We also implemented our designs on embedded microcontrollers. A microcontroller is a small computer on a chip composed of a processor, programmable peripherals, memory (and in most cases program memory), and RAM as well. It is widely-used in embedded systems like remote controls, washing machines, or control systems of a car. In this section, we give brief information on the microcontrollers we have used in our research.

3.2.1 ATmega8A

The ATmega8A [Atm13] belongs to the high-performance, low-power Atmel AVR 8-bit microcontroller series. ATmega8A uses 8-bit AVR [Atm10] instruction set and has an advanced Reduced Instruction Set Computer (RISC) architecture with 130 instructions. Most of these instructions are executable in a single clock cycle. ATmega8A reaches up to 16 Million Instructions per Second (MIPS) at a frequency of 16 MHz. Moreover, it comes with three different memory segments: 8 kB of program memory, 512 bytes of EEPROM, and 1 kB internal Static Random Access Memory (SRAM). For 8-bit microcontrollers, AVR is dominating the market along with PIC [Mic] (see [Fri]), thus it was meaningful to choose an ATmega8A microcontroller for our lightweight crypto primitive implementations. In order to simulate the behavior of our implementations, we also made use of Atmel Studio Integrated Development Environment (IDE) [Atm].

3.2.2 STM32F4DISCOVERY

The STM32F4DISCOVERY [STM13], as can be seen in Figure 3.2, is based on an ARM 32-bit Cortex-M4 CPU and has a frequency up to 168 MHz. Furthermore, it is equipped with 192 kB of SRAM and 1 MB of Flash memory. The ST-LINK/V2 with SWD connector makes programming and debugging via USB possible. As ARM dominates the market for 32-bit microcontrollers and widely in use, we chose this microcontroller for the 32-bit assembly implementation of PRIDE (see Chapter 10 for details).

3.2.3 EnergyMicro EF32GG-STK3700

We synthesized PRINCE and PRIDE C implementations on EnergyMicro EF32GG-STK3700 [Sil13] Evaluation Board. It has an ARM 32-bit CORTEX-M3 microprocessor and contains sensors together with peripherals that demonstrate some of the microcontroller’s many capabilities. The board features an on-board SEGGER J-Link debugger and an Advanced Energy Monitoring system, allowing to program, debug, and perform real-time current profiling of the application without using external tools. It comes with 1 MB Flash and 128 kB RAM. As the compiler, we selected SimplicityStudio [IDE Sil14], which is based on Eclipse [Ecl] and uses GNU ARM C compiler.
Figure 3.2: STM32F4DISCOVERY
Part II

Resource-efficient Implementations of Existing Ciphers
Chapter 4

An Analysis of Recently Developed Lightweight Block Ciphers

This chapter presents an area, power, and energy analysis of some of the recently developed lightweight block ciphers and different implementations of the AES algorithm. We show that area is not always correlated to power and energy consumption, which is of importance for mobile battery-powered devices. Our work can be used as a guide while selecting algorithms based on the area, energy, and key/block length requirements of applications.

Contents of this Chapter

<table>
<thead>
<tr>
<th>4.1 Introduction</th>
<th>4.2 Background</th>
<th>4.3 Analysis Strategy</th>
<th>4.4 Results</th>
<th>4.5 Conclusions and Future Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>22</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
</tbody>
</table>

4.1 Introduction

The purpose of this investigation is to perform an area, power, and energy analysis of some of the most recently developed lightweight block ciphers along with the NIST standard AES which can be helpful for both the engineering and theoretical communities concerned with lightweight cryptography. Given that the lightweight algorithms are particularly interesting for battery-powered or passive systems such as RFID tags, a valid energy consumption prediction is very desirable. Furthermore, we want to present a comparative study on the performance of the existing lightweight primitives and AES. This is a necessary step for us in order to draw our road map for further research in lightweight cryptography area, which will be presented in the following parts of this thesis.

There have been several investigations on performances of lightweight block ciphers and AES. In addition to software performances and hardware area consumption figures of lightweight algorithms given in their corresponding publications (e.g., BKL\textsuperscript{+}07, GNL11, CDK09, HSH\textsuperscript{+}06, SMMK12), comparative studies on software CMM13, EGG\textsuperscript{+}12 and hardware EKU\textsuperscript{+}07, KSPS12, Cry14, KDH\textsuperscript{+}12 platforms have also been provided. Except the recent work of Kerkhof et al. KDH\textsuperscript{+}12, these have focused on area and power consumption figures, and they
do not discuss energy consumption. In \cite{KDH+12}, energy consumptions of six different block ciphers are evaluated as well as the area, power consumption, and throughput figures. Conclusions are drawn with respect to round unrolling, parallelism, and pipelining. Other related works in the past were focused mainly on other, more specific aspects of low-cost applications. For instance, the work of Singelée et al. \cite{SSBV11} focuses on the computation and communication energy budget of authentication protocols for active RFID tags. Accordingly, they consider other cryptographic primitives that can be used for authentication, i.e., ECC-based protocols. On the other hand, the AES algorithm as the main standard for encryption in the past decade, has been already evaluated with regard to energy consumption in several previous studies \cite{HV06, dMGSP08, TFs08}. In particular the work of Tillich et al. \cite{TFs08} examines several different AES Sbox implementations that are based on three different design strategies. The results addressed the consequences of different design strategies on critical path delay, silicon area, and power consumption.

All those works contributed to the better understanding of low-cost design principles. As mentioned above, the requirements of extreme low-cost applications of today require more lightweight solutions as advocated by the new block ciphers’ proposals. Our work extends those studies with a suite of the recent lightweight symmetric schemes. Our work covers six different lightweight block cipher architectures and five different implementations of AES block cipher. The differences between the AES architectures are based on the different implementations of Sbox.

Note that this chapter presents a collaborative work, which was later published at RFIDSec in 2013 \cite{BDE+13}; however, only the parts with author’s contribution are included in this thesis.

4.2 Background

In this section, we provide some background information on the evaluated block cipher architectures. The information is grouped according to similarities in the architectures. Note that we do not present the algorithmic details of the analyzed ciphers in this thesis, but refer to their original publications.

4.2.1 Fully-parallel Implementations

We have implemented fully-parallel (round-based) versions of AES-128, CLEFIA-128, PRESENT-80, LED-128, KLEIN-64, and mCrypton-96, where the number next to each cipher represents the chosen key length for the implementation. Among these, AES and CLEFIA are 128-bit block ciphers, whereas all others are 64-bit. For a fair comparison, we have implemented the encryption-only version of each cipher. All the implemented block ciphers share the same structure. Any of these block ciphers can be implemented as shown in Figure 4.1 where the round function is instantiated only once. In this case, the initial input (upon a start signal) of the round function is the sum of the input key and the plaintext (i.e., the initial state). It is processed by the combinational round function and the next state is generated. It is then stored in the state register, whose output becomes the input to the round function in the next cycle. The iteration count is as many rounds as the cipher is defined for. Finally, the ciphertext can be taken either from the state register output or some internal node of the round function block. The datapath width inside the round function block is equal to the cipher block size, i.e., 128
bits for AES and CLEFIA, and 64 bits for all others, while the datapath width of the key scheduling block depends on the selected key size. In the case of LED, we use a fixed key (fixed means either the original input key, or a function of it). Figure 4.2 illustrates the no key-update case for such a round-based implementation.

This is basically the design strategy we used in all our round-based implementations. In order to keep the round numbers minimum, we got the ciphertext from internal nodes inside the round function instead of the outputs of the state registers. This way, we ensured that the cipher could be run in maximal throughput, that is the distance between two consecutive starts is equal to the number of rounds.

We furthermore implemented fully-parallel AES in various flavors. Mainly, we focused on the Sbox, which is the most area consuming unit inside the AES algorithm. The first implementation (AES_lut_128) uses Look-Up Table (LUT)-based Sboxes. The second version implements the Sbox in the composite field $GF((2^4)^2)$ (AES_4_2_128), and the third version in the composite field $GF(((2^2)^2)^2)$ (AES_2_2_2_128), which are explained in [Can05]. We have also observed that
the isomorphic transform matrices used for composite implementations are very area-consuming due to the high Hamming Weight (HW), that correspond to XOR gates in hardware. In order to reduce this, it is possible to use other isomorphic matrices, that are affine equivalent to the original matrices. Of course, this requires that the corresponding affine and inverse affine transformations have to be applied to the plaintext and ciphertext, respectively. Similarly, the key scheduling also needs to be carried out in the affine equivalent “domain”. Depending on the choice of the affine transformation, it is possible to reduce the area or the power (or even both) consumption of the overall design. We chose transforms that would minimize the power consumption, and have implemented two more flavors of AES namely affine-transformed in the composite field $GF((2^4)^2)$ (AES$_{	ext{iso},4,2,128}$), and affine-transformed in the composite field $GF(((2^2)^2)^2)$ (AES$_{	ext{iso},2,2,2,128}$).

### 4.2.2 Implementation of KATAN

KATAN is a block cipher that belongs to a family of small and efficient hardware-oriented block ciphers. KATAN ciphers include KATAN32, KATAN48, and KATAN64. All three ciphers use 80-bit keys and have a different block size (KATANn has an $n$-bit block size). All three block ciphers are highly compact, with the one having the smallest block size resulting in the smallest circuit area of only 802 GEs [CDK09].

In KATAN, the plaintext is loaded in two registers. In each round, several bits are taken from the registers and entered in two non-linear Boolean functions. The output of the Boolean functions is loaded to the least significant bits of the registers (after they are shifted). This is done in an invertible manner. To ensure sufficient mixing, 254 rounds of the cipher are executed. A round counting Linear Feedback Shift Register ([LFSR]) is used instead of a counter, for counting the rounds to stop the encryption after 254 rounds, and to introduce more diffusion as well. As there are 254 rounds, an 8-bit LFSR with a sparse feedback polynomial can be used. The LFSR is initialized with some state, and the cipher has to stop running the moment the LFSR arrives at the predetermined state. The key schedule of the KATAN cipher loads the 80-bit key into another LFSR (the least significant bit of the key is loaded to position 0 of the LFSR). In each round, positions 79 and 78 of the LFSR are generated as the round’s subkey, and the LFSR is clocked twice [CDK09].

Note that the base hardware implementations of KATAN were provided by our co-authors from the Katholieke Universiteit Leuven and Radboud University Nijmegen. Then, their implementations were adjusted to be in accordance with our general synthesis structure and synthesized using our tools.

### 4.3 Analysis Strategy

This section explains the architectural decisions for the implementation of the ciphers and how the design parameters are evaluated.

#### 4.3.1 Architectural Decisions

In order to perform a fair comparison, we make the following architectural decisions:

- All inputs and outputs of the cipher are buffered by a flip-flop.
4.4. Results

We consider encryption-only architectures. This is justified by the fact that the most popular modes of operation do not need decryption in the constrained environment [Dwo01].

4.3.2 Evaluation of Design Parameters

The area reports are generated after hierarchical synthesis in Cadence Encounter RTL Compiler v10.1 using UMC 130 nm low-leakage Faraday technology library. The area numbers in the tables are given in terms of GE.

Our evaluation method consists of estimating the pre-layout power consumption and the derived energy using RTL Compiler and ModelSim simulations. Each module is first synthesized for the best power. We used Cadence Encounter RTL Compiler v10.1 in this step in order to get the power results (and the above-mentioned area figures). The generated netlists are then used to simulate the actual module with 100 random keys together with 10 random plaintexts per key to get the best statistics. From these ModelSim simulations, SAIF files are generated. In the last step, the SAIF files are sent back to the synthesis tool together with the netlist from the initial synthesis to run power analysis.

4.4 Results

In this section, we list our results for the different cipher implementations. Further, we detect anomalies based on the fact that we expect the dynamic power consumption to be larger for designs with a larger area. The anomalies represent architectures of which (some) gates contribute less to the static and/or dynamic power consumption than the gates of other architectures. This is mainly related to the number of transistors that are conducting (for the static power consumption) and the number of nodes that switch (for the dynamic power consumption). Energy per bit is calculated by dividing the total power by the clock frequency and then multiplying by the cycle count, followed by dividing by the block length.

A comparison of the AES implementations are given in Table 4.1. The parallel architectures only differ in the way the Sbox was implemented. The table shows that the architectural differences in the Sboxes cause significant differences in area, power, and energy. The reason for some architectures to have a larger dynamic power consumption compared to others while they have a smaller area is probably caused by the fact that these architectures give rise to more internal glitches. In addition, some architectures have a larger static power consumption compared to others while they have a smaller area, which is probably because of the fact that parts of the architecture are not being used all the time during the computation.

In the KATAN designs, the block size changes while the key size is the same. Therefore a slight change in area and also in static power consumption is observed in Table 4.2. But the dynamic power is quite low due to the simplistic round operations included in KATAN. When comparing LED and CLEFIA, both with block length 128, it is noticeable that the area ratio is much smaller than the power consumption ratio, in favor of LED. The reason is that LED is based on a fixed key and a simpler round computation compared to CLEFIA. The energy comparison also turns out in favor of LED. In the same way, the round function of PRESENT is much simpler in terms of logic depth compared to mCrypton, resulting in a similar trend. The energy comparison turns out in favor of mCrypton though. This is due to the fact that PRESENT needs more cycles.
Table 4.1: Performance and energy numbers of various AES realizations all using 128-bit key lengths

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Key length</th>
<th>Encryption time (# cycles)</th>
<th>Freq. (kHz)</th>
<th>Area (GE)</th>
<th>Static power (µW)</th>
<th>Dynamic power (µW)</th>
<th>Energy per bit (pJ/bit)</th>
<th>Energy (nJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES_2_2_128</td>
<td>2</td>
<td>10</td>
<td>100</td>
<td>12405</td>
<td>24.64</td>
<td>210.15</td>
<td>183.29</td>
<td>23.46</td>
</tr>
<tr>
<td>AES_4_128</td>
<td>2</td>
<td>10</td>
<td>100</td>
<td>11453</td>
<td>21.37</td>
<td>135.26</td>
<td>122.37</td>
<td>15.66</td>
</tr>
<tr>
<td>AES_iso_2_2_128</td>
<td>2</td>
<td>10</td>
<td>100</td>
<td>15442</td>
<td>30.41</td>
<td>52.85</td>
<td>65.05</td>
<td>8.33</td>
</tr>
<tr>
<td>AES_iso_4_2_128</td>
<td>2</td>
<td>10</td>
<td>100</td>
<td>13052</td>
<td>25.19</td>
<td>37.06</td>
<td>48.63</td>
<td>6.23</td>
</tr>
<tr>
<td>AES_lut_128</td>
<td>2</td>
<td>10</td>
<td>100</td>
<td>19591</td>
<td>30.81</td>
<td>96.11</td>
<td>99.16</td>
<td>12.69</td>
</tr>
</tbody>
</table>

Table 4.2: Performance and energy numbers of lightweight block ciphers implementations

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Block/Key length</th>
<th>Encryption time (# cycles)</th>
<th>Freq. (kHz)</th>
<th>Area (GE)</th>
<th>Static power (µW)</th>
<th>Dynamic power (µW)</th>
<th>Energy per bit (pJ/bit)</th>
<th>Energy (nJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLEFIA</td>
<td>128/128</td>
<td>18</td>
<td>100</td>
<td>6941</td>
<td>13.24</td>
<td>37.09</td>
<td>70.78</td>
<td>9.06</td>
</tr>
<tr>
<td>KLEIN</td>
<td>64/64</td>
<td>12</td>
<td>100</td>
<td>2760</td>
<td>4.88</td>
<td>2.18</td>
<td>13.24</td>
<td>0.85</td>
</tr>
<tr>
<td>LED</td>
<td>64/128</td>
<td>48</td>
<td>100</td>
<td>3194</td>
<td>5.62</td>
<td>2.34</td>
<td>59.70</td>
<td>3.82</td>
</tr>
<tr>
<td>mCrypton</td>
<td>64/96</td>
<td>13</td>
<td>100</td>
<td>3197</td>
<td>5.80</td>
<td>2.50</td>
<td>16.86</td>
<td>1.08</td>
</tr>
<tr>
<td>PRESENT</td>
<td>64/80</td>
<td>31</td>
<td>100</td>
<td>2195</td>
<td>3.75</td>
<td>1.14</td>
<td>23.69</td>
<td>3.82</td>
</tr>
<tr>
<td>Katan_32</td>
<td>32/80</td>
<td>254</td>
<td>100</td>
<td>801</td>
<td>1.52</td>
<td>0.43</td>
<td>154.78</td>
<td>4.94</td>
</tr>
<tr>
<td>Katan_48</td>
<td>48/80</td>
<td>254</td>
<td>100</td>
<td>925</td>
<td>1.71</td>
<td>0.49</td>
<td>116.42</td>
<td>5.60</td>
</tr>
<tr>
<td>Katan_64</td>
<td>64/80</td>
<td>254</td>
<td>100</td>
<td>1048</td>
<td>1.94</td>
<td>0.56</td>
<td>99.22</td>
<td>6.34</td>
</tr>
</tbody>
</table>

Note that all the implementations listed in Tables 4.1 and 4.2 are implemented and synthesized from scratch using our resources.

4.5 Conclusions and Future Work

In this chapter, we evaluated the area, power consumption, and energy of lightweight block cipher implementations and different AES architectures. We discussed the differences in dynamic power consumption in relation to the area. The results show that the parallel AES core with LUT-based Sboxes is the largest in terms of area, but consumes the least dynamic power of all parallel cores. For the other ciphers, the difference depends on the algorithm, mostly on the complexity of the round function.

It is evident from the results presented in this chapter, dynamic power consumption plays an important role on the energy/power consumption of cryptographic chips. Although in this work, industrial tools are used to generate dynamic power consumption results, one can also investigate the HDL implementation of a cipher and estimate the dynamic power consumption.
through measuring the toggle activity. The basic idea is to measure the total number of bit
toggles, i.e., bit flips, that happen during the encryption of a single block of a given algorithm.
This metric can be in particular helpful when considering energy. The energy consumption in
Complementary Metal Oxide Semiconductor (CMOS) circuits is dominated by dynamic power
dissipation. This is directly proportional to the number of bit toggles and hence provides a
good prediction for the energy consumption of a cipher. As future work, the relation between
the number of toggles from an HDL implementation and the power reports from the synthesis
of this implementation can be investigated.
Chapter 5

RAM-Based Ultra-Lightweight FPGA Implementation of PRESENT

This chapter proposes two different FPGA implementations of the lightweight cipher PRESENT. The main design strategy for both designs is the utilization of existing RAM blocks in FPGAs for the storage of internal states, thereby reducing the slice count. In the first design, Sboxes are realized within the slices, while in the second design they are also integrated into the same RAM block used for state storage. Both designs are well-suited for lightweight applications, which are implemented on low-cost FPGA devices. Besides low-area, a reasonable throughput is also obtained even though it is not the first concern. In addition to a single block RAM, the two designs occupy only 83 and 85 slices and produce a throughput of 6.03 and 5.13 Kbps at 100 KHz system clock on a Xilinx Spartan XC3S50 device, respectively.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>Previous Work</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>The Proposed PRESENT Implementations on FPGA</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>Results and Discussion</td>
<td>37</td>
</tr>
<tr>
<td>5.5</td>
<td>Conclusion and Future Work</td>
<td>37</td>
</tr>
</tbody>
</table>

5.1 Introduction

So far, several implementations of lightweight cryptographic algorithms have been realized on ASICs, which can also be seen in Chapter 4. While being very desirable for mass production and low-power consumption, ASICs also suffer from high costs associated with high Non-recurring Engineering (NRE) costs and long manufacturing times, which may be the killing factor for low-volume products. FPGA, present attractive alternatives for this portion of the market with their low or zero NRE costs, and short time-to-market properties.

Block RAMs on FPGA can be considered as one of their major advantages over ASICs, as RAMs require the utilization of special memory generators in ASICs that add further to manufacturing costs. However, block RAMs can also be considered as a curse for FPGA, especially in massively-computational applications (such as block ciphers implementations), where registers are dominantly-used for state data storage and RAMs are left unutilized.
Chapter 5. **RAM-Based Ultra-Lightweight FPGA Implementation of PRESENT**

In this study, we aim to demonstrate that block RAM-based implementations of block ciphers are also possible and viable on FPGAs, which result in “more slices left” for other applications. As a proof-of-concept, the lightweight block cipher PRESENT is selected as our target algorithm mainly due to its lightweight nature and wide utilization.

The PRESENT block cipher, whose suitability for compact implementations on ASICs has been demonstrated several times, can also be implemented on FPGAs efficiently. Until now, there have been register-based implementations of PRESENT on FPGAs [YK09, SSPP09]. However, none of them made use of the already existing block RAMs on FPGAs.

In this work, 128-bit key length for PRESENT is considered for comparison with previous works. We propose two different RAM-based implementations of PRESENT in the following sections.

### 5.2 Previous Work

Due to the relatively high area and power consumption of FPGAs with respect to ASICs, they have not been the target application platform for lightweight applications until recently. As a result, there have been few publications on the design and implementation of lightweight ciphers on FPGAs. Thereby, most of the existing PRESENT implementations generally target ASIC platform, as can be seen in [BKL+07, EKU+07, KSPS12, Cry14, KDH+12].

Another line of research dealt with design strategies for better FPGA implementations of ciphers [RSQL03]. This work presented new descriptions of DES and Triple-DES in order to have better results on FPGAs.

Out of the few PRESENT implementations on FPGAs (including [Pos09]), we focus on two promising works [YK09, SSPP09]. The first one is mainly a comparison of the PRESENT cipher to the HIGHT cipher on FPGAs. The second one is a detailed survey of PRESENT implementations on FPGAs using different logic optimizers. Both designs rely on registers for state storage. Therefore, they both result in slice counts above 100, while offering high data throughputs. However, in most lightweight applications, lower slice count is more important than high throughput. Hence, we set our goal to have a lower slice count.

### 5.3 The Proposed PRESENT Implementations on FPGA

As mentioned before, in order to achieve a smaller slice count in our PRESENT implementation, we opted for using one of the existing block RAMs on the FPGA for state and key storage instead of using registers. We identified two different RAM-based approaches: One is to implement the Sboxes on slices, and the second approach is to store the Sboxes in RAM as a lookup table. The second technique is expected to give a smaller slice count. The drawback of the second technique is the increased cycle count and complexity of its control block compared to the first approach.

For both approaches, a 64-bit state is represented as $4 \times 4$ state matrix with 4-bit entries. In order to store data in a $4 \times 4$ matrix, $4 \times 16$-bit entries are required for state and $8 \times 16$-bit entries are required for the key (as we implement the 128-bit key PRESENT in this work). However, this requirement is doubled by using memory in a ping-pong fashion: Rounds of PRESENT are named as even and odd rounds, and the result of an even round is written into an odd-
round address. Likewise, the result of the following odd round is written into the following even address. By doing so, existing memory space is used instead of registers. This scheme is shown in Figure 5.1.

![Diagram of PRESENT data flow scheme in the RAM-based design for a 3-round example](image)

**5.3.1 State Processing**

While Sboxes can be applied to each 16-bit entry in parallel, due to the nature of permutation layer, which is shown in Figure 5.2, it is not possible to do the same thing for permutation. Therefore, a different approach is applied. This approach can be best explained by a toy version of PRESENT with only 3 rounds as shown in Figure 5.1.

In the first round (Round-1), the first 16-bit of input data and key are read from address $S_0$ and $K_0$, then XORed and Sboxed in the following cycle. The result is sent to a temporary register to appear in the next cycle. Also, the first target address $S_4$ is read in the same cycle. In the next cycle, permutation is performed by using the first bit of every nibble in temporary register (which is the XORed and Sboxed value) and stored to $S_4$ via shifting it by 4 bits to left. At the same time, temporary register is shifted left by 1-bit, in order to get the correct permutation values for the next cycle. The same operation is performed for addresses $S_5$, $S_6$, $S_7$, and $K_0$.
and $S_7$. At the end of 6 cycles, first nibbles of $S_4$, $S_5$, $S_6$, and $S_7$ are ready. The procedure is repeated for all other input addresses $S_1$, $S_2$, and $S_3$. The cycle count for each of these is also 6 cycles. In total, this processing of state takes $4 \times 6 = 24$ cycles for each round. This scheme can be seen in Figure 5.3.

For the on-RAM Sbox version of the implementation, the only difference is the additional cycle for reading Sbox values from memory, which can be seen in Figure 5.4. Therefore, the total cycle count of state processing in each round is 28 cycles.

### 5.3.2 Key Schedule

At the end of every round, key scheduling of PRESENT is performed. Again, the first address (for first round, $K_0$) is read to appear in the following cycle. Then, it is shifted by 3 bits and stored in the corresponding address in order to perform key scheduling. At the same time, least significant 3 bits of the read key data is stored in a 3-bit key temporary register for the shifting part of scheduling. The value kept in key temporary register is also written into the
5.3. The Proposed PRESENT Implementations on FPGA

<table>
<thead>
<tr>
<th>PH</th>
<th>CYC</th>
<th>PORT-A OPERATION</th>
<th>PORT-A OUTPUT</th>
<th>TMP REGISTER</th>
<th>PORT-B OPERATION</th>
<th>PORT-B OUTPUT</th>
<th>PORT-B INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>READ S0</td>
<td></td>
<td></td>
<td>READ K0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>READ S4</td>
<td>S0,0 S0,1 S0,2 S0,3</td>
<td>k0,0 k0,1 k0,2 k0,3</td>
<td>WRITE S4</td>
<td>S1,0 S1,1 S1,2 S1,3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>READ S5</td>
<td>b1,0 b1,1 b1,2 b1,3</td>
<td>WRITE S4</td>
<td>S1,0 S1,1 S1,2 S1,3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>READ S6</td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S5</td>
<td>S1,4 S1,5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>READ S7</td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S6</td>
<td>S1,8 S1,9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S7</td>
<td>S1,12 S1,13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>READ S1</td>
<td></td>
<td></td>
<td>READ K1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>READ S4</td>
<td>b0,0 b0,1 b0,2 b0,3</td>
<td>k0,0 k0,1 k0,2 k0,3</td>
<td>WRITE S4</td>
<td>S1,0 S1,1 S1,2 S1,3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>READ S5</td>
<td>b1,0 b1,1 b1,2 b1,3</td>
<td>WRITE S4</td>
<td>S1,0 S1,1 S1,2 S1,3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>READ S6</td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S5</td>
<td>S1,4 S1,5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>READ S7</td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S6</td>
<td>S1,8 S1,9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S7</td>
<td>S1,12 S1,13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>READ S2</td>
<td></td>
<td></td>
<td>READ K2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>READ S4</td>
<td>b0,0 b0,1 b0,2 b0,3</td>
<td>k0,0 k0,1 k0,2 k0,3</td>
<td>WRITE S4</td>
<td>S1,0 S1,1 S1,2 S1,3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>READ S5</td>
<td>b1,0 b1,1 b1,2 b1,3</td>
<td>WRITE S4</td>
<td>S1,0 S1,1 S1,2 S1,3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>READ S6</td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S5</td>
<td>S1,4 S1,5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>READ S7</td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S6</td>
<td>S1,8 S1,9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S7</td>
<td>S1,12 S1,13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>READ S3</td>
<td></td>
<td></td>
<td>READ K3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>READ S4</td>
<td>b0,0 b0,1 b0,2 b0,3</td>
<td>k0,0 k0,1 k0,2 k0,3</td>
<td>WRITE S4</td>
<td>S1,0 S1,1 S1,2 S1,3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>READ S5</td>
<td>b1,0 b1,1 b1,2 b1,3</td>
<td>WRITE S4</td>
<td>S1,0 S1,1 S1,2 S1,3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>READ S6</td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S5</td>
<td>S1,4 S1,5 S1,6 S1,7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>READ S7</td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S6</td>
<td>S1,8 S1,9 S1,10 S1,11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>tmp &lt;&lt; 1</td>
<td>WRITE S7</td>
<td>S1,12 S1,13 S1,14 S1,15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: PRESENT cycle count for the state processing phases of each round (on-slice Sbox version)

The corresponding address with the read and 3-bit shifted key data at the same cycle. This process is performed for all 8 addresses, which results in 10 cycles as seen in Figure 5.5.

For the **on FPGA** Sbox version, again the Sbox value read part is added to this flow, which results in an increase of 2 cycles as shown in Figure 5.6

### 5.3.3 Final Round

At the last round, Round-32, only XOR operation is performed and the result is written-back to the original input addresses. To perform this, an XOR and write cycle comes after every read data and key read. This results in 8 cycles for both approaches, as there is no Sbox operation in this phase. This scheme is shown in Figure 5.7.
5.3.4 Overall Design

According to the data flow explained above, the hardware circuit of the design is defined. As can be seen from Figure 5.8, the regular on-slice Sbox implementation has rather simple logic; however, it uses 4 Sboxes that increase the slice count. In Figure 5.9, the block diagram of on-RAM Sbox design is shown. It is easy to see that the control logic of this implementation is more complex, compared to the first approach. Therefore, even though it has no on-slice Sboxes,
5.3. The Proposed PRESENT Implementations on FPGA

<table>
<thead>
<tr>
<th>PH</th>
<th>CYC</th>
<th>PORT-A OPERATION</th>
<th>PORT-A OUTPUT</th>
<th>KEY TMP REGISTER</th>
<th>PORT-B OPERATION</th>
<th>PORT-B INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>READ K0</td>
<td>k0,0-3</td>
<td>WRITE K12</td>
<td>k0,0-3 &gt;&gt; 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ K1</td>
<td>k0,4-7</td>
<td>WRITE K13</td>
<td>k0,0-3 &lt;&lt; 13</td>
<td>k0,4-7 &gt;&gt; 3 = k1,20-23</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ K2</td>
<td>k0,8-11</td>
<td>WRITE K14</td>
<td>k0,4-7 &lt; 13</td>
<td>k0,8-11 &gt;&gt; 3 = k1,24-27</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ K3</td>
<td>k0,12-15</td>
<td>WRITE K15</td>
<td>k0,8-11 &lt; 13</td>
<td>k0,12-15 &gt;&gt; 3 = k1,28-31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ K4</td>
<td>k0,16-19</td>
<td>WRITE K8</td>
<td>k0,12-15 &lt; 13</td>
<td>k0,16-19 &gt;&gt; 3 = k1,6-3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ K5</td>
<td>k0,20-23</td>
<td>WRITE K9</td>
<td>k0,16-19 &lt; 13</td>
<td>k0,20-23 &gt;&gt; 3 = k1,4-7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ K6</td>
<td>k0,24-27</td>
<td>WRITE K10</td>
<td>k0,20-23 &lt; 13</td>
<td>k0,24-27 &gt;&gt; 3 = k1,8-11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ K7</td>
<td>k0,28-31</td>
<td>WRITE K11</td>
<td>k0,24-27 &lt; 13</td>
<td>k0,28-31 &gt;&gt; 3 = k1,12-15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ K8</td>
<td>k0,0-3</td>
<td>WRITE K12</td>
<td>k0,28-31 &lt; 13</td>
<td>k0,0-3 &gt;&gt; 3 = k1,16-19</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>READ k0,0-1</td>
<td>k0,0-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>WRITE K8</td>
<td>k1,0-1</td>
<td>k1,0-3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.6: PRESENT cycle count for the key expansion phase of each round (on-RAM Sbox version)

<table>
<thead>
<tr>
<th>PH</th>
<th>CYC</th>
<th>PORT-A OPERATION</th>
<th>PORT-A OUTPUT</th>
<th>PORT-A INPUT</th>
<th>PORT-B OPERATION</th>
<th>PORT-B OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>READ S4</td>
<td>S3,0 S3,1 S3,2 S3,3</td>
<td>S4,0 S4,1 S4,2 S4,3</td>
<td>k3,0 k3,1 k3,2 k3,3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>WRITE S0</td>
<td>S3,0 S3,1 S3,2 S3,3</td>
<td>k3,0 k3,1 k3,2 k3,3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>READ S5</td>
<td>S3,4 S3,5 S3,6 S3,7</td>
<td>k3,4 k3,5 k3,6 k3,7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>WRITE S1</td>
<td>S3,4 S3,5 S3,6 S3,7</td>
<td>k3,4 k3,5 k3,6 k3,7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>READ S6</td>
<td>S3,8 S3,9 S3,10 S3,11</td>
<td>k3,8 k3,9 k3,10 k3,11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>WRITE S2</td>
<td>S3,8 S3,9 S3,10 S3,11</td>
<td>k3,8 k3,9 k3,10 k3,11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>READ S7</td>
<td>S3,12 S3,13 S3,14 S3,15</td>
<td>k3,12 k3,13 k3,14 k3,15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>WRITE S3</td>
<td>S3,12 S3,13 S3,14 S3,15</td>
<td>k3,12 k3,13 k3,14 k3,15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.7: PRESENT cycle count for the final round (same in both on-slice and on-RAM Sbox versions)

it ended up with a higher slice count than the first approach, which is not in accordance with our initial guess.
Chapter 5. **RAM-Based Ultra-Lightweight FPGA Implementation of PRESENT**

**Figure 5.8:** PRESENT block diagram (key expansion module shown on the right) for on-slice Sbox version

**Figure 5.9:** PRESENT block diagram (key expansion module shown on the right) for on-RAM Sbox version
5.4 Results and Discussion

Table 5.1 shows the performance comparison of both designs to prior work. As shown in the table, our designs achieve the lowest slice counts with the addition of a block RAM. On the cycle count, our designs are slower, but still offer respectable throughput for lightweight applications. Our figure of merit (throughput per slice) can reach one third of the previous most compact work.

Table 5.1: Performance comparison

<table>
<thead>
<tr>
<th>Design</th>
<th>Key size</th>
<th># of slices &amp; RAMs</th>
<th>Cycle count</th>
<th>Max freq. @100KHz (MHz)</th>
<th>Tput @max.freq. (kbps)</th>
<th>Tput/slice @max.freq. (Mbps/slice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our work (on-slice Sbox)</td>
<td>128</td>
<td>83 / 1</td>
<td>1062</td>
<td>129.77</td>
<td>6.03</td>
<td>0.094</td>
</tr>
<tr>
<td>Our work (on-RAM Sbox)</td>
<td>128</td>
<td>85 / 1</td>
<td>1248</td>
<td>135.15</td>
<td>5.13</td>
<td>0.082</td>
</tr>
<tr>
<td>[YK09]</td>
<td>128</td>
<td>117 / 0</td>
<td>256</td>
<td>114</td>
<td>24.96</td>
<td>0.24</td>
</tr>
<tr>
<td>[SSPP09]</td>
<td>128</td>
<td>202 / 0</td>
<td>32</td>
<td>254</td>
<td>200</td>
<td>2.51</td>
</tr>
</tbody>
</table>

5.5 Conclusion and Future Work

As the cost of FPGAs get lower together with their power consumption, they become more and more attractive alternatives for lightweight security applications that are realized on sensor nodes, RFIDs, etc. It is crucial to implement the lightweight cryptography on such devices in a way to ensure the maximum possible amount of resources left for other circuitry that are sharing the device. Utilization of the RAMs on FPGAs allows this at least to a certain limit.

In this work, we presented two different RAM-based implementations of the lightweight cipher algorithm PRESENT on FPGAs. With their lowest reported slice counts, both implementations are suitable for lightweight applications. While the first implementation is fully-implemented on slices, the second one integrates the Sboxes into the block RAM as well.

Our on-slice Sbox and on-RAM Sbox designs are good candidates for the application of the shared Sbox technique in [PMK11] and the block memory content scrambling technique given in [GM11] for SCA-resistance. As the next step of this study, these SCA countermeasures can be applied to our designs and their effects on the area as well as their resistance to SCA attacks can be observed.
Chapter 6

On the Suitability of SHA-3 Finalists for Lightweight Applications

In this chapter, we investigate the suitability of SHA-3 finalists for lightweight applications. For each finalist, we try to achieve the lowest reported gate count while maintaining a respectable throughput. We mainly favor a word-serial approach in our designs to achieve low gate count, where the word size varies from 32 to 64 bits depending on the structure of the hash function and the trade-off between the throughput and area. All hash function cores are realized in Verilog HDL, synthesized using 90 nm UMC CMOS standard-cell library and optimized for area for prototyping. A generic First In First Out (FIFO)-based Input/Output (IO) interface is also built in order to establish data transfer between an external controller and the active hash function core.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>39</td>
</tr>
<tr>
<td>6.2</td>
<td>Implementations of the Finalist Hash Functions</td>
<td>40</td>
</tr>
<tr>
<td>6.3</td>
<td>Interface</td>
<td>59</td>
</tr>
<tr>
<td>6.4</td>
<td>Results and Discussion</td>
<td>60</td>
</tr>
<tr>
<td>6.5</td>
<td>Conclusion</td>
<td>61</td>
</tr>
</tbody>
</table>

6.1 Introduction

[NIST] announced a public competition in November 2007 in order to develop a new cryptographic hash algorithm. The winning algorithm was planned to be named as "SHA-3" and the hash algorithms specified in Federal Information Processing Standard (FIPS) 180-3, Secure Hash Standards [NIS08], were planned to be augmented. There were numerous submissions to the SHA-3 competition. It proceeded round by round, and in the last round there were only 5 finalists left: BLAKE [AHMP10], Grøstl, JH [Wu11], Keccak [BDPA11] and Skein [FLS+10]. In October 2012, during the course of this thesis work, the winner of the competition was announced as Keccak.

The SHA-3 competition had been a very attractive process, there were many studies and discussions on the submitted algorithms. Implementation of the algorithms was an important part of these investigations. Several software and hardware implementations dealt with effective and high performance realization of the candidates on a wide range of platforms from...
embedded processors to custom ASICs. Out of these, KKI+11, TFI+09, GSH+12 compare (sometimes compact) hardware implementations of different SHA-3 candidates, including the finalists, on ASIC. Other works provided comparative analyses of “compact and lightweight” implementations of the finalists on FPGA platform KDVC+11, KYS+11, JA11. However, the suitability of the SHA-3 candidates for lightweight applications was not comprehensively studied. In this work, we investigated the suitability of the SHA-3 finalists for lightweight applications and tried to come up with suggestions for possible lightweight extensions.

The main target of this study is to present efficient compact implementations of SHA-3 finalists offering the lowest possible gate count (and therefore the lowest power consumption), whereas the resultant throughput is still within the limits desirable for lightweight applications. One approach to achieve this target is to replace registers by RAM(s) and implement minimal combinational circuitry necessary for the realization of computational operations. Another approach is to keep the registers, but perform computational operations serially, thereby saving logic and interconnection area. We opted for the latter option, mainly because of the non-standard block memory interfaces and performances offered by different process technologies. We also believe that the structures we propose for each hash function can be easily modified and used within a hybrid approach.

In our study, we chose the 256-bit message digest option for all finalists. Our designs are both suitable for ASIC and FPGA platforms. However, we have synthesized our implementations in 90 nm UMC CMOS technology. Area-optimized synthesis results show that Grøstl offers the lowest gate count, while BLAKE offers the best throughput and throughput/area numbers. We have also compared the finalists with each other in order to observe the overall performance.

6.2 Implementations of the Finalist Hash Functions

Before giving the details of each implementation, we describe each hash function briefly in order to clarify our architectures.

6.2.1 BLAKE

Algorithm

BLAKE is a family of four hash functions: BLAKE-224, BLAKE-256, BLAKE-384, and BLAKE-512, which follows the HAIFA iteration mode BD07. The compression function depends on a salt and the number of bits hashed so far (as counter): A large inner state is initialized from the initial value, the salt and the counter; and it is injectively updated by message-dependent rounds until it is finally compressed to return the next chain value, as shown in Figure 6.1.

The inner state of the compression function is represented as a $4 \times 4$ matrix of words. In one round of BLAKE-256, all four columns and then all four disjoint diagonals are updated independently. In the update of each column or diagonal, two message words are input according to a round-dependent permutation as shown in Figure 6.2.

Table 6.1 shows the specification of BLAKE for 256-bit message digest.
6.2. Implementations of the Finalist Hash Functions

The serialized architecture for BLAKE is given in Figure 6.3. The first operation is the initialization, where the data is written into the state registers as 32-bit words in 16 clock cycles. The salt, hash and message registers, which are also shown in Figure 6.3, store the salt, the hash and the message, respectively. The state words are then processed by the half \( G_i \) function block shown in Figure 6.4, together with the corresponding values from the other registers, and written back onto the state register. The \( G_i \) function module operates on each column for \( G_0 \)–\( 3 \), and then four disjoint diagonals for \( G_4 \)–\( 7 \) twice because of its “half” structure. This structure, while reducing the area, doubles the cycle count.

As shown in Figure 6.5, \( G_0 \)–\( 3 \) is processed at first, in halves (namely \( H_1 \) and \( H_2 \)); followed by the processing of \( G_4 \)–\( 7 \), again in halves. The multiplexers are switched in order to make sure that the sequence of the serially-processed words gives the same result as a parallel implementation. This process is repeated for 14 rounds, and a new message block is injected after the 14th round.
Figure 6.3: BLAKE serial architecture

Figure 6.4: $G_i$ half function
6.2. Implementations of the Finalist Hash Functions

Figure 6.5: BLAKE serial data flow
Chapter 6. On the Suitability of SHA-3 Finalists for Lightweight Applications

Figure 6.6: BLAKE timing diagram

(if it exists). Injection of the message blocks continues until the last block. The finalization process returns the next chain value (or message digest, if it is the last message block).

The whole process is explained in phase-round-cycle concept in Figure 6.6. In phase-0, the salt is read in 8 cycles. In the following 4 cycles, the length of the message block is read, which is phase-1. Following the length, the first message block is read in phase-2 in 16 cycles. In phase-3, the data processing is performed for 14 rounds (each round in 16 cycles). The next message block is read in phase-4. However, after the last message block, the message digest is written back in the first 8 cycles of phase-4.

6.2.2 Grøstl

Algorithm

Grøstl is a collection of hash functions, which can return message digests from 8 to 512 bits in 8-bit steps. The variant returning \( n \) bits is called Grøstl-\( n \). Hashing starts by padding the input message \( M \) and splitting it into \( l \)-bit messages \( m_1, \ldots, m_t \). Each message block is then processed sequentially by the iterative compression function \( f \), whose other input is the \( l \)-bit chaining input with an initial value of \( h_0 = iv \), as shown in Figure 6.7. For Grøstl variants with \( n \) up to 256 (which covers our case), \( l \) is defined to be 512. After the processing of the last message block, the output \( H(M) \) of the hash function is computed as \( H(M) = \Omega(h_t) \). Here \( \Omega \) is the output transformation, whose output size is \( n \) bits, where \( n \leq 2l \).

Figure 6.7: Grøstl compression function

The compression function \( f \) is based on two \( l \)-bit permutations \( P \) and \( Q \), which is defined as \( f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h \); and the output function is defined by \( \Omega(x) = \text{trunc}_n(P(x) \oplus \text{trunc}_n(Q(x))) \).
Implementations of the Finalist Hash Functions

$x$, where $\text{trunc}_n(x)$ discards all but the trailing $n$ bits of $x$. Both functions are illustrated in Figure 6.8. Figure 6.9 shows details of $P$ and $Q$ permutations.

Figure 6.8: Grøstl construction function $f$ (left) and output function $\Omega$ (right)

Figure 6.9: $P$ and $Q$ permutations

Table 6.2 shows the specification of Grøstl for 256-bit message digest.

Table 6.2: Grøstl specifications

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Word</th>
<th>Message</th>
<th>Block</th>
<th>Rounds</th>
<th>Digest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grøstl-256</td>
<td>32-bit</td>
<td>$(2^{573} - 577)$-bit</td>
<td>512-bit</td>
<td>10</td>
<td>256-bit</td>
</tr>
</tbody>
</table>
Chapter 6. On the Suitability of SHA-3 Finalists for Lightweight Applications

Implementation

The serialized architecture for Grøstl is shown in Figure 6.10. There exists only a single block for both P and Q operations in order to save area, which also allows us to use the same block for both f and Ω functions. For the f function, message and previous hash result (which is iv at the first round) are selected as input. For the output function Ω, the only input comes from the hash register and zero is selected instead of the message.

While the message is processed inside the P/Q module in P mode, it is also stored inside the temp register. In the Q mode, the result of P is stored inside the temp register while the message is restored. It is then processed in Q mode, and its result is combined with the P result (restored from the temp register) and the previous hash value. The detailed block diagram of P/Q module is shown in Figure 6.11. It basically implements a modified version of the serial AES-like data flow in [HAHH06] via SubBytes, ShiftBytes and MixBytes functions. The data flow for a 4 × 4 toy version of ShiftBytes and MixBytes are given in Figure 6.12, note that ShiftBytes operation is different for P and Q.

![Figure 6.10: Grøstl serial architecture](image)

The whole process is explained in phase-half round-round-cycle concept in Figure 6.13. In phase-0, the length is read in 10 cycles. Phase-1 is for reading the initialization vector iv. Following this, the message blocks are read and processed. Finally, in phase-3, the message digest is written back.

6.2.3 JH

Algorithm

JH is a family of four hash algorithms — JH-224, JH-256, JH-384, and JH-512. In the design of JH, a compression function is constructed from a large block cipher with constant key. The generalized d-dimensional AES design methodology is applied in the design of the large cipher. In our case of 256-bit digest, d is set to 8, hence the compression function is named as F_8. It sequentially processes the padded and split message blocks m_1, ..., m_t, starting with an initial vector (iv), as shown in Figure 6.14.

F_8 is bijective due to the block cipher, whose block size is 2m bits. Its structure is shown in Figure 6.15 together with the internal function E_8. The 2m-bit hash value H_{i-1} and the m-bit
message block $M_{(i)}$ are compressed into the $2m$-bit $H_{(i)}$. $E_8$ is also bijective and applies SPN and Maximum Distance Separable (MDS) constructions to the bit array. MDS is applied before
Figure 6.13: Grøstl timing diagram
6.2. Implementations of the Finalist Hash Functions

The first and after the last rounds. The round function \( R_8 \) consists of an Sbox layer (selected via round constants), a linear transformation layer (applied on bytes), and a permutation layer \( P_8 \) (composed of three permutations), whose details can be seen in Figure 6.16. \( R_8 \) is repeated 42 times.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Word</th>
<th>Message</th>
<th>Block</th>
<th>Rounds</th>
<th>Digest</th>
</tr>
</thead>
<tbody>
<tr>
<td>JH-256</td>
<td>32-bit</td>
<td>(&lt; 2^{64})-bit</td>
<td>512-bit</td>
<td>42</td>
<td>256-bit</td>
</tr>
</tbody>
</table>

**Implementation**

The serialized architecture for JH is given in Figure 6.17. A 32-bit datapath is used in the serialized implementation of JH. The state register is filled with the sum (XOR) of the initialization vector and the message block at the beginning of the process, while the message is also backed up in the message register for post-processing. Upon completion of the rounds, the output of...
Chapter 6. On the Suitability of SHA-3 Finalists for Lightweight Applications

Figure 6.16: Three layers of round function

The $E_8$ block is combined with the backed up message to form the next value of the state register (hash), which in turn is summed with the next message block. This process continues until all the message blocks are processed.

Figure 6.17: JH serial architecture

The group/de-group block realizes the grouping and de-grouping steps of $E_8$ function. It only performs grouping/de-grouping at word-level. Instead of implementing bit-level grouping/de-grouping, $E_8$ round function is modified in order to support operation on the word-level-grouped input and produce output compatible with word-level de-grouping. Serialized $E_8$ round function consists of an Sbox, the linear transformation block, and the $\pi_d$, $P'_d$, and $\phi_d$ partial permutation blocks. All, except the $P'_d$-module, operate on 32 bits.

The serial data flow of JH is shown in Figure 6.18. It starts with the grouping round, which lasts for 32 cycles. This round is followed by $R_8$ round function for 42 rounds (each of them is again 32 cycles). After $R_8$ process, de-grouping round is performed. These grouping and de-grouping operations result in two additional rounds, which make 44 rounds in total. For the last message block, one extra quarter round is required for squeezing the output.
Figure 6.18: JH serial flow
Chapter 6. On the Suitability of SHA-3 Finalists for Lightweight Applications

The whole process is explained in phase-half round-round-cycle concept. In phase-0, the length of the message block is read. Then, in phase-1, initialization vector is read and stored in the state register. In phase-2, the message blocks are read in every round-0 and these message blocks are processed from round-1 to round-44. Also, the message digest is written back in round-44 of the last message block, again in phase-2. This scheme can be seen in Figure 6.19.

![Figure 6.19: JH timing diagram](image)

### 6.2.4 Keccak

**Algorithm**

Keccak, the winning hash function, is a family of hash functions based on the sponge construction [BDPA07]. The fundamental function is the Keccak-$f[b]$ permutation, which consists of a number of simple rounds with logical operations and bit permutation. $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ is the width of both the permutation and the state in the sponge construction. In our work, we concentrate on Keccak-$f[1600]$ with 256-bit message digest.

The state of Keccak is organized in $5 \times 5$ lanes, each with $w$-bits, where $w \in \{1, 2, 4, 8, 16, 32, 64\}$, and $b = 25w$. The Keccak[$r, c, d$] sponge function (Figure 6.20) is obtained by applying the sponge construction to Keccak-$f[r+c]$ with the parameters capacity $c$, bit rate $r$ (which are 512 and 1088, respectively, for Keccak-$f[1600]$). The flow of Keccak-$f$ and the details of the steps are given in Figure 6.21. The number of rounds $n_r$ depends on the permutation width, which is calculated by $n_r = 12 + 2 \times l$, where $2^l = w$. This yields 24 rounds for Keccak-$f[1600]$.

Table 6.4 shows the specification of Keccak-$f[1600]$ for 256-bit message digest.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Word</th>
<th>Message Block</th>
<th>Rounds</th>
<th>Digest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keccak-256</td>
<td>64-bit</td>
<td>$&lt; 2^{128}$-bit</td>
<td>1088-bit</td>
<td>24</td>
</tr>
</tbody>
</table>

**Implementation**

The serialized architecture for Keccak is given in Figure 6.22. In the serial design, the data is processed in lanes, which is 1/25 of the whole state. The state registers, numbered 24 – 0,
6.2. Implementations of the Finalist Hash Functions

Figure 6.20: Sponge construction of Keccak

Figure 6.21: Keccak-$f$ function and steps of the function
are used to store the internal state, and the four summation registers (rightmost registers numbered 4 – 0) store the row sums. The operational blocks that implement a Keccak round are the \( \theta, \rho, \pi, \chi, \iota \)-modules. All, but \( \pi \)-module, operate on a single lane. \( \pi \)-step is executed in parallel on all 25 lanes. It is a fixed permutation operation, and the only area cost comes from the additional multiplexers and routing. There is additional area cost caused by sum registers (required for \( \theta \)-step) and two temporary registers (required for \( \chi \)-step). However, this additional area is compensated by the huge area saving of the serialized processing and the resulting single lane combinational blocks.

![Keccak serial architecture](image)

Figure 6.22: Keccak serial architecture

The processing starts with round 31, where the length of the message block is read. Then round 0 comes, where the data is written in lanes into the state registers and each row sum is accumulated inside the sum registers. The first incoming data is lane(0, 0) and it is shifted into the state register 24 while sum register 4 is filled with the same value. In the next cycle, state register 24 is shifted into state register 23 and filled with the incoming lane(1, 0). In parallel, sum register 4 is shifted into the sum register 3, and re-initialized with lane(1, 0). At the end of the first 5 cycles, the first 5 lanes of data are in state registers 24 to 20, while sum registers 4 to 0 have the first lanes of each column. In the following cycles, incoming data are added onto the sum registers and shifted into the state registers. At the end of the first 25 cycles, state registers contain the full state and sum registers contain the row sums.

Starting with the next cycle, \( \theta \) and \( \rho \) operations are run in parallel from lane(0, 0) until lane(4, 4), covering the whole state. These operations are completed in 25 cycles. It is followed by another 25 cycles, where \( \pi , \chi \), and \( \iota \) operations are performed. Since \( \pi \) can only be executed on the whole state, it is done in parallel with the first lane of \( \chi \). The \( \iota \) operation (round constant addition) is also done in the same cycle. In the following 24 cycles, the \( \chi \) operation is performed on the remaining lanes, completing the first round. Each of these 25 cycles are named as half rounds. The row summations for the following round are also performed in parallel with \( \pi \), \( \chi \), and \( \iota \) operations of the current round, as an additional optimization. A full round takes 50 cycles to complete.

At the end of the 24 rounds, the second half round of the “last” round is used for “squeezing” the message digest. The timing diagram in Figure 6.23 shows the round, half round, and cycles for processing of two message blocks.
The whole data processing in each half round is explained by a $3 \times 3$ lanes toy-version of Keccak in Figure 6.24 instead of the actual $5 \times 5$ lanes configuration.

Figure 6.24: Keccak data flow

6.2.5 Skein

Algorithm

Skein is a family of hash functions with three different internal state sizes: 256, 512 and 1024 bits, where Skein-512 is the primary hash function and can be used for all current hashing applications. Skein hash function is built out of a tweakable block cipher (ThreeFish), which
allows hashing configuration data along with the input text in every block, making every instance of the compression function unique. In addition to the ThreeFish tweakable block cipher (256, 512 and 1024-bit block sizes) at the core, Skein is built up of a Unique Block Iteration (UBI), which maps an arbitrary input size to a fixed output size, and an optional argument system that allows supporting different optional features. The normal (straightforward) hashing option we use can be seen in Figure 6.25. The first block is for configuration, the following instances are for message processing, and the last block is for output processing.

![Figure 6.25: Skein normal hashing scheme](image)

ThreeFish tweakable block cipher is defined for 256, 512 and 1024-bit block sizes. The key is the same size as the block, and the tweak value is 128 bits for all block sizes. Each one of Skein-512’s 72 rounds consists of four MIX functions followed by a permutation of the eight 64-bit words. A subkey is added every four rounds. The word permutation is the same for every round, and the rotation constants repeat themselves every eight rounds. A key schedule is also performed for generating subkeys from the original key and the tweak. Figure 6.26 shows ThreeFish-512 construction for four rounds together with the internal details of the MIX function, which is an ARX construction.

Table 6.5 shows the specification of Skein for 256-bit message digest.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Word</th>
<th>Message</th>
<th>Block</th>
<th>Rounds</th>
<th>Digest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skein-256</td>
<td>64-bit</td>
<td>&lt; $2^{64}$-bit</td>
<td>512-bit</td>
<td>72</td>
<td>256-bit</td>
</tr>
</tbody>
</table>

Table 6.5: Skein specifications

Implementation

The serialized architecture for Skein is given in Figure 6.27. In round-0, the rightmost eight key expansion registers are filled with input key in 8 cycles, while all input key words are accumulated in the leftmost key register. This practically implements the key expansion process defined for ThreeFish. Following this round, state register is filled the sum of the input message block and the subkey generated in the previous round. In parallel, key expansion process continues within the key registers. At the same time, the message block is backed up inside the message register for post-processing following the completion of all ThreeFish rounds.
6.2. Implementations of the Finalist Hash Functions

![Four rounds of ThreeFish-512](image1)

Figure 6.26: Four rounds of ThreeFish-512

![Skein serial architecture](image2)

Figure 6.27: Skein serial architecture
ThreeFish processing inside the state register is done via a 128-bit MIX block and a fully parallel 512-bit permutation block, which is a fixed 64-bit word based permutation. Its only cost is multiplexers. The 128-bit MIX block requires an additional 64-bit temporary register in order to collect 128-bits of data. At the end of round-42, ThreeFish operation is completed, and round-43 is used to add the stored messages onto the ThreeFish result (UBI operation) in order to obtain the next state of the hash. The operation is repeated until all message blocks are processed. The serial data flow of Skein is shown in Figure 6.28.

The whole process is explained in phase-round-cycle concept. In phase-0, the length of the message block is read. Then, in phase-1, 512-bit initialization vector is directly read from RAM which makes additional ThreeFish run not necessary. In phase-2, the message blocks are read and processed. Following this, the hash value is updated in phase-3. Phase-2 and phase-3 are repeated in series, until all message blocks are processed. After the processing of the last message block, the message digest is written back in that block’s phase-3. This scheme can be seen in Figure 6.29.
6.3 Interface

All five hash modules are connected to a 32-bit [FIFO] based [IC] interface module in order to connect to the external world in an Integrated Circuit (IC). Internal interface with modules is 64-bit for Keccak and 32-bit for all other blocks. The [FIFO] is organized as an even/odd couple in order to provide 64 bits necessary for the Keccak block (both [FIFO] active) and 32 bits for the others (odd [FIFO] active in odd cycles, even [FIFO] active in even cycles). A simple [REQ]uest/[ACK]nowledge signalling scheme is implemented, where [REQ] signal is set when [FIFO] is almost empty and [ACK] is set when the result is ready. 2016 bytes of memory exist for [MESSAGE]/[DATA] and 32 bytes are present for [HASH] result (message digest). The architecture enables only the selected module, and disables the others via clock gating. This interface architecture is shown in Figure 6.30.

Figure 6.30: Interface model
Chapter 6. On the Suitability of SHA-3 Finalists for Lightweight Applications

Table 6.6: Comparison of our work with previous works

<table>
<thead>
<tr>
<th>Finalist</th>
<th>Techn. (nm)</th>
<th>Area (kGE)</th>
<th>Message block size (bits)</th>
<th>Cycles per block (MHz)</th>
<th>Tput (kbps @100kHz)</th>
<th>Tput/area (bps per GE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAKE</td>
<td>180</td>
<td>13.58</td>
<td>512</td>
<td>215</td>
<td>816</td>
<td>63</td>
</tr>
<tr>
<td>BLAKE</td>
<td>180</td>
<td>8.64</td>
<td>512</td>
<td>100</td>
<td>N.A.</td>
<td>63</td>
</tr>
<tr>
<td>Our BLAKE</td>
<td>90</td>
<td>11.3</td>
<td>N.A.</td>
<td>240</td>
<td>213</td>
<td>18.88</td>
</tr>
<tr>
<td>Grostl</td>
<td>350</td>
<td>14.622</td>
<td>512</td>
<td>56</td>
<td>N.A.</td>
<td>261</td>
</tr>
<tr>
<td>Our Grostl</td>
<td>90</td>
<td>9.2</td>
<td>512</td>
<td>N.A.</td>
<td>1280</td>
<td>40</td>
</tr>
<tr>
<td>JH</td>
<td>180</td>
<td>58.832</td>
<td>512</td>
<td>380.22</td>
<td>39</td>
<td>1313</td>
</tr>
<tr>
<td>JH</td>
<td>90</td>
<td>31.864</td>
<td>512</td>
<td>353</td>
<td>N.A.</td>
<td>1314</td>
</tr>
<tr>
<td>Our JH</td>
<td>90</td>
<td>13.6</td>
<td>512</td>
<td>N.A.</td>
<td>1440</td>
<td>36</td>
</tr>
<tr>
<td>Keccak</td>
<td>130</td>
<td>9.36</td>
<td>1088</td>
<td>200</td>
<td>5160</td>
<td>20</td>
</tr>
<tr>
<td>Our Keccak</td>
<td>90</td>
<td>15.2</td>
<td>1088</td>
<td>N.A.</td>
<td>1200</td>
<td>91</td>
</tr>
<tr>
<td>Skein</td>
<td>350</td>
<td>12.890</td>
<td>512</td>
<td>80</td>
<td>N.A.</td>
<td>25</td>
</tr>
<tr>
<td>Skein</td>
<td>90</td>
<td>22.562d</td>
<td>512</td>
<td>50</td>
<td>10</td>
<td>2694</td>
</tr>
<tr>
<td>Our Skein</td>
<td>90</td>
<td>15.5</td>
<td>512</td>
<td>N.A.</td>
<td>592</td>
<td>86</td>
</tr>
</tbody>
</table>

a) This compact core uses an external memory to hold the message block and does not provide salted hashing.
b) This value includes the area of the RAM. The coprocessor uses 5 kGE with external RAM (as reported in Keccak documentation). Including RAM area yields 9.3 kGE.
c) Skein-256-256
d) Skein-512-256

6.4 Results and Discussion

In our study, we achieved better results than most of the previous works in terms of area and throughput. Grostl and BLAKE give the best gate counts. Best throughput numbers are presented by BLAKE and Keccak, while the best results are provided by BLAKE and Keccak in terms of throughput/area.

Note that, except for Keccak, all hash functions have half the internal state size with respect to 512-bit message digest option. Such a normalization for Keccak will result in Keccak-800 – 256, and will yield the best gate count and worst throughput. It is also worth mentioning that the throughput of Grostl can be quadrupled at the expense of an additional 2 kGE (estimated), making it the second best in terms of throughput, while preserving its top position with the smallest area.

Table 6.6 lists our results for all finalists as well as comparison with previous works.
6.5 Conclusion

Since the existing security standards and primitives are most of the time not suitable for deployment in lightweight devices, there have already been several creative implementations of these standards targeted for lightweight applications, in addition to the new lightweight algorithm proposals. While these algorithms have mostly focused on block ciphers, there are also recent examples on lightweight hash functions. Unfortunately, these studies were independent of the SHA-3 standardization process, where the suitability of the candidates for lightweight applications was almost neglected. We believe that the two efforts should somehow be combined or at least associated.

In this work, we made such an association possible through a suitability analysis of the SHA-3 finalists for lightweight applications. We limited our focus on reaching lightweight figures for area, which also results in good numbers for average power consumption in most applications; and tried to get the lowest possible recorded gate counts for all five finalists. Use of block memories is avoided for compatibility on different platforms. We have been successful in reaching our target of lowest gate count, and we have presented very compact implementations of SHA-3 finalists. However, despite our lightweight approach, SHA-3 finalists are still larger compared to lightweight block ciphers (see Chapter 4). We believe that this is a result of having no specific lightweightness target for SHA-3 candidates. Other hash functions, which are specially-tailored for lightweight applications, such as PHOTON, QUARK, and SPONGENT (which follow sponge construction like Keccak), result in better figures compared to our serial implementations of SHA-3 finalists.
Chapter 7

IPSecco: A Lightweight and Reconfigurable IPSec Core

In this chapter, we propose a reconfigurable lightweight IPSec hardware core, which is implemented on FPGA. Our architecture supports the main IPSec protocols; namely Authentication Header (AH), Encapsulating Security Payload (ESP), and Internet Key Exchange (IKE). Instead of re-implementing the common “too heavy” IPSec configurations, which are not suitable for pervasive devices, we evaluate efficient implementations of standardized and/or well-known lightweight and hardware-friendly algorithms. In particular, we examine different versions of PRESENT, hash functions Grøstl and PHOTON, and a very compact ECC core. As a consequence, we present IPSecco, an IPSec core with adequate security and only moderate resource requirements, making it suitable for lightweight devices.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>63</td>
</tr>
<tr>
<td>7.2</td>
<td>Related Work</td>
<td>64</td>
</tr>
<tr>
<td>7.3</td>
<td>Implementation</td>
<td>65</td>
</tr>
<tr>
<td>7.4</td>
<td>Results and Discussion</td>
<td>72</td>
</tr>
<tr>
<td>7.5</td>
<td>Conclusion and Future Work</td>
<td>74</td>
</tr>
</tbody>
</table>

7.1 Introduction

With the technological development in today’s world, more and more computing devices enter the market, as mentioned previously in Chapter 1. Many of them are connected over the Internet or local networks in order to communicate with each other. Naturally, an increasing number of these devices are also resource-constrained devices and used in pervasive computing applications. These lightweight devices also need to connect to the Internet for device-to-device communication and product updates – mostly over a wireless network. There must be a common language for devices across networks to share information with each other. The Internet Protocol (IP) [RFC1, RFC2, RFC3, RFC4] is the primary communication protocol for transferring data between parties across a network. It defines datagram structures that are encapsulating the data to be delivered. For resource-constrained environments like embedded systems, there is even a lightweight version of IP, namely lwIP [lw1], which reduces the resource utilization.
Nevertheless, the rapid increase in the utilization of these computing devices led to security problems also in communication over networks. Nowadays, many devices require authentication and encryption of the data they receive and send. The same is valid for area-constrained devices, many of them provide critical data. For devices with no resource limitation, a security enhancement to IP called IPSec [RFC2401, RFC2402, RFC2403], has already been proposed. IPSec defines a family of protocols to provide security services such as confidentiality – to prevent undesired access attempts to the data transmission, data integrity – to make sure that the transferred data is not changed, and authentication – to identify the information source. In IPSec, there are different protocols to provide mentioned services. For instance, the AH protocol provides data authentication. The ESP protocol defines mechanisms for confidentiality and data integrity. Finally, the IKE protocol is used for establishing secure connections. These protocols use different cryptographic primitives such as encryption, hashing, and modular arithmetic in order to provide security services. A minimum set of algorithms, which must be supported in an IPSec implementation for AH [ESP] and IKE protocols, was defined in “Cryptographic Suites for IPSec” [RFC2401, RFC2402, RFC2403] for standardization purposes. For example, in “Cryptographic Suite B” [RFC2401, RFC2402, RFC2403], the AES cipher is used in Galois/Counter Mode (GCM) to provide authenticated encryption. The Hash-based Message Authentication Code (HMAC) [RFC5869] construction is used with the SHA for AH services. For exchanging keys between parties, IKE uses the Diffie-Hellman key exchange [RFC5996]. However, none of these recommendations are actually targeted at resource-constrained devices. To the best of our knowledge, there exists no standardized lightweight IPSec protocol in the literature. In this work, in order to have a lightweight IPSec core, we exchange regular algorithms with lightweight ones – standardized and/or well-known lightweight algorithms. For instance, we use PRESENT instead of AES. As hash functions, we evaluate SHA-3 finalist Grøstl and the lightweight hash function proposal PHOTON, which uses the PRESENT Sbox.

Due to the mentioned lightweight device constraints, a lightweight IPSec core must be low-cost and low-power. Software solutions suffer from low performance (when compared to hardware) and some hardware implementations, such as ASIC implementations, lack the flexibility and programmability offered by software. Hence, using an FPGA platform for the designed hardware core seems to be the perfect solution to achieve our goals and stay reconfigurable. We implement both the encryption/hashing algorithms and modes of operation in hardware. Selecting Xilinx Spartan FPGAs as the target platform provided us with reconfigurability, we therefore can switch between different lightweight algorithms or implementations depending on our needs with less effort. For these platforms, we evaluate and propose the IPSecco-80 and IPSecco-128 cores, which are configured to achieve 80-bit and 128-bit symmetric security.

### 7.2 Related Work

Hardware-related implementations of IPSec-specific functionality have unfortunately been relatively scarce, except for a spark of recent activity.

In SRKH, the authors propose an architecture for implementing IPSec on a Xilinx Virtex-4 FPGA board. Cryptographic primitives are realized as reconfigurable coprocessors, which are attached to a MicroBlaze soft-core Xil09. The MicroBlaze is responsible for handling the protocol layer and reconfiguring the coprocessor according to the type of primitive that is required. The supported primitives are programmed into the coprocessor on-demand, i.e., the
7.3 Implementation

Our IPSecco implementation supports a tight coupling with a soft-core, such as Xilinx' MicroBlaze or PicoBlaze, as depicted in Figure 7.1. The main idea here is to let the soft-core handle the protocol layer of IPSec whereas our core executes all cryptographic operations, thus accelerating the supported cryptographic primitives. Contrary to the mentioned related work, our focus is on providing a small core that is able to handle all IPSec requirements while maintaining a very low footprint.

![Figure 7.1: Integrating IPSecco](image)
Table 7.1: IPSecco configurations

<table>
<thead>
<tr>
<th></th>
<th>AH (HMAC)</th>
<th>ESP (GCM)</th>
<th>IKE</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPSecco-80</td>
<td>PHOTON</td>
<td>PRESENT-80$^R$</td>
<td>ECC (secp160r1)</td>
</tr>
<tr>
<td>IPSecco-128</td>
<td>Grøstl</td>
<td>PRESENT-128$^S$</td>
<td>ECC (secp256r1)</td>
</tr>
</tbody>
</table>

R: RAM-based  
S: Serial

Our core supports PRESENT-80 and PRESENT-128, Grøstl, PHOTON, and ECC. For both PRESENT variants we have implemented RAM-based and serial versions. The RAM-based implementation is taken from the approach that has already been explained in Chapter 5 – in this chapter we also implement the 80-bit key version of that idea. The RAM-based version is targeted for extra low area utilization, while the serial version is optimized for higher speeds (compared to the RAM-based version).

To complement the respective version with a hash function, we use a RAM-based implementation of PHOTON and a serial implementation of Grøstl. Considering the key sizes and different versions, the overall IPSecco core is available in two different configurations (see Table 7.1). IPSecco-80 provides a security level equivalent to 80-bit symmetric security, while the IPSecco-128 core achieves 128-bit security.

The design choices and implementation details of the three distinct blocks (enabling AH, ESP, IKE of IPSecc) of our core are given in the following.

7.3.1 Message Authentication

In order to provide data integrity and authenticity in the AH mode, IPSecc defines a symmetric message authentication code (MAC). The biggest advantage of a MAC is its relatively high speed compared to digital signatures. However, MACs are symmetric so that both parties have to agree on a joint key and also do not ensure confidentiality of the transmitted information. IPSecc requires the utilization of the HMAC construction, which provides a fixed size authentication tag for arbitrary messages and can be provably secure under some circumstances. The HMAC value of a message $X$ is computed as

$$\text{HMAC}_K(X) = H( (K^+ \oplus \text{opad}) || H(K^+ \oplus \text{ipad}) || X )$$

where opad and ipad are padding constants, and $K^+$ is the key $K$ padded to a length equal to the block length of the hash function. In order to make the construction secure, it is required that the key length used is equal or greater than the output length of the employed hash function $H$.

The implementation of an HMAC wrapper, given a hash function $H$, is rather straightforward as can be seen in Figure 7.2. We only require a shift register, implemented as Shift Logical Right (SLR), that stores the key (as it is needed twice) and also the hash output after the first call to the hash function. We have implemented two modern, lightweight hash functions, to be used in our wrapper.
7.3. Implementation

Figure 7.2: Architecture of the HMAC core

PHOTON

One implementation option we provide for IPSec is the lightweight hash function PHOTON, which is designed for hardware efficiency and low area consumption. PHOTON is based on the well-established and well-analyzed sponge framework [BDPA07] and supports several parameter sets for different levels of security. In this work, we have implemented PHOTON-160/36/36, which provides 80-bit collision resistance and has an input block length of 36 bits. The state is represented as a $7 \times 7$ nibble matrix, which is first initialized with some constants. For each message block, the message is XORed with the content of the state and afterwards a permutation is applied (12 times for every input block). This permutation consists of adding constants, applying the PRESENT Sbox to every entry of the state, rotating some rows and finally mixing the columns of the state. After all message blocks have been absorbed, the squeezing phase begins, in which the hash output is generated by further applications of the permutation and the extraction of the parts of the state.

Note that this module is not implemented by the author of this thesis. Our co-author Thomas Pöppelmann implemented the PHOTON module as an Application-Specific Processor (ASP) that stores its program code and also the state in a dual-port block RAM in order to save slice resources. Some operations are directly applied to the state (e.g., Sbox or XORing of constants) and do not involve any registers. Other instructions, like shifting and re-ordering of entries of the state, are performed with the help of two 4-bit registers. One register is also able to execute Galois field arithmetic, which is necessary in the MixColumns layer. Constants are generated by LFSR, which are also used to trigger the execution of static branch instructions and hold the state (e.g., the number of applications of the permutation layer to a message block).

Grøstl

Our serialized Grøstl architecture is taken from the Grøstl design given in Chapter 6 (see Section 6.2.2).
Instead of using our Grøstl implementation as part of HMAC, another possibility is to use a different message authentication method. In their submission to NIST, the authors of Grøstl state that an envelope construction such as

\[ \text{MAC}_K(X) = H(K^+ \| X^+ \| K) \]

gives security similar to HMAC, but requires only one iteration of the hash function and no wrapper.

### 7.3.2 Authenticated Encryption

GCM provides authenticated encryption with low overhead and was originally designed [Dwo07] for 128-bit block ciphers, e.g., AES. However, for lightweight block ciphers such as PRESENT, a modification was proposed [MV05] to extend GCM to ciphers with a block size of only 64-bit.

Encryption with GCM allows for four different inputs: an initialization vector IV, a plaintext P, a key K, and additional data A, which is not encrypted but authenticated. In our notation, A is split into m blocks A₁, A₂, ..., Aₘ of 64 bits and P (and thus also C) in n blocks P₁,...,Pₙ. The output of the encryption step is a ciphertext C₁,...,Cₙ and an authentication tag T, which can be used to validate the integrity of C and A.

We have adapted and slightly modified the GCM proposal for 64-bit block sizes in order to correct what we believe is a flaw in the document. Accordingly, GHASH₆₄(H,A,C) is defined by the recursive relation

\[
X_i = H \cdot \begin{cases} 
0 & i = 0, \\
(X_{i-1} \oplus A_i) & \forall i = 1, ..., m - 1, \\
(X_{i-1} \oplus \mathcal{P}(A_m^n)) & i = m, \\
(X_{i-1} \oplus C_i) & \forall i = m + 1, ..., m + n - 1, \\
(X_{i-1} \oplus \mathcal{P}(C^n_m)) & i = m + n,
\end{cases}
\]

and finally

\[ \text{GHASH₆₄}(H,A,C) = (((X_{m+n} \oplus \mathcal{L}(A)) \cdot H \oplus \mathcal{L}(C)) \cdot H. \]

Here, \( \mathcal{P}() \) returns a zero-padded string of 64 bits, \( \mathcal{L}(\cdot) \) returns the zero-padded bit length of its input. Building on the construction of GHASH₆₄, the authenticated encryption operations are defined by the following equations:

\[
H = E(K, 0^{64})
\]

\[
Y_0 = \begin{cases} 
IV \| 0^{31} \| 1 & \text{if } \text{len}(IV) = 32 \\
\text{GHASH₆₄}(H, \{\}, IV) & \text{otherwise}
\end{cases}
\]

\[
Y_i = \mathcal{T}(Y_{i-1}) \quad \forall i = 1, ..., n
\]

\[
C_i = P_i \oplus E(K, Y_i) \quad \forall i = 1, ..., n - 1
\]

\[
C_n = P_n \oplus \mathcal{M}_a(E(K, Y_n))
\]

\[
T = \mathcal{M}_t(\text{GHASH₆₄}(H,A,C) \oplus E(K,Y_0))
\]

\(^1\)We have tried to contact the authors of the respective document but received no answer.
7.3. Implementation

Here, \( I(\cdot) \) denotes the 32-bit incrementation while \( M_x(\cdot) \) returns the \( x \) most significant bits. In the original proposal [MV05] we find

\[
Y_0 = \begin{cases} 
0^{31}|1||IV & \text{if } \text{len}(IV) = 32 \\
\text{GHASH64}(H, \{\}, IV) & \text{otherwise}
\end{cases}
\]

which, in case the length of the \( IV \) is 32 bits, severely limits the possible counter values \( Y_i \) to \( 2^{32} \) instead of \( 2^{64} \) possibilities.

Figure 7.3 shows the architecture of the PRESENT/GCM core. Since GCM is basically a combination of the well-known Counter (CTR) mode and multiplication/addition in Galois Field (GF) \( GF(2^{64}) \), this is also reflected in our design. Our design can be used for encryption as well as for decryption – both modes include calculating the authentication tag \( T \). In case of decryption, the computed \( T \) has to be verified by the user of our core. Note that Figure 7.3 depicts a simplified view of the implemented architecture.

For IPSecco, we implement PRESENT for both key options, PRESENT-80 and PRESENT-128.
The RAM-based implementation of PRESENT is taken from the approach presented in Chapter 5. However, in Chapter 5 only the design for PRESENT-128 is presented. For our work, we also implement PRESENT-80 – the difference of this design is in the last phase of the operation flow (key schedule). The overall data flow of PRESENT-80 is similar to PRESENT-128; however, in the key scheduling part we have less clock cycles as a result of the shorter key length. The main modification here is in the addressing scheme. As mentioned before, the cycle count is less than PRESENT-128; however, the non-symmetric nature of PRESENT-80’s addressing logic does not allow for a significant reduction of the slice count. In total, PRESENT-80 takes 969 clock cycles while PRESENT-128 takes 1062 clock cycles (note that we have implemented the on-slice Sbox version in this chapter, not the on-RAM Sbox version). As stated above, the data flow for both PRESENT-80 and PRESENT-128 is more or less the same except the key scheduling part. The general block diagram for both versions is given by Figure 7.4.

![Figure 7.4: Architecture of the RAM-based PRESENT implementation](image)

**PRESENT-serial**

In the serial implementations of PRESENT-80 and PRESENT-128, instead of the regular 4-bit datapath as depicted in [RPL+08], we use a 16-bit datapath approach. Using a 4-bit datapath would certainly decrease the slice count as the overall circuit uses less combinational circuitry. However, we also want to have a better performance compared to regular serial implementations. Therefore, we selected the 16-bit datapath for the serial implementations of PRESENT. The data is processed in 16-bit chunks, which means we only update 16 bits of the state in each clock cycle.

As can be seen in Figure 7.5, we have four 16-bit state registers and five 16-bit key registers in serial PRESENT-80 implementation. These registers normally act like shift registers, except during the data load and permutation/key scheduling. We take the data and key input in the
first round and we start processing at the same time. The key addition and substitution are performed at the same time in each cycle on the output of registers $SR_0$ and $KR_0$. Note that we have four 4-bit Sboxes for our 16-bit state in the substitution step. At the end of each round, the permutation is applied on the full state as it is not possible to serialize the permutation step in a cheap way. The same is true for key scheduling, we therefore update the whole key registers at the end of each round, in parallel to permutation.

![Figure 7.5: Architecture of the serial PRESENT-80 implementation](image)

The serial implementations for PRESENT-80 and PRESENT-128 are similar. The main difference of PRESENT-128 is the additional three key registers for key storage. As a result of having a larger key size, the PRESENT-128 implementation takes more clock cycles than PRESENT-80. One round of PRESENT-80 is processed in 5 clock cycles while it takes 8 cycles for PRESENT-128. Furthermore, there is an additional multiplexer at the input of $SR_1$ to select between key-added & Sboxed and unprocessed $SR_0$ output, as we have to wait for a few cycles before the permutation process at the end of each round (which is done in parallel with the key scheduling to preserve the data flow). Figure 7.6 shows the serial PRESENT-128 implementation.

7.3.3 Key Exchange

RFC 6379 [RFCg] defines the recommended cryptographic primitives for IPSec for security levels of 128-bit and above. For key exchange, and a security level equivalent to AES-128, either RSA or use of prime field elliptic curve cryptography is required. We opted for ECC; the given curves are the same as recommended by NIST and earlier by SECG [Sta00]. The naming conventions are slightly different, so the NIST curve $ECC_{p256}$ was formerly known as secp256r1. While NIST only standardized the secp curves from 224-bit upwards, there also exist recommended curves for 160-bit and 192-bit arithmetic, named secp160r1 and secp192r1.

In 2011, a reconfigurable ECC-p core [VGMH] using a microcode approach [VMG+10] was introduced at ReConFig². The design uses a very small arithmetic and logic unit (Arithmetic

²The co-author of the publication related to this chapter, Oliver Mischke, is also a co-author of [VGMH]
Logic Unit (ALU) and a control unit, which is able to read and decode certain instruction sequences from RAM. It is able to perform modular arithmetic from point addition and doubling up to point multiplication. Because the program code (e.g., how to perform doubling, addition, a fast NIST reduction, etc.) and all necessary constants (base point, etc.) are stored in RAM, it is possible to switch from one curve to another just by changing the stored program code and data in the RAM. This can happen on the fly after the initialization of the FPGA.

The ability to change the prime curve for ECC operations just by swapping the memory content makes this design very suitable for less expensive, older FPGAs. This directly is beneficial for the Xilinx Spartan-3 Series, which do not support partial reconfiguration. We have therefore chosen to implement a similar design as [VGM11], but instead of implementing ECC-p224/secp224r1, we implemented secp160r1 alongside ECC-p256/secp256r1 to cover a wider range of security levels. As ECC-p256/secp256r1 provides an equivalent security level as AES-128 or PRESENT-128, secp160r1 is a very good match for lower security cryptographic primitives like the implemented PRESENT-80.

Note that the ECC block described here is implemented by our co-author Oliver Mischke. The architecture of our ECC module is depicted in Figure 7.7 and is, as mentioned before, very similar to the one of [VGM11]. A control unit (PRG CTL) reads and decodes instructions from one RAM and passes that information to the modular ALU. That unit features a small arithmetic unit using a 16-bit datapath, which can perform basic operations like multiplication of two 16-bit values, addition, subtraction, and comparison. It also contains two dual-port RAMs to be able to read both operands and store the result of a computation in a single clock cycle.

Compared to [VGM11], our design is slightly smaller, most likely because we focused only on Xilinx Spartan FPGAs with dedicated Multipliers/Digital Signal Processing (DSP)s. Thus, there was no need to optimize the datapath to make the same design fast on different FPGA architectures.

7.4 Results and Discussion

Table 7.2 and Table 7.3 show our synthesis results for the two representatives of Xilinx’ Spartan family. We have evaluated two major design strategies, RAM-based implementations and serial implementations. The RAM-based approach makes use of the FPGA’s on-board RAM; in
7.4. Results and Discussion

As expected, the resource consumption of our design is significantly lower than that of previously published work on IPSec FPGA implementations. For example, in [SRK11], AES SHA-256, and a modular exponentiation core for elliptic curve Diffie-Hellman was implemented on a Xilinx Virtex-4 FPGA. While the Virtex-4 series is significantly more powerful than our Spartan-3 target, it employs a similar 4-input LUT architecture. Therefore, the slice count as well as LUT and flip-flop usage can be compared to each other. Their AES core uses 1862 slices, while our similarly secure PRESENT-128 implementation in the smallest option (RAM) only requires 84 slices. The area difference is still large when looking at the hash functions; their SHA-256 implementation requires 924 slices, while our similarly secure serial Grøstl only requires 349 slices.

Reducing the security requirements to an equivalent symmetric security of 80 bits would further reduce the necessary slice count, especially since we then could use a low area, RAM-based implementation of PHOTON instead of Grøstl. Note that these numbers alone of course do not allow a fair comparison, since the cores of [SRK11] are most likely designed to allow a higher throughput while we primarily focused on minimizing the slice count. For the modular exponentiation core, the slice count is quite comparable; namely, 499 slices and three DSP48s in [SRK11] versus 569 slices and one 18-bit block multiplier in our design.
### Table 7.2: Characteristics of our implementation for Xilinx Spartan-3

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#Slices</th>
<th>#LUTs</th>
<th>#FFs</th>
<th>#RAMs</th>
<th>#MULs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESENT-80&lt;sup&gt;R&lt;/sup&gt;</td>
<td>84</td>
<td>135</td>
<td>38</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-80&lt;sup&gt;S&lt;/sup&gt;</td>
<td>131</td>
<td>222</td>
<td>153</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-80/GCM&lt;sup&gt;R&lt;/sup&gt;</td>
<td>356</td>
<td>630</td>
<td>448</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-80/GCM&lt;sup&gt;S&lt;/sup&gt;</td>
<td>402</td>
<td>714</td>
<td>561</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-128&lt;sup&gt;R&lt;/sup&gt;</td>
<td>84</td>
<td>130</td>
<td>38</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-128&lt;sup&gt;S&lt;/sup&gt;</td>
<td>161</td>
<td>295</td>
<td>201</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-128/GCM&lt;sup&gt;R&lt;/sup&gt;</td>
<td>356</td>
<td>632</td>
<td>450</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-128/GCM&lt;sup&gt;S&lt;/sup&gt;</td>
<td>443</td>
<td>771</td>
<td>609</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>PHOTON&lt;sup&gt;R&lt;/sup&gt;</td>
<td>91</td>
<td>160</td>
<td>50</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PHOTON/HMAC&lt;sup&gt;R&lt;/sup&gt;</td>
<td>133</td>
<td>235</td>
<td>80</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Grøstl&lt;sup&gt;S&lt;/sup&gt;</td>
<td>349</td>
<td>647</td>
<td>376</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Grøstl/HMAC&lt;sup&gt;S&lt;/sup&gt;</td>
<td>601</td>
<td>1119</td>
<td>693</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ECC-160</td>
<td>569</td>
<td>1028</td>
<td>507</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>ECC-256</td>
<td>569</td>
<td>1028</td>
<td>507</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>IPSecco-80</td>
<td>1107</td>
<td>1986</td>
<td>1123</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>IPSecco-128</td>
<td>1613</td>
<td>2918</td>
<td>1809</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

R: RAM-based  
S: Serial

### 7.5 Conclusion and Future Work

In this chapter, we have presented how lightweight building blocks for the execution of the popular IPSec protocol can efficiently be realized on reconfigurable hardware. Our research shows that even a complex protocol suite, which requires several cipher standards and modes of operation, can be implemented even on very low-cost and energy-efficient Spartan FPGA. By selecting algorithms that are generally designed for efficient realization in hardware, and by adapting them to the specification of our target device, we were able to achieve a significant reduction in terms of area usage compared to related work. With our results we show that standardized security can be achieved on reconfigurable hardware with only a modest investment on slice resources. This leaves more space for other functionality on the target hardware.

As a lightweight IPSec core that will probably be used mostly on low-power devices deployed in the field, an attacker could employ side-channel analysis to obtain secret keys or introduce faults into the crypto cores or management modules. As protection mechanisms against such attacks usually require large amounts of device resources and make analysis hard, we did not consider them within the scope of this work. However, generic protection layers <sup>GM11</sup> can be integrated in future as well as countermeasures that are specifically designed to use in lightweight applications.

Another interesting topic would be the investigation of different algorithms and their combinations, such as CLEFIA, Keccak, HIGHT, etc., and also the primitives that can make use of resource-sharing. Note that a generic IPSec core requires hash functions, block ciphers, as well
7.5. Conclusion and Future Work

Table 7.3: Characteristics of our implementation for Xilinx Spartan-6

<table>
<thead>
<tr>
<th></th>
<th>#Slices</th>
<th>#LUTs</th>
<th>#FFs</th>
<th>#RAMs</th>
<th>#MULs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESENT-80&lt;sup&gt;R&lt;/sup&gt;</td>
<td>26</td>
<td>83</td>
<td>38</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-80&lt;sup&gt;S&lt;/sup&gt;</td>
<td>48</td>
<td>167</td>
<td>153</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-80/GCM&lt;sup&gt;R&lt;/sup&gt;</td>
<td>138</td>
<td>455</td>
<td>451</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-80/GCM&lt;sup&gt;S&lt;/sup&gt;</td>
<td>164</td>
<td>515</td>
<td>572</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-128&lt;sup&gt;R&lt;/sup&gt;</td>
<td>31</td>
<td>82</td>
<td>38</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-128&lt;sup&gt;S&lt;/sup&gt;</td>
<td>62</td>
<td>231</td>
<td>201</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-128/GCM&lt;sup&gt;R&lt;/sup&gt;</td>
<td>141</td>
<td>454</td>
<td>459</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT-128/GCM&lt;sup&gt;S&lt;/sup&gt;</td>
<td>174</td>
<td>512</td>
<td>624</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>PHOTON&lt;sup&gt;R&lt;/sup&gt;</td>
<td>30</td>
<td>89</td>
<td>50</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PHOTON/HMAC&lt;sup&gt;R&lt;/sup&gt;</td>
<td>60</td>
<td>152</td>
<td>80</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Grøstl&lt;sup&gt;S&lt;/sup&gt;</td>
<td>136</td>
<td>411</td>
<td>385</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Grøstl/HMAC&lt;sup&gt;S&lt;/sup&gt;</td>
<td>275</td>
<td>808</td>
<td>760</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ECC-160</td>
<td>221</td>
<td>630</td>
<td>482</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>ECC-256</td>
<td>221</td>
<td>630</td>
<td>482</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>IPSecco-80</td>
<td>424</td>
<td>1301</td>
<td>1101</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>IPSecco-128</td>
<td>670</td>
<td>1950</td>
<td>1866</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

R: RAM-based
S: Serial

as an asymmetric construction. These primitives are occupying valuable resources on the device but are not always executed simultaneously (e.g., HMAC is not needed when only ESP mode with authenticated encryption enabled is used). One common solution for efficient resource-sharing is partial reconfiguration, which is only available on more powerful but also more expensive and larger FPGAs. Therefore, further effort can be put into the integration and sharing of components between different ciphers (e.g., Sbox lookup tables, block RAMs, or registers for keys) in order to further reduce the size of our design.
Part III

New Resource-efficient Constructions for the Ubiquitous Computing Era
Chapter 8

Targeting Low-latency: A Lightweight Block Cipher PRINCE

In this chapter, the design of a new lightweight block cipher that specifically targets low-latency encryption is investigated. The main design target is to keep the gate count below the existing ciphers in the literature (actually, as low as possible while keeping a certain level of security), and the latency at exactly one clock cycle. The entire development process was a major effort that involved many people. However, this chapter focuses on the contributions of the thesis author, which are investigations on hardware cost, implementation of different hardware architectures, performance on ASIC and FPGA platforms and microcontrollers (together with comparison to the performance of other prominent block ciphers in the literature), and hardware cost estimation for possible side-channel resistant constructions of the cipher. In the light of these investigations, we identify a hardware-friendly low-latency cipher, namely PRINCE.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>79</td>
</tr>
<tr>
<td>8.2</td>
<td>PRINCE Background</td>
<td>81</td>
</tr>
<tr>
<td>8.3</td>
<td>Hardware Investigations</td>
<td>88</td>
</tr>
<tr>
<td>8.4</td>
<td>Hardware Implementations</td>
<td>89</td>
</tr>
<tr>
<td>8.5</td>
<td>Software Implementations</td>
<td>93</td>
</tr>
<tr>
<td>8.6</td>
<td>Investigations on Countermeasures Against Potential SCA on PRINCE</td>
<td>94</td>
</tr>
<tr>
<td>8.7</td>
<td>Concluding Remarks</td>
<td>96</td>
</tr>
</tbody>
</table>

8.1 Introduction

The majority of the lightweight ciphers, as mentioned previously, have been optimized according to a dominant metric, which is chip area. This is certainly a valid optimization objective in cases where there are extremely tight power or cost constraints, in particular for passive RFID tags. However, in many other applications, there are several other implementation parameters according to which a cipher should have lightweight characteristics. For instance, instant authentication and block-wise read/write access to memory devices (e.g., solid-state hard disks) need low-latency encryption and instant response time.
For such cases, there is a certain demand for symmetric ciphers that can instantaneously encrypt/decrypt a given plaintext/ciphertext. Here, instantaneous means that the entire encryption and decryption should take place within the shortest possible delay. This seemingly simple problem poses a considerable challenge with today’s cryptosystems – in particular if encryption and decryption should both be available on a given platform. Software implementations of virtually all strong ciphers take hundreds or thousands of clock cycles, making them ill-suited for a designer aiming for low-latency cryptography. In the case of stream ciphers implemented in hardware, the high number of clock cycles for the initialization phase makes them unsuitable for this task, especially when secret keys need to be regularly changed. Moreover, if we want to encrypt small blocks selected at random (e.g., encryption of sectors on solid-state disks), stream ciphers are not convenient.

This leaves block ciphers as the remaining viable solution. However, the round-based, i.e., iterative, nature of virtually all existing block ciphers, as shown for the case of AES makes low-latency implementation a non-trivial task. A round-based hardware architecture of AES-128 requires ten clock cycles to output a ciphertext, which we do not consider instantaneous as it is still too long for some applications. As a remedy, the ten rounds can be loop-unrolled, which means the circuit realizing a single round is repeated ten times. Now, the cipher returns a ciphertext within a single clock cycle – but at the cost of a very long critical path. This yields a very slow absolute response time and clock frequency, e.g., in the range of a few MHz. Furthermore, the unrolled architecture has a high gate count in the range of several tens of thousand implying a high power consumption and resource utilization. Both of these are undesirable, especially when the intended applications are in the embedded domain.

Following the same motivation and reasoning with this work, Knežević et al. [KNR12] compared several lightweight ciphers with respect to latency and as a conclusion they called for new designs that are optimized for low-latency. As a summary of the above discussions, the following criteria are of importance for such new low-latency cipher designs:

1. The cipher should perform instantaneous encryption, i.e., a ciphertext should be computed within a single clock cycle. There should be no warm-up phase.

2. When implemented in modern chip technology, low delays (resulting in moderately high clock rates) should be achieved.

3. The hardware costs should be moderate (i.e., considerably lower than fully-unrolled versions of AES or PRESENT).

4. Encryption and decryption should both be possible with low-cost and low-overhead.

Here, it is important to point out that the existing lightweight block ciphers, such as PRESENT, do not fulfill Criteria 2 and 3 (low-latency, low-area) due to their large number of rounds. Therefore, the number of rounds should be set in a way to address these two criteria without sacrificing security. Another crucial point worth mentioning is the need for (almost) identical hardware utilization for encryption and decryption in order to fulfill Criterion 4. This is an important requirement since the unrolled nature of instantaneous ciphers leads to circuits

\[^1\]A possible exception are random-access stream ciphers such as Salsa [Ber08]
that are large and it is thus clearly advantageous if large parts of the implementation can be used both for encryption and decryption.

Taking all of these into account, we set our goal to design a new block cipher optimized with respect to the given criteria when implemented in hardware. In the following sections, we introduce our low-latency lightweight block cipher proposal PRINCE and present the details of the specific parts of the design process.

8.2 PRINCE Background

The development process of PRINCE was a major effort involving many people from Ruhr University Bochum, Technical University of Denmark, and NXP Semiconductors. We present the specification of the PRINCE cipher and explain its necessary design details briefly in this section. The details of the individual contributions of the co-authors can be found in the respective publications [BCG+12a, BCG+12b].

8.2.1 Highlights of PRINCE

Before going into the details, we would like to highlight some important features of PRINCE. Besides being a new lightweight cipher for the first time optimized with respect to the goals above, PRINCE has several innovative features.

First of all, a fully-unrolled design increases the possible design choices enormously. With a fully-unrolled cipher, the traditional need for a cipher to be iterative with very similar round functions disappears. In turn, that allowed us to efficiently implement a cipher where decryption with one key corresponds to encryption with a related key. This property, we refer to as \( \alpha \)-reflection, is of independent interest and its soundness against generic attacks is proven in the original PRINCE publication [BCG+12a], which was presented at ASIACRYPT in 2012 (also, a full version is available on IACR Cryptology ePrint Archive [BCG+12b]). As a consequence of the \( \alpha \)-reflection, the overhead of implementing decryption over encryption became negligible. Note that the previous approaches to minimize the overhead of decryption over encryption, for example in the ciphers NOEKEON [DPAR00] and ICEBERG [SPR+04], usually require a multiplexer in each round. While for a round-based implementation this does not make a difference, our approach is clearly preferable for a fully-unrolled implementation, as we require a multiplexer only once at the beginning of the circuit.

Another difference to known lightweight ciphers like PRESENT is that we balanced the cost of the substitution layer and the linear layer. Our investigations on hardware cost, which will be explained in detail in the following sections, showed that optimizing the cost of the chosen Sbox has a major influence on the overall cost of the cipher. As an Sbox that performs well in one technology does not necessarily perform well in another technology, we propose the PRINCE-family of ciphers that allows to choose the Sbox freely within a (large) set of Sboxes fulfilling the certain criteria. In our work, we favored one Sbox, which was one of the smallest in all three different technologies we used.

Our choice for the linear layer can be seen as a solution “in-between” the bit-permutation layers of PRESENT (implemented with wires only) and [AES] (implemented with considerable combinatorial logic). With the expense of only 2 additional XOR gates per bit over a simple bit-permutation layer, we achieved an almost- MDS property that helps to prove much better
bounds against various classes of attacks. This in turn allowed us to significantly reduce the number of rounds, hence the latency.

As a result of all these properties, PRINCE compares very favorable to existing ciphers. For the same time constraints and technologies, PRINCE uses approximately 6 times less area than PRESENT-80 and 14-15 times less area than AES-128. In addition to this, our design uses about 4-5 times less area than the other ciphers in the literature (see Section 8.4 and in particular Tables 8.2 and 8.3 for a detailed comparison in different technologies). To facilitate further study and fairer comparisons, we reported our synthesis results using also the open-source standard-cell library NANGATE. Note that, although it is not the main objective of the cipher, PRINCE compares reasonably well to other lightweight ciphers also when implemented in a round-based fashion.

8.2.2 Cipher Description

PRINCE is a 64-bit block cipher with a 128-bit key. The key is split into two parts of 64 bits each,

\[ k = k_0 || k_1 \]

and extended to 192 bits by the mapping

\[(k_0 || k_1) \rightarrow (k_0 || k'_0 || k_1) := (k_0 || ((k_0 \gg 1) \oplus (k_0 \gg 63))) || k_1).\]

PRINCE is based on the so-called FX-construction [Bir05, KR96]: The first two subkeys \(k_0\) and \(k'_0\) are used as whitening keys, while the subkey \(k_1\) is the 64-bit key for a 12-round block cipher we refer to as PRINCE\(_{\text{core}}\). This scheme is depicted in the figure below.

The detailed test vectors for PRINCE encryption and decryption are given in the appendix (Part V).

**Specification of PRINCE\(_{\text{core}}\)**

The following figure shows the whole encryption process of PRINCE\(_{\text{core}}\).
As can be seen from the figure, PRINCE\textsubscript{core} consists of 5 regular rounds, a middle layer, 5 inverse rounds, together with round constant and subkey additions at the beginning and at the end. Regular round and inverse round are similar, only the functions and their order are inverted. Each round of PRINCE\textsubscript{core} consists of a subkey addition, a substitution layer, a linear layer, and the addition of a round constant.

- **Subkey Addition**
  Here the 64-bit state is XOR ed with the 64-bit subkey $k_1$.

- **Substitution Layer**
  In the substitution layer, the state is divided into 16 nibbles and these nibbles go through 16 same 4-bit Sboxes. Inverse of this Sbox is used for the inverse round. The PRINCE Sbox and its inverse in hexadecimal notation is given in the following table.

<table>
<thead>
<tr>
<th>$x$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S[x]$</td>
<td>B</td>
<td>F</td>
<td>3</td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>9</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>E</td>
<td>5</td>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>$S^{-1}[x]$</td>
<td>B</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>F</td>
<td>D</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Linear Layer**
  In the round linear layer, the 64-bit state is multiplied with a $64 \times 64$ matrix $M$, which is defined in Section 8.2.3. Inverse of this matrix, $M^{-1}$, is used for the inverse round.

- **Round Constant Addition**
  In this step, a 64-bit round constant $RC_i$ is XOR ed with the state. We define PRINCE round constants below in hexadecimal format.

| $RC_i$ | 0000000000000000 |
| $RC_1$ | 13198a2e03707344 |
| $RC_2$ | a493822299f31d0 |
| $RC_3$ | 082efa98ec6e89 |
| $RC_4$ | 452821e638d01377 |
| $RC_5$ | be5466c34e90c6c |
| $RC_6$ | 7e84f78fd955c1 |
| $RC_7$ | 85840851f1ae43aa |
| $RC_8$ | c882d32f25323c54 |
| $RC_9$ | 6a51195e0e3610d |
| $RC_{10}$ | d3b5a399ca0c2399 |
| $RC_{11}$ | c0ac29b7c97c50dd |

Note that, for all $0 \leq i \leq 11$, $RC_i \oplus RC_{11-i}$ is the constant $\alpha = c0ac29b7c97c50dd$. We define $RC_0 = 0$ and the first five round constants $RC_1, \ldots, RC_5$ together with $\alpha$ are derived from the fraction part of $\pi = 3.141\ldots$, in order to ensure a random-like behavior while at the same time being publicly verifiable. The second issue can nevertheless be ignored, but any randomly-generated round constants should be a valid choice.

- **Middle Layer**
  In the middle layer, the 64-bit state first goes through a substitution layer, then gets multiplied with a $64 \times 64$ matrix $M'$ defined in Section 8.2.3 and finally goes through an inverse substitution layer.
8.2.3 Design Decisions

This section discusses the design decisions of PRINCE.

In our design, we opt for a traditional Sbox-based – and especially an SPN – construction. Note that in our case, which is a low-latency design, an SPN cipher is preferable over a Feistel-cipher, since a Feistel-cipher operates only on half the state resulting often in a higher number of rounds. In order to minimize the number of rounds and still achieve security against linear and differential attacks, we adopted the wide-trail strategy [Dae95].

As not all round functions have to be identical for a cipher aiming for a fully-unrolled implementation as PRINCE, it is very tempting to directly use the concept of “code-concatenation” [DR09] to achieve a high number of active Sboxes over 4 rounds of the cipher. However, similar round functions are not only beneficial for a serial implementation, but also very helpful for ensuring a minimum number of active Sboxes. Assume that, using the code-concatenation approach, one can ensure that rounds $R_i$ to $R_{i+3}$ have at least 16 active Sboxes. While this is nice, the problem is that it does not ensure that rounds $R_{i-1}$ to $R_{i+2}$ or $R_{i+1}$ to $R_{i+4}$ have 16 active Sboxes as well if the individual rounds are very different in nature. We therefore decided to follow a design that on one hand allows to use the freedom given by a fully-unrolled design and on the other hand still keeps the round functions similar enough to prove some bounds on the resistance against linear and differential attacks.

In this context, one of the main features of the design is that the decryption can be implemented on top of the encryption with a minimal overhead. This is achieved by designing a cipher that is symmetric around the middle round, has a very simple key scheduling and a special choice of round constants.

As the round constants satisfy $RC_i \oplus RC_{11-i} = \alpha$ and $M'$ is an involution (see Section 8.2.3), we deduce that the core cipher is such that the inverse of “PRINCEcore parametrized with $k$” is equal to “PRINCEcore parametrized with $(k \oplus \alpha)$”. We call this property of PRINCEcore the $\alpha$-reflection property. For the details and analysis of this property, we refer to our main PRINCE publications [BCG+12a, BCG+12b]. Note that this is also a work in progress that is investigated by the co-authors of the PRINCE publication, more details might be available in near future.

From $\alpha$-reflection property, it follows that for any expanded key $(k_0||k_0'||k_1)$, $D_{(k_0||k_0'||k_1)}(\cdot) = E_{(k_0'||k_0||k_1 \oplus \alpha)}(\cdot)$

where $\alpha$ is the 64-bit constant $\alpha = c0ac29b7c97c50dd$.

Thus, one only has to do a very cheap change to the master key and afterwards reuse the exact same circuit for decryption.

Aligning Encryption with Decryption

The use of a core cipher having the $\alpha$-reflection property together with two additional whitening keys offers a nice alternative to the usual design strategy, which consists of using involutional components – NOEKEON, Khazad, Anubis, ICEBERG, or SEA [SPGQ06] are some examples of such ciphers with involutional components. Actually, the general construction used in PRINCE has the following advantages:

- It allows a much larger choice of Sboxes, which may lead to a lower implementation cost, since the Sbox is not required to be an involution. It is worth noticing that the fact that
both the Sbox and its inverse are involved in the encryption function and it does not affect the cost of the fully-unrolled implementations we consider.

■ In ciphers with involutonal components, the overhead due to the implementation of the inverse key scheduling can be reduced by adding some symmetry in the subkey sequence. But this may introduce weak keys or potential slide attacks. The fact that all components are involutions may also introduce some regularities in the cyclic structure of the cipher, which can be exploited in some attacks [Bir03]. The resistance of PRINCE to these type of attacks are extensively discussed in our main PRINCE publications [BCG+12a, BCG+12b].

■ It is an open problem to prove the security of ciphers with ideal, involutonal components against generic attacks. The main PRINCE publications [BCG+12a, BCG+12b] show that the ciphers with the $\alpha$-reflection property (for $\alpha \neq 0$) has a proof of security similar to that of the FX-construction.

■ Previous approaches to minimizing the overhead of decryption over encryption usually require multiplexers in each round while our approach requires a multiplexer only once at the beginning of the circuit.

The impact of these advantages can be seen better in the hardware implementations, which are presented in Section 8.4.

Choosing the Sbox

The cost of the Sbox, i.e., its area and critical path, is a substantial part of the overall cost (see Section 8.3). Thus, choosing an Sbox that minimizes those costs is crucial for obtaining competitive results. As the cost of an Sbox depends on various parameters, such as the technology, the synthesis tool, and the library used, one cannot expect that there is one optimal Sbox for all environments. In fact, in order to achieve optimal results, it is preferable to choose your favorite Sbox.

In order to ensure the security of the resulting design, an Sbox $S : \mathbb{F}_2^4 \rightarrow \mathbb{F}_2^4$ for the “PRINCE-family” has to fulfill the following criteria:

(1) The maximal probability of a differential is $1/4$.
(2) There are exactly 15 differentials with probability $1/4$.
(3) The maximal absolute bias of a linear approximation is $1/4$.
(4) There are exactly 30 linear approximations with absolute bias $1/4$.
(5) Each of the 15 non-zero component functions has algebraic degree 3.

As it can be deduced, e.g., from [LP07], there are only 8 Sboxes fulfilling those criteria up to affine equivalence. Thus, another way of defining an Sbox for the PRINCE-family is to say that it has to be affine equivalent to one of the eight Sboxes $S_i$ given in Table 8.1.

Note that $S_0$ is equivalent to the inverse function in $\mathbb{F}_{16}$ and the defined PRINCE Sbox is equivalent to $S_7$. 
Table 8.1: All Sboxes for the PRINCE-family up to affine equivalence

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0, 0x1, 0x2, 0xD, 0x4, 0x7, 0xF, 0x6, 0x8, 0xC, 0x5, 0x3, 0xA, 0xE, 0xB, 0x9</td>
<td>0x0, 0x1, 0x2, 0xD, 0x4, 0x7, 0xF, 0x6, 0x8, 0xC, 0x9, 0xB, 0xA, 0xE, 0x5, 0x3</td>
<td>0x0, 0x1, 0x2, 0xD, 0x4, 0x7, 0xF, 0x6, 0x8, 0xC, 0xB, 0x9, 0xA, 0xE, 0x3, 0x5</td>
<td>0x0, 0x1, 0x2, 0xD, 0x4, 0x7, 0xF, 0x6, 0x8, 0xC, 0x9, 0xB, 0xA, 0xE, 0x9, 0x3, 0x5</td>
<td>0x0, 0x1, 0x2, 0xD, 0x4, 0x7, 0xF, 0x6, 0x8, 0xC, 0xE, 0xB, 0xA, 0x9, 0x3, 0x5</td>
<td>0x0, 0x1, 0x2, 0xD, 0x4, 0x7, 0xF, 0x6, 0x8, 0xC, 0x9, 0xB, 0xA, 0x5, 0x9, 0xC, 0x3</td>
<td>0x0, 0x1, 0x2, 0xD, 0x4, 0x7, 0xF, 0x6, 0x8, 0xC, 0x9, 0x5, 0xB, 0xA, 0x3, 0x5</td>
<td>0x0, 0x1, 0x2, 0xD, 0x4, 0x7, 0xF, 0x6, 0x8, 0xC, 0x9, 0xB, 0xA, 0x5, 0x9, 0x3, 0x5</td>
</tr>
</tbody>
</table>

Construction of Diffusion Matrices

As mentioned before, the 64-bit state is multiplied with a $64 \times 64$ matrix $M$ in the round linear layer, and $M'$ in the middle layer, which are defined below.

We have different requirements for these two different linear layers. The $M'$-layer is only used in the middle round, thus $M'$ has to be an involution to ensure the $\alpha$-reflection property. This requirement does not apply for the $M$-layer used in the round functions. Here we want to ensure full diffusion after two rounds. To achieve this, we combine the $M'$-mapping with an application of matrix $SR$ that behaves like the AES ShiftRows and permutes the 16 nibbles in the following way:

$$0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 \rightarrow 0 | 5 | 10 | 15 | 4 | 9 | 14 | 3 | 8 | 13 | 2 | 7 | 12 | 1 | 6 | 11$$

Thus, $M = SR \circ M'$.

In addition, as the implementation costs should be minimized, the number of ones in the matrices $M'$ and $M$ should be minimal, while at the same time guaranteeing that at least 16 Sboxes are active in 4 consecutive rounds (see [BCG+12a] [BCG+12b] for details). So, trivially each output bit of an Sbox has to influence 3 Sboxes in the next round and therefore the minimum number of ones per row and column is 3. Hence, we can use the following four $4 \times 4$ binary matrices as building blocks for the $M'$-layer:

$$M_0 = \begin{pmatrix} 0000 \\ 0100 \\ 0010 \\ 0001 \end{pmatrix}, \quad M_1 = \begin{pmatrix} 1000 \\ 0000 \\ 0100 \\ 0001 \end{pmatrix}, \quad M_2 = \begin{pmatrix} 1000 \\ 0100 \\ 0000 \\ 0001 \end{pmatrix}, \quad M_3 = \begin{pmatrix} 1000 \\ 0100 \\ 0010 \\ 0000 \end{pmatrix}.$$

In the next step, we generate $4 \times 4$ block matrices $\hat{M}^{(i)}$, where each row and column is a permutation of the four $4 \times 4$ matrices $M_0, \ldots, M_3$. The row permutations are chosen such that we obtain a symmetric block matrix. The choice of the building blocks and the symmetric structure ensures that the resulting $16 \times 16$ binary “AES MixColumns-like” matrices are involutions. We define:
\[
\hat{M}^{(0)} = \begin{pmatrix}
M_0 & M_1 & M_2 & M_3 \\
M_1 & M_2 & M_3 & M_0 \\
M_2 & M_3 & M_0 & M_1 \\
M_3 & M_0 & M_1 & M_2 \\
\end{pmatrix},
\hat{M}^{(1)} = \begin{pmatrix}
M_1 & M_2 & M_3 & M_0 \\
M_2 & M_3 & M_0 & M_1 \\
M_3 & M_0 & M_1 & M_2 \\
M_0 & M_1 & M_2 & M_3 \\
\end{pmatrix}.
\]

Note once again that the following criteria were taken into account for the construction of these 16 × 16 binary MixColumns-like matrices:

- Branch number 4.
- Exactly 3 ones in each row and column.
- Exactly 3 ones in each 4 × 4 sub-block.
- Involution.

In order to obtain a permutation for the full 64-bit state, we construct a 64 × 64 block diagonal matrix \( M' \) with \( (\hat{M}^{(0)}, \hat{M}^{(1)}, M^{(1)}, M^{(0)}) \) as diagonal blocks. The matrix \( M' \) is an involution with \( 2^{32} \) fixed points, which is average for a randomly chosen involution \([FS09, \text{Page 596}]\). The linear layer \( M \) is not an involution any more due to the composition of \( M' \) and ShiftRows. Both matrices are given in the appendix (Part V). Also, more details on linear layer design can be found in the original publications \([BCG^{+}12a, BCG^{+}12b]\).

**The Key Expansion**

The 128-bit key \((k_0||k_1)\) is extended to a 192-bit key \((k_0||k_0'||k_1)\) by a linear mapping of the form

\[
(k_0||k_1) \mapsto (k_0||P(k_0)||k_1).
\]

This expansion should make the peeling of rounds (both at the beginning and at the end) by partial key guessing difficult for the attacker. A hardware-optimal choice for \( P \) is

\[
P(x) = (x \gg 1) \oplus (x \gg 63),
\]

i.e., \( P(x_{63}, \ldots, x_0) = (x_0, x_{63}, \ldots, x_2, x_1 \oplus x_{63}) \). More details on the design decisions of key expansion is given in the original publication \([BCG^{+}12a, BCG^{+}12b]\).

**8.2.4 Security Analysis**

The security analysis of PRINCE is not in the scope of this thesis. Due to the interdisciplinary nature of this work, the author was focused on the selection and design of efficient building blocks for the cipher, together with the implementations. Therefore, we leave the security discussion of PRINCE to the original publications \([BCG^{+}12a, BCG^{+}12b]\), in which the security claims for PRINCE and a detailed security evaluation of its general construction regarding classical attacks are provided. Note that these attacks do not only include linear, differential, and algebraic attacks, but also the recently-introduced biclique attacks \([BKR11]\).

It is worth mentioning that the performed security analysis together with third-party analyses have not revealed any critical and/or practical weaknesses of PRINCE so far.
8.3 Hardware Investigations

Besides the main target low-latency, low-cost hardware implementation is one of the design objectives of PRINCE. To achieve low-latency, a fully-unrolled design is considered for PRINCE, which makes it very difficult to come up with a low-cost implementation if the existing solutions for the block cipher layers are used. However, with “carefully-tailored” new solutions for PRINCE components, the given targets can be achieved. Therefore, during the design process of PRINCE, the cost of each function was investigated exhaustively and each component was designed in order to get the lowest possible gate count without compromising security.

One of the most critical and expensive operations of the cipher is the substitution, where we use the same Sbox 16 times (rather than having 16 different Sboxes). Therefore, the design efforts of PRINCE started with an Sbox search, to find the most suitable one for the target design specifications. We analyzed many Sbox instances to identify a candidate with optimal combinational logic and propagation paths. In order to achieve an implementation with low-latency and a low gate count, the targeted unrolled design is implemented with the resulting optimal Sbox.

The hardware cost of the second critical part, linear layer, is also investigated carefully. The underlying diffusion matrices are designed with the least possible number of ones, in turn XOR operations, in order to keep the gate count as low as possible while keeping the security at the desired level.

In the following sections, we explain both processes in detail.

8.3.1 Searching for the Optimal Sbox

As mentioned before, Sboxes are the most critical building blocks in existing block cipher implementations, in terms of both area and delay. In our design, we tried to overcome this area and propagation delay problem. Unfortunately, neither of the existing approaches on Sboxes and their implementations [SSA+07, HAHH06, RPP08, KY10, Rij00, MPL+11, GC09, Can05] is applicable to PRINCE, where the design target is an unrolled cipher with low-area and low-latency. The only viable solution is to implement a very low gate count combinational 4-bit Sbox with optimal propagation paths. Therefore, a brute-force-like search is performed to find such an Sbox. We followed a two-step approach in our “search for an optimal Sbox for PRINCE”.

Firstly, we generated 64000 candidate Sboxes, which are based on the 8 core Sboxes satisfying the properties introduced in Section 8.2.3. These 64000 Sboxes are simply the affine equivalents of the core Sboxes. Then, all Sboxes and their inverses (in total 128000) are synthesized for minimum area. As PRINCE is mainly targeted for ASIC platforms, the synthesis is performed on ASIC with different technology libraries. In the analysis process, Cadence Encounter RTL Compiler v10.1 is used for synthesis. Since the gate count and delay parameters are heavily technology-dependent, the implementations have been synthesized for two different technology libraries: 90 nm low-leakage Faraday Library from UMC, and 45 nm generic NANGATE Open Cell Library. In all syntheses, typical operating conditions are assumed. The synthesis took approximately 15 days (nearly one week for synthesis in each technology) on our cluster. Figure 8.1 shows the distribution of the Sboxes, inverse Sboxes, and the average of both with respect to gate counts in GE for 45 nm NANGATE and 90 nm UMC technologies. Note that the vertical axis “Number of Sboxes” denotes the Sbox frequency for the corresponding GE.
Figure 8.1: Sbox distributions with respect to gate counts

Following the first – so-called – elimination step, the smallest 100 Sboxes on both technologies are selected and then re-synthesized for minimum power and minimum delay. The reason for the elimination was mainly the synthesis time of the Sboxes – it would take even longer with the given restrictions. This process is repeated for three different process technologies, this time also including 130 nm low-leakage Faraday Library from UMC. The only Sbox that is in top 10 in terms of area, power, and speed in all three technologies is selected to be used in PRINCE, which was previously given in Section 8.2.2.

8.3.2 Low-cost Linear Layer

The linear layer applications in PRINCE can actually be defined as binary matrix multiplications. As a result, the HW of the multiplied matrix gives the number of XOR operations, which estimates the cost of the linear layer (excluding the synthesizer optimizations). We therefore tried to keep the number of ones in our diffusion matrices as low as possible. As already described in Section 8.2.3, the two 64 × 64 diffusion matrices that we use in PRINCE are made up of 4 16 × 16 binary matrices. Each of these matrices have 3 ones per row and column, which is a requirement for the needed security level, as explained before. In total, the HW of each diffusion matrix is 192, which means that the cost of our linear layer is approximately equal to the cost of 192 XOR gates. The cost in GE of course differs for each technology library; however, the cost of an XOR gate is nearly 2.25 – 2.5 GE for most of the technology libraries. Hence, each application of the linear layer costs approximately 450 GE. Except for the zero-cost wiring, which is the case for PRESENT’s permutation layer in hardware implementations, our linear layer is one of the cheapest designs for the given security level. More realistic results (synthesis output) is presented in the following section.

8.4 Hardware Implementations

In this section, we provide the hardware implementations of PRINCE. Both unrolled (which is the real target), and round-based architectures of PRINCE are considered for implementation. Together with the results, a comparison of PRINCE to other block ciphers in literature is also provided.
In the implementation process, *Cadence NCVerilog 06.20-p001* is used for simulation and *Cadence Encounter RTL Compiler v10.1* is used for synthesis. Since the gate count and delay parameters are heavily technology dependent, the implementations have been synthesized for three different technology libraries: 130 nm and 90 nm low-leakage Faraday Libraries from UMC, and 45 nm generic NANGATE Open Cell Library. All ciphers are implemented in Verilog-HDL and typical operating conditions were assumed in all syntheses.

### 8.4.1 Unrolled Architecture

The unrolled version of PRINCE is a direct mapping of the cipher defined in Section 8.2.2 to hardware. Multiplexers, which select the encryption and decryption keys accordingly, are the only encryption/decryption overhead. The only costs associated with the key whitening stages are XOR gates and multiplexers used for whitening key selection. However, in practice, due to the unrolled nature of the implementation, these additions reduce to XOR operations with constants, which in turn reduce to inverters or no additional gates at all. Furthermore, these inverters are combined with the preceding or following matrix multiplications, which are implemented with cascaded XOR gates. In cases where an XOR is sourced by the output from an inverter, or is sourcing the input of an inverter, it is simply replaced by an Exclusive NOR (XNOR) gate and the sourced/sourcing inverter is removed. Since both XOR and XNOR have the same gate count, the overall effect of the round constant addition on area reduces to zero.

Unrolled implementation results are listed in Table 8.2 for different technologies with respect to different timing constraints. Together with PRINCE, we also implemented PRESENT-80, PRESENT-128, LED-128, and AES-128. The same metrics are also applied to these ciphers in order to adequately evaluate the achievements of our new cipher (note that in some cases the key size – and also our security claim – is different: PRINCE does not claim to offer 128-bit security and security against related key-attacks). In order to achieve both encryption and decryption capability in PRESENT and LED, both Sboxes and inverse Sboxes are implemented and their output is selected by a multiplexer. This, in turn, doubled the Sbox area with respect to an encryption-only implementation. For AES we just had to implement the inverse affine transform since the finite field inversion module could be shared between encryption and decryption.

In addition to this comparison, Table 8.3 shows the extrapolated results (which are calculated by removing register and control logic area from the total gate count, and multiplying the rest by the number of rounds) for other unrolled cipher instances obtained from round-based cipher implementations provided by previous works. Note that all ciphers in the table include encryption and decryption functionality with 128-bit key size, however the comparison is difficult as the block size is different in some cases (also note that the ciphers having 128-bit block size are obviously much bigger and more power consuming than a 64-bit block cipher).

We also measured the maximum frequencies achievable by the unrolled versions of PRINCE under two different conditions: The frequency where the area of the synthesized design starts to deviate from the unconstrained area – 158.9, 38.4 and 35.5 MHz, and the frequency where the timing slack becomes zero – 212.8, 71.8 and 54.3 MHz. Both figures are given for 45 nm, 90 nm, and 130 nm, respectively.
8.4. Hardware Implementations

Table 8.2: Area/power comparison of unrolled versions of PRINCE and other ciphers

<table>
<thead>
<tr>
<th>Tech. Constr.(ns)</th>
<th>Nangate 45 nm Generic</th>
<th>UMC 90 nm Faraday</th>
<th>UMC 130 nm Faraday</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINCE</td>
<td>Area(GE)</td>
<td>Power(mW)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8260</td>
<td>38.5</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>8263</td>
<td>17.9</td>
<td>3162</td>
</tr>
<tr>
<td></td>
<td>8263</td>
<td>8.3</td>
<td>10000</td>
</tr>
<tr>
<td></td>
<td>7996</td>
<td>26.3</td>
<td>3162</td>
</tr>
<tr>
<td></td>
<td>3969</td>
<td>10.9</td>
<td>10000</td>
</tr>
<tr>
<td></td>
<td>7966</td>
<td>3.9</td>
<td>29.8</td>
</tr>
<tr>
<td></td>
<td>7966</td>
<td>11.8</td>
<td>11.8</td>
</tr>
<tr>
<td>PRESENT-80</td>
<td>Area(GE)</td>
<td>Power(mW)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>63942</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>51631</td>
<td>3162</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50429</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>513062</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49723</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49698</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>119196</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>51700</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>51790</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>PRESENT-128</td>
<td>Area(GE)</td>
<td>Power(mW)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>68908</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56668</td>
<td>3162</td>
<td></td>
</tr>
<tr>
<td></td>
<td>55467</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>51271</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14911</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>99.1</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>142071</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>54576</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>54525</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>126351</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56732</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56722</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>LED-128</td>
<td>Area(GE)</td>
<td>Power(mW)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>109811</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>109958</td>
<td>3162</td>
<td></td>
</tr>
<tr>
<td></td>
<td>109697</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>281240</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>286779</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>98100</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>236770</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>235106</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>AES-128</td>
<td>Area (GE)</td>
<td>Power (mW)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>135051</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>135093</td>
<td>3162</td>
<td></td>
</tr>
<tr>
<td></td>
<td>118440</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>421997</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>410835</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>347860</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>141060</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>130764</td>
<td>10000</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.3: Extrapolated area of unrolled versions of other ciphers against PRINCE

<table>
<thead>
<tr>
<th>CLEFIA-128 [AH11]</th>
<th>Area* (GE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>28035 (18 rounds unrolled, 130nm CMOS)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HIGHT-128 [HS+06]</th>
<th>Area* (GE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>42688 (32 rounds unrolled, 250nm CMOS)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mCrypton-128 [LK06]</th>
<th>Area* (GE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>37653 (13 rounds unrolled, 130nm CMOS)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25668 (31 rounds unrolled, 130nm CMOS)</td>
<td></td>
</tr>
</tbody>
</table>

* Area requirements extrapolated from round-based implementations.

8.4.2 Round-based Architecture

PRINCE is also synthesized as a round-based implementation to make a fair comparison with the existing works in the literature. Figure 8.2 shows the block diagram for the round-based implementation. To get a low-cost round-based implementation, we tried to maximize shared use of the operational blocks. This way, double use of the resources can be avoided. In our case, matrix multiplications in the linear layer gave a larger gate count than both the Sbox and inverse Sbox layers; therefore, by taking this layer in the middle of the round function (instead of putting Sboxes in the middle) and building the other blocks accordingly, we have achieved the smallest possible area.

A comparison of PRINCE round-based implementation with PRESENT-80, PRESENT-128, LED-128, AES-128, which are implemented in our technology with the same metrics that we use for PRINCE, is shown in Table 8.4.

We also provide a comparison of PRINCE with the reported results of other ciphers in the literature, such as CLEFIA, HIGHT, mCrypton, and Piccolo, which follows in Table 8.5. As in the unrolled case, all implementations are for encryption and decryption functionality with 128-bit key size.
Figure 8.2: Round-based implementation of PRINCE

Table 8.4: Performance comparison of round-based versions of PRINCE and other ciphers

<table>
<thead>
<tr>
<th>Technology</th>
<th>Area (GE)</th>
<th>Freq. (MHz)</th>
<th>Power (mW)</th>
<th>Tput (Gbps)</th>
<th>Area (GE)</th>
<th>Freq. (MHz)</th>
<th>Power (mW)</th>
<th>Tput (Gbps)</th>
<th>Area (GE)</th>
<th>Freq. (MHz)</th>
<th>Power (mW)</th>
<th>Tput (Gbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nangate 45 nm Generic</td>
<td>3779</td>
<td>666.7</td>
<td>5.7</td>
<td>3.56</td>
<td>3286</td>
<td>188.7</td>
<td>4.5</td>
<td>1.00</td>
<td>3491</td>
<td>153.8</td>
<td>5.8</td>
<td>0.82</td>
</tr>
<tr>
<td>UMC 90 nm Faraday</td>
<td>3105</td>
<td>833.3</td>
<td>1.2</td>
<td>1.67</td>
<td>2795</td>
<td>222.2</td>
<td>2.1</td>
<td>0.44</td>
<td>2909</td>
<td>196.1</td>
<td>2.5</td>
<td>0.39</td>
</tr>
<tr>
<td>UMC 130 nm Faraday</td>
<td>3707</td>
<td>833.3</td>
<td>1.6</td>
<td>1.67</td>
<td>3301</td>
<td>294.1</td>
<td>3.4</td>
<td>0.59</td>
<td>3458</td>
<td>196.1</td>
<td>2.9</td>
<td>0.39</td>
</tr>
<tr>
<td>PRESENT-80</td>
<td>3105</td>
<td>833.3</td>
<td>1.2</td>
<td>1.67</td>
<td>2795</td>
<td>222.2</td>
<td>2.1</td>
<td>0.44</td>
<td>2909</td>
<td>196.1</td>
<td>2.5</td>
<td>0.39</td>
</tr>
<tr>
<td>PRESENT-128</td>
<td>3309</td>
<td>312.5</td>
<td>0.5</td>
<td>0.41</td>
<td>3076</td>
<td>103.1</td>
<td>1.9</td>
<td>0.13</td>
<td>3407</td>
<td>78.13</td>
<td>2.4</td>
<td>0.10</td>
</tr>
<tr>
<td>AES-128</td>
<td>15880</td>
<td>250.0</td>
<td>5.8</td>
<td>2.91</td>
<td>14691</td>
<td>78.1</td>
<td>14.3</td>
<td>0.91</td>
<td>16212</td>
<td>61.3</td>
<td>18.8</td>
<td>0.71</td>
</tr>
<tr>
<td>LED-128</td>
<td>3309</td>
<td>312.5</td>
<td>0.5</td>
<td>0.41</td>
<td>3076</td>
<td>103.1</td>
<td>1.9</td>
<td>0.13</td>
<td>3407</td>
<td>78.13</td>
<td>2.4</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Table 8.5: Performance of round-based versions of other ciphers against PRINCE

<table>
<thead>
<tr>
<th>Technology</th>
<th>Technology (nm)</th>
<th>Area (GE)</th>
<th>Tput @100kHz (kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINCE</td>
<td>130</td>
<td>3491</td>
<td>533.3</td>
</tr>
<tr>
<td>CLEFIA-128 [AH11]</td>
<td>130</td>
<td>2678</td>
<td>73.0</td>
</tr>
<tr>
<td>HIGHT-128 [HSH+06]</td>
<td>130</td>
<td>2678</td>
<td>188.3</td>
</tr>
<tr>
<td>mCrypton-128 [LK06]</td>
<td>130</td>
<td>4108</td>
<td>492.3</td>
</tr>
<tr>
<td>Piccolo-128 [SIH+11]</td>
<td>130</td>
<td>1260</td>
<td>237.0</td>
</tr>
</tbody>
</table>

8.4.3 Performance of Both Architectures on FPGA

In order to observe the performance of the above-mentioned architectures (Sections 8.4.1 and 8.4.2) on reconfigurable devices, we also mapped them to an FPGA platform. Although PRINCE is targeted for ASIC platforms, the performance of PRINCE on FPGA could be of interest for researchers, e.g., for prototyping of different cryptographic applications,
8.5. Software Implementations

for utilization in reconfigurable applications, etc. The cores were synthesized, mapped, and place & routed on Xilinx Virtex-6 ML605 Evaluation Board (with XC6VLX240T FPGA on it) using Xilinx ISE Design Tool. Note that the presented figures are only the place & route numbers, and we do not provide any comparisons – these numbers are given just to provide some informative results. Our results are shown in Table 8.6 and Table 8.7 for unrolled and round-based implementations, respectively.

Table 8.6: Performance comparison of unrolled versions of PRINCE and other ciphers on FPGA

<table>
<thead>
<tr>
<th>Cipher</th>
<th># of slice LUTs (all used as logic)</th>
<th># of occupied slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINCE</td>
<td>1178 out of 150720</td>
<td>504 out of 37680</td>
</tr>
<tr>
<td>PRESENT-80</td>
<td>5662 out of 150720</td>
<td>2192 out of 37680</td>
</tr>
<tr>
<td>PRESENT-128</td>
<td>5611 out of 150720</td>
<td>2224 out of 37680</td>
</tr>
<tr>
<td>LED-128</td>
<td>15560 out of 150720</td>
<td>5034 out of 37680</td>
</tr>
<tr>
<td>AES-128</td>
<td>19929 out of 150720</td>
<td>6591 out of 37680</td>
</tr>
</tbody>
</table>

Table 8.7: Performance comparison of round-based versions of PRINCE and other ciphers on FPGA

<table>
<thead>
<tr>
<th>Cipher</th>
<th># of slice regs. (all used as FF)</th>
<th># of slice LUTs (all used as logic)</th>
<th># of occupied slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINCE</td>
<td>74 out of 301440</td>
<td>571 out of 150720</td>
<td>224 out of 37680</td>
</tr>
<tr>
<td>PRESENT-80</td>
<td>150 out of 301440</td>
<td>347 out of 150720</td>
<td>108 out of 37680</td>
</tr>
<tr>
<td>PRESENT-128</td>
<td>198 out of 301440</td>
<td>399 out of 150720</td>
<td>143 out of 37680</td>
</tr>
<tr>
<td>LED-128</td>
<td>79 out of 301440</td>
<td>506 out of 150720</td>
<td>176 out of 37680</td>
</tr>
<tr>
<td>AES-128</td>
<td>269 out of 301440</td>
<td>3031 out of 150720</td>
<td>1133 out of 37680</td>
</tr>
</tbody>
</table>

8.5 Software Implementations

In order to observe the software performance, PRINCE is also implemented both in C and 8-bit AVR assembly. The runtime, code size, and energy consumption of the C implementation is evaluated on the evaluation board EnergyMicro EF32GG-STK3700, which has a 32-bit ARM CORTEX-M3 microprocessor. As the compiler, we selected SimplicityStudio IDE, which is based on Eclipse and uses the GNU ARM C compiler. The results are given in Table 8.8. The performance of PRINCE on Atmel ATmega8A microcontroller is presented in Table 8.9. Note that the Sbox in AVR implementation is implemented in its ANF, which is used in bitsliced

Our C code was run on this microcontroller by Jan Zimmer.

93
implementations in order to achieve better software performance (see following chapters for bitslicing).

<table>
<thead>
<tr>
<th>Run time (clock cycles)</th>
<th>Code size (bytes)</th>
<th>Energy (µJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINCE 42881/43980</td>
<td>1447</td>
<td>102.4/102.8</td>
</tr>
</tbody>
</table>

Table 8.9: Performance of PRINCE on 8-bit Atmel ATmega128

<table>
<thead>
<tr>
<th>Run time (clock cycles)</th>
<th>Code size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINCE 3614</td>
<td>1108</td>
</tr>
</tbody>
</table>

8.6 Investigations on Countermeasures Against Potential SCA on PRINCE

Besides the security of the algorithm itself, susceptibility of its implementation to SCA is one of the main concerns when the algorithm is implemented in a hostile environment. In this section, we define the possible side-channel adversary model that we take into account while evaluating the vulnerability of an implementation of PRINCE and state the potential countermeasures together with a discussion of their feasibility with respect to the additional resource requirements on area and time.

8.6.1 Adversary Model

According to the definition given in [MM12]:

- An attack that combines \( v \) different time instances – usually in \( v \) different clock cycles – of each power trace is called \( v \)-variate attack, and

- the order of an attack – regardless of \( v \) – is defined by the order of the statistical moments that are considered in the attack.

Since an implementation of PRINCE is supposed to perform, e.g., an encryption in one clock cycle, the attack domain is restricted to univariate attacks only. On the other hand, if we suppose that a masking scheme is applied to avoid a dependency of the side-channel leakage with
processed (intermediate) values, all shares, e.g., the mask and masked data, must be processed at the same time. This poses another restriction to the attack domain of first-order attacks.

According to the scenarios, in which PRINCE is intended for use, we consider two cases where the adversary measures the corresponding side-channel leakage with either plaintext or ciphertext selected as input. Therefore, we define the adversary model as a side-channel attacker, who is able to perform a first-order univariate attack on a set of collected side-channel observations of either random or selected inputs/outputs of the cipher.

8.6.2 Potential Countermeasures for PRINCE Against SCA

Due to the operational conditions of PRINCE, the options to deploy a side-channel countermeasure are significantly restricted. For instance, shuffling is clearly not an option since all the operations are performed in one clock cycle. In the domain of masking, a possible solution can be threshold implementation [NRS11], which is based on – minimum 3-shared – secret sharing (Boolean masking) scheme and multi-party computation technique. If implemented correctly, it can provide security against first-order attacks. Below we discuss the possible options to realize a threshold implementation of PRINCE.

Choosing an Sbox for a Threshold Implementation

Using a threshold implementation to encrypt a block in a single clock cycle, we cannot directly apply the principle of decomposing an Sbox into several layers. We have to focus on threshold implementations in a single stage. The PRINCE Sbox corresponds to the class $C_{231}^3$ in [BNN+12], resulting in a cost of approximately 2000 GE due to 5 shares while avoiding decomposition. Thus, even the protection of only a single round at the beginning and at the end of the algorithm leads to a cost that is prohibitively high.

We therefore investigated the case of a 3-bit Sbox instead of a 4-bit Sbox. We consider $S : \mathbb{F}_3^2 \rightarrow \mathbb{F}_3^2$. A component function is a Boolean function $S_b : \mathbb{F}_3^2 \rightarrow \mathbb{F}_2$ defined as a linear combination (specified by $b$) of output bits of $S$, i.e., $S_b(x) = \langle b, S(x) \rangle$. Up to affine equivalence, there are only 4 classes of 3-bit Sboxes. The four classes can actually be classified by the number of linear component functions they have. One equivalence class can be classified by having 8 linear component functions, the second one has 4 linear component functions, the third has 2 linear component functions and the last has no linear component functions. The last class is clearly the only one that is cryptographically strong and therefore the only one of interest for us. This function corresponds in particular to the inverse function in $\mathbb{F}_2^{23}$. The main cryptographic criteria of this class are as follows.

1. The maximal probability of a differential is $1/4$.
2. The maximal absolute bias of a linear approximation is $1/4$.
3. Each of the 7 non-zero component functions has algebraic degree 2.

The function furthermore corresponds to the class $Q_3^3 [BNN+12]$. In order to reduce the hardware cost of a masked 3-bit Sbox to be used in an SCA-resistant implementation, we need to select one of the smallest Sboxes of this class. We therefore synthesized all Sboxes in the class, in a similar fashion to the syntheses presented in Section 8.3.1 for
4-bit Sboxes. The area of 3-bit Sboxes ranged from 6.5 to 11.75 \( \text{GE} \) – out of these, 36 Sboxes were with an area requirement of 6.5 \( \text{GE} \).

Note that even if we suppose that we can decompose a 3-bit Sbox into two stages, the cost of implementing a 3-shared version of a single Sbox results roughly in 100 \( \text{GE} \) (possibly higher for the 4-shared version without decomposition).

**Further Design Issues**

Using 3-bit Sboxes in special masking-rounds is non-trivial, since it complicates the security analysis significantly. Our considered adversary is able to choose the plaintexts, hence he can extend the attack to further rounds by partially fixing the plaintexts. Furthermore, in order to make it more difficult for the attacker to attack later rounds, we have to use a rather complex linear layer for high diffusion after two Sbox layers. We therefore considered a linear layer as a \( 64 \times 64 \) binary matrix with 9 ones per row and column (11 for its inverse), which ensures a maximal diffusion.

The operations of a **protected round** of PRINCE are listed in the following together with their estimated cost (estimations are for 90 nm low-leakage Faraday Library from UMC).

1. A key addition (150 \( \text{GE} \))
2. 21 3-bit protected Sboxes: 1-bit goes through the Sbox layer unmodified (2100 \( \text{GE} \))
3. A linear layer: A \( 64 \times 64 \) binary matrix with 9 ones per row and column – 11 ones per row and column for its inverse (1300 and 1600 \( \text{GE} \), respectively)
4. A second key addition (150 \( \text{GE} \))
5. Again the same Sbox layer (2100 \( \text{GE} \))

Note that the linear layer and key addition costs are tripled after sharing. Therefore, one protected-round of PRINCE results in at least 7 \( \text{kGE} \) which leads to a total cost of approximately 23 \( \text{kGE} \) for an unrolled implementation of the overall cipher – excluding the cost associated with the random number generation for the masks. Nevertheless, even this cost is prohibitively high compared to the original cipher.

**8.7 Concluding Remarks**

In this chapter, we described the new low-latency and low-cost cipher PRINCE, together with a variety of its implementations on different platforms, ranging from ASICs and FPGAs to software implementations. PRINCE results in a very small footprint, while achieving lowest latency compared to the existing lightweight solutions without compromising security. The compactness of PRINCE is the result of an extensive search on the layers of the cipher, especially on the Sbox. We also considered potential SCA-resistant constructions for PRINCE; however, such countermeasures increase the cipher area significantly. While PRINCE so far has been resistant to mathematical attacks and analysis, it is of course open to further analysis and comments.
Chapter 9

Towards Software-efficiency: NLU Instruction Set Extension

In this chapter, we take a step towards the software-efficiency of cryptographic algorithms. We address the open question of efficient cipher implementations on CPUs by introducing a non-linear/linear ISE, namely NLU, which is capable of implementing non-linear operations, namely Sboxes, expressed in their ANF and linear operations expressed as binary matrix multiplication, “multiply-and-add”, form. We propose a modular 8-bit architecture for NLU which allows its adoption to different CPU sizes for possible future extensions. However, in this work, we base our NLU design only on the widely-used 8-bit AVR instruction set. In order to demonstrate the coding advantage of the proposed ISE, we present 8-bit AVR implementations of standard cryptographic algorithms both “with NLU” and “without NLU”.

Contents of this Chapter

9.1 Introduction ................................................. 97
9.2 Related Work and the Proposed ISE Model ....................... 98
9.3 Unified Non-linear/Linear Hardware Unit .......................... 101
9.4 Applications ................................................ 105
9.5 Results and Discussion ...................................... 114
9.6 Conclusion and Future Directions .............................. 115

9.1 Introduction

Most of the progress in lightweight cryptography has been on the efficiency of the hardware implementations so far. However, lightweight designs with respect to software have barely been addressed. This is in contrast to the fact that many pervasive devices are equipped with embedded CPUs instead of dedicated ASICs and a large percentage of ciphers are implemented in software on these CPUs. Furthermore, even if a cipher is implemented in hardware on an end-device, e.g., on an RFID tag, it is not uncommon that a corresponding embedded software implementation is needed on the other end of the communication link, e.g., within an RFID reader. As most of such devices come without a support for cryptographic operations, and due to the lack of software-oriented cryptographic solutions, the realizations of these applications unfortunately result in costly implementations.
As a result of their hardware-efficient nature, the existing lightweight ciphers are generally based on hardware-friendly but software-unfriendly operations. For instance, bit-permutations with zero costs in hardware require several number of CPU cycles and/or lookup tables in software. Also, Sboxes in the substitution layer often require relatively large lookup tables in software, such as the T-tables in the case of AES [Koc09]. This development has somewhat led us to the paradox situation: Many of the “lightweight” and/or widely-used ciphers are not particularly lightweight with respect to their software implementations. In this work, we try to improve the cryptographic block cipher implementations on embedded CPUs by proposing an extension unit.

Our approach is based on a non-linear/linear operation unit, namely NLU, which is capable of computing common cipher operations substitution (non-linear) and permutation (linear). In NLU non-linear operations (Sboxes) are expressed in their ANF and linear operations are expressed as binary matrix multiplications (we name it as multiply-and-add). NLU targets embedded microcontrollers and is, hence, 8 bits wide. We designed NLU as an extension to the widely-used Atmel AVR instruction set. The modular architecture of NLU allows its adoption also to 16, 32, 64, and even 4-bit CPUs, giving the flexibility to be used with different microcontrollers and microprocessors in future.

For demonstration purposes, we show that NLU can result in time-area product reductions between 20-68% for the widely-used and standardized ciphers PRESENT, CLEFIA, Serpent, and AES. Note that Serpent and AES do not claim to be lightweight, but they are selected for different reasons. For instance, Serpent is a bitsliced design (see [ABK98]), which results in better software implementations. AES is the NIST standard and widely-used in many embedded applications. Interestingly, it also has better software performance in comparison to most of the existing lightweight ciphers. Therefore, for both ciphers, it is of interest to see the amount of cost reduction when implemented using NLU.

9.2 Related Work and the Proposed ISE Model

The performance enhancement of software implementations of cryptographic algorithms is not entirely a new idea. There have already been several studies on this subject. However, most of the previous works rely on implementation of cipher-specific instructions, which – in hardware – maps to plugging the specific cipher as a coprocessor into the main module [OE10, GGP08]. This approach, while providing the best performance improvement in terms of execution time, results in a considerable increase in microprocessor area. For example, in [OE10], the authors state that the area increase in the total area of the processor to be less than 65%. Furthermore, such an approach limits the performance improvement to a few specific ciphers. For instance, in addition to well-known Intel AES ISE [Sha12], there have been specific ISE proposals for AES targeting better performance on 32-bit processors [TsS05, Ts06] and 8-bit microcontrollers [TH08]. Moreover, [Ts05] suggests using ECC extensions for AES acceleration, which supports more cryptographic operations compared to others. However, these works are still limited to a few specific algorithms, such as AES and ECC and they do not support many different crypto algorithms at the same time. Other works try to address the software performance enhancement issue by introducing relatively complex instruction utilization [LSY01, ML01, SYL08, Rub07].

Note that 8-bit CPU still hold a huge share of the worldwide microprocessor market.
9.2. Related Work and the Proposed ISE Model

Note that many of these works, except [TH08], do not really target lightweight cryptography or resource-constrained devices.

The resource limitation on constrained devices is two-fold: In terms of hardware, it should occupy the smallest possible silicon area; in terms of software, it should be realized with minimum number of instructions. These two requirements usually contradict each other. The smallest possible silicon area is achieved by implementing all cryptographic operations in software on a small-footprint embedded processor; while the smallest possible code size can be achieved by dedicated hardware – either in the form of specific crypto-units, or crypto-coprocessors attached to an embedded processor. Unfortunately, smallest silicon area results in the maximum number of instructions and longest execution time, while the shortest code results in the highest silicon area.

In our case, we bridge the gap between these two approaches, which also comes with a compromise: We can neither reach the minimal silicon area nor the shortest code. Instead, we try to find a compromise point by introducing minimum additional silicon area that results in a code size reduction (especially time-area product reduction of about 20-68%, which can be considered as “high” for most applications). We also target a generic solution that would apply to (nearly) any cryptographic block cipher.

We take the standard ALUs, which exist even in the simplest microcontrollers, as our base approach. In fact, this is also the basis for Floating Point Unit (FPU)s, where floating point instructions are realized on specific hardware modules. These modules are physically-connected to the main data bus as the ALU and logically-connected to the instruction decoder, which forwards each floating point instruction to the FPU. The previous works on cryptographic instruction set extensions have also proposed additional “crypto” units that function exactly like the ALU or the FPU.

However, unlike the previous cipher-specific extensions, we propose a non-linear/linear operation unit. We design the NLU as a single input (source) and single output (destination) unit, where the additional operands (ANF or binary matrix coefficients) are stored inside a configuration register. The choice of the operations relies on investigation of several different lightweight and regular ciphers (both block and stream) [BKL+07, BCG+12a, SSA+07, NIS01, ABK98, HSH+06, CDK99, GNL11, LK06, GPPR11, SIH+11, HMM08, BD08, CP08] as well as recently-introduced lightweight hash functions that use block-cipher-like internal permutations [BDPA11, BKL+11, GPP11, AHMN13]. The two common operations in all of these ciphers are substitution (the non-linear operation to introduce confusion) and permutation (the linear operation to introduce diffusion). In the following, we explain the realization of these two operations.

9.2.1 Realization of Substitution

Substitution can be realized in software either by table-lookup or by its ANF expression. For lightweight ciphers, generally 4-bit Sboxes are chosen, unlike the 4-to-6 transformation in DES [NIS99] or the 8-to-8 transformation in AES. This led us to support only 4-bit substitutions in NLU along with the fact that 8-bit substitutions are more efficiently implemented as lookup tables.

In a lookup table implementation, a 4-bit substitution means a maximum of 16 different 4-bit output words (for a bijective substitution) selected by 4-bit input, resulting in a total storage.
of 16 × 4 = 64 bits. In an ANF-based implementation, each output bit is expressed in terms of the 1st to 4th degree product terms of input bits masked with 16 bits of ANF coefficients, which again results in a total storage of 16 × 4 = 64 bits. In other words, both approaches result in the same storage cost for 4-bit substitutions. However, at the end of our area cost analyses, we have observed that ANF gives the better area results; so, we opted for ANF in the non-linear part of NLU as it might also provide additional flexibility for the implementations of different ciphers (in turn, different Sboxes). Furthermore, it could even be used in the implementation of non-substitution type of operations, which might still require ANF form.

Another important parameter regarding the substitution operation is the input/output width. While we have chosen to support 4-bit substitutions only, the input operand width is enforced by the target microcontroller architecture. For most of the lightweight embedded applications, this starts from 4 bits (very low-end processors) and might go up to 32 bits (high-end embedded processors). Since the operations within the substitution layer of a cipher can be executed in parallel, it is possible to put one or more of the 4-bit Sboxes within the same substitution module, where nibbles of the input operand are distributed to each of these Sboxes, and their output nibbles are concatenated to form the output operand of 4 to 32 bits. In this case, only the number of the Sboxes inside the substitution module will change (one for 4-bit, eight for 32-bit). Apart from the configuration of the ANF coefficients, substitution requires an instruction with source and destination registers/memory addresses as operands.

We accomplish this functionality via the NNL (non-linear operation) instruction of NLU, which accepts source and destination as input and output arguments, respectively. In our case of 8-bit embedded processors, 8-bit content of the source is split into two nibbles, on which the non-linear operation described by the ANF coefficients is performed in parallel. The two 4-bit results are then combined to form the output for the destination.

### 9.2.2 Realization of Permutation

Permutation requires input/output widths larger than (almost always multiples of) Sbox widths. Therefore, in our case of 4-bit Sboxes, the permutation layer should be realized at least 8 bits wide. However, the upper limit for that is also enforced by the target microcontroller’s datapath width. For instance, with an operand width of 8 bits, it is infeasible to implement a 64-bit generic permutation block directly in one instruction. Hence, we decided to design our “permutation” hardware module as an 8 × 8 binary matrix multiplication, where we can use the 8-bit operand effectively for many different datapath widths. The choice of 8 × 8 is mainly due to the 64 bits storage required for the matrix coefficients, so that the storage required for the ANF coefficients can be reused.

In our scheme, it is possible to implement the “multiplication with a large matrix” by means of repetitive multiply-and-adds with the small 8 × 8 matrices. In other words, the permutation layer can be realized with a matrix multiplication unit and an accumulator. Since loading coefficients for each of these smaller matrices requires additional clock cycles and code space, it also makes sense to make reuse of each small matrix. This can be made possible by implementing a FIFO register of depth \(d\) instead of a single register \((d = 1)\) for the accumulator part of permutation unit.

FIFO register depth \(d\) is yet another parameter for the design. Our investigations on several different ciphers showed us that most permutation operations require inputs from a maximum
of 32 bits (8 Sboxes), which means that choosing \( d = 4 \) would be sufficient for most applications. It should be possible to accumulate on to any desired part of the FIFO register, which means choosing from one of the four bytes of the 32 bits. Furthermore, the permutation operation should also be run in multiply-only mode. In order to realize this, there are two options: We need either a single instruction with an input and output operand, 1-bit for the mode selection of (multiply-only or multiply-and-add), and 2 bits for the FIFO output selection in the multiply-and-add mode; or two separate instructions – one for multiply-only, and one for multiply-and-add.

In our design, we have opted for the latter case and implemented two new instructions – NMU (multiply-only) and NMA (multiply-and-add). Both instructions take the usual source and destination registers as input and output arguments, and perform binary matrix multiplication on the input byte. For NMU, this is the final result. In the case of NMA, the result of the multiplication is added onto the content of the \( s \)th register in the FIFO in order to obtain the final result, where \( s \) is the second input argument for this instruction. In both cases, the final result is both sent to the destination register and pushed into the FIFO.

9.2.3 Loading the Substitution and Permutation Coefficients

The configuration of the bits for the ANF or binary matrix coefficients is the last issue in our instruction set architecture. In parallel to our choice of 8-bit operands, we can perform this configuration by 8 bits at a time. It can be done either by writing 8 bits into a target portion of 64 bits selected via 3-bit address, or by shifting 8 bits into the 64-bit configuration register. The latter option is a better choice when supported with the selection of the number of bits to be shifted. In our case, the shifting changes from 1 to 8 bits, which is selected by 3 bits, resulting in an instruction with an 8-bit immediate value for the configuration and 3 bits of length selection.

This configuration is accomplished via the NLD (load NLU configuration) instruction. Its input arguments are the 8-bit input stream, \( K \), and the 3-bit value for the number of bits to be shifted, \( n \). NLD has no output arguments, as the NLU configuration register is the default target.

9.2.4 NLU Instructions

Details of the NLU instructions (supported by hardware choices) are given in Table 9.1. These instructions assume the 8-bit AVR microcontroller architecture with 16-bit instructions, which is very common for most of the embedded processors. However, they can also be modified for larger (or smaller) instruction widths. Note that \( Rs \) stands for the source register and \( Rd \) stands for the destination register. Other parameters are already mentioned in the previous sections.

9.3 Unified Non-linear/Linear Hardware Unit

In order to realize the explained NLU instructions, we implement a unified hardware unit performing non-linear and linear operations as explained in Section 9.2. In this unified structure, we have non-linear and linear operational sub-units, a 64-bit configuration register to store the ANF/matrix coefficients, and also an accumulation module for the add part of the linear
Table 9.1: NLU instructions

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Syntax</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLD</td>
<td>NLD n , K</td>
<td>Load NLU configuration</td>
<td>CONF ← CONF ≪ K[MSB–n], if n&gt;0, else CONF ← CONF ≪ K</td>
</tr>
<tr>
<td>NNL</td>
<td>NNL Rd, Rs</td>
<td>NLU non-linear operation</td>
<td>Rd(7:4) ← ANF[Rs(7:4)], Rd(3:0) ← ANF[Rs(3:0)]</td>
</tr>
<tr>
<td>NMU</td>
<td>NMU Rd, Rs</td>
<td>NLU multiply operation</td>
<td>Rd ← M × Rs, FIFO ← FIFO ≪ [M × Rs]</td>
</tr>
<tr>
<td>NMA</td>
<td>NMA s, Rd, Rs</td>
<td>NLU multiply-and-add operation</td>
<td>Rd ← M × Rs ⊕ FIFO(s), FIFO ← FIFO ≪ [M × Rs ⊕ FIFO(s)]</td>
</tr>
</tbody>
</table>

Figure 9.1: Overall structure of NLU

The 64-bit Register is connected to both the non-linear and linear units together with the 8-bit input data (operand). The output of the NLU is selected by the mode input – non-linear result or linear result. mode is set to 0 for NNL instruction, and to 1 for NMU and NMA instructions.

Figure 9.1: Overall structure of NLU

In our NLU proposal, the push input is activated whenever an NLD instruction is issued. The immediate value part of the instruction is pushed from right to left in n bits, which results in t clock cycles (where 1 ≤ t ≤ 8) to store a new 64-bit coefficient data. During the configuration phase, the register acts like a shift-register. sel selects the number of most signficant bits of the immediate value to be pushed. This is accomplished by an 8-to-1 barrel-shifter of 64 bits (not shown on the circuit schematic). After the configuration of the coefficient values, NLU can be operated in either non-linear or linear mode with respect to the issued instruction. The 64-bit coefficient data from the configuration register is connected to both the non-linear and linear units together with the 8-bit input data (operand). The output of the NLU is selected by the mode input – non-linear result or linear result. mode is set to 0 for NNL instruction, and to 1 for NMU and NMA instructions.
9.3. Unified Non-linear/Linear Hardware Unit

9.3.1 Non-linear Unit

The non-linear unit consists of many AND and XOR gates, which actually express the Sbox in ANF as mentioned previously. The input (masking) coefficient values define the ANF coefficients of the Sbox used in the cipher. In our case of 8-bit operand, we use two identical Sboxes in parallel, thereby sharing the same ANF masking coefficients. The most significant 4 bits of the input data go into the first Sbox and the least significant 4 bits to the second Sbox.

The outputs of the Sboxes are concatenated to form the 8-bit output in the same order as the input bits. For each Sbox output, the first 16 bits of the mask value define the first bit of the Sbox result, the second 16 bits define the second bit, and so on. A detailed schematic of the non-linear unit with all the masking coefficients \( m_{0...63} \) is shown in Figure 9.2.

9.3.2 Linear Unit

The linear operations unit partly performs bitwise operations expressed in multiply-and-add form. The linear unit handles the multiply-only step by multiplying the 8-bit input data with an \( 8 \times 8 \) binary matrix, whose coefficients are taken from the 64-bit configuration register. This matrix multiplication can be expressed as follows.

\[
\begin{align*}
\text{out}_7 &= (m_{63} \land \text{in}_7) \oplus \ldots \oplus (m_{56} \land \text{in}_0) \\
\text{out}_6 &= (m_{55} \land \text{in}_7) \oplus \ldots \oplus (m_{48} \land \text{in}_0) \\
&\vdots \\
\text{out}_1 &= (m_{15} \land \text{in}_7) \oplus \ldots \oplus (m_{8} \land \text{in}_0) \\
\text{out}_0 &= (m_{7} \land \text{in}_7) \oplus \ldots \oplus (m_{0} \land \text{in}_0)
\end{align*}
\]

Figure 9.3 depicts this multiplication scheme and Figure 9.4 presents the detailed circuit diagram of the linear matrix multiplication unit. An additional accumulation circuit performs the add step of multiply-and-add structure. When \( \text{acc} \) is set (i.e., NMA instruction is issued), the selected FIFO register output is added to the matrix multiplication result. \( sro \) input (corresponding to the \( n \) parameter of the NMA instruction) decides the register output to be taken. The output of each multiply-only or multiply-and-add operation is pushed into the FIFO via the mac input to allow the following accumulations, which is activated with each NMU or NMA instruction. The additional accumulation circuit can be seen in Figure 9.1 (on the right side of the figure, next to the linear unit).

9.3.3 Area and Power Consumption of NLU Module

In order to observe the impact of the NLU unit on area (which would be added to the overall microcontroller area cost), we implemented the unit as synthesizable RTL code and synthesized in UMC 90 nm low-leakage Faraday library. Our implementation occupies 1752 GE and the (simulational) power consumption of the unit is 28.59 \( \mu \)W @100 kHz (which is the case for smart cards). Therefore, our design is very compact and low-cost as an extension unit for Atmel 8-bit AVR microcontrollers.
Chapter 9. Towards Software-efficiency: NLU Instruction Set Extension

Figure 9.2: Non-linear unit
9.4 Applications

We demonstrate the effectiveness of the NLU in terms of code space and execution time by implementing two lightweight ciphers and two full size ciphers using the target instruction set, 8-bit AVR. As the lightweight ciphers, we chose PRESENT and CLEFIA, which are the lightweight ISO standards; and as the full size ciphers, we chose Serpent and AES. We implemented two different versions of each of these ciphers on Atmel’s 8-bit AVR microcontroller in assembly using both “standard instruction set” and “standard instruction set together with NLU instruction set extension” in order to have a fair comparison. The implementation details for these ciphers are given in the following.

9.4.1 PRESENT Software Implementation

The permutation layer of PRESENT is extremely simple for hardware implementations: It is only bit-shuffling, which has zero cost. However, the same arguments are not valid for software implementations. The bit-shuffling is a very costly operation when implemented with a standard instruction set. 4-bit Sbox also makes the direct software implementation harder in “non-4-bit” processors. Programmers usually prefer to use 8-bit $T$-tables, where the 16 entries of the Sbox are repeated 16 times at the low nibbles of 256 bytes, whereas the high nibble changes once in every 16 entries. This implementation makes it possible to run two parallel Sbox operations at a time, while costing 256 bytes instead of the ideal case of 8 bytes for Sbox storage.

In an optimized PRESENT implementation in AVR assembly, each 1-byte Sbox operation takes 6 instructions, one of which is a program memory load instruction with 3 cycles while others are all single cycle instructions. Therefore, total cost for the substitution layer (i.e., 64 bits) is 64 cycles and 48 instructions. There is also 256 bytes of $T$-table storage within the program memory. Permutation is much worse, where it takes 16 cycles to complete for each byte, resulting in a total of 128 cycles (plus overhead) per round for the permutation layer. Similar arguments are also valid for the key expansion, where the 80-bit key has to be rotated by 61 bits in every round. In total, this implementation occupies 660 bytes of program (Flash)
memory and takes 10792 cycles to complete the encryption of a single 64-bit input block with an 80-bit key.

In our implementation of PRESENT using the NLU ISE, it takes 8 instructions (and in turn, 8 clock cycles) to load the ANF coefficients into the NLU configuration register. Then it takes 1 cycle per byte for the Sbox operation, as shown in the following assembly code segment:

Figure 9.4: Linear unit – detailed circuit
9.4. Applications

; load ANF bits for the PRESENT Sbox
NLD 0, 0xb3
NLD 0, 0x92
NLD 0, 0x67
NLD 0, 0x0b
NLD 0, 0xde
NLD 0, 0x43
NLD 0, 0x4a
NLD 0, 0x80

; perform non-linear Sbox operation
NNL r18, r18
NNL r19, r19
NNL r20, r20
NNL r21, r21
NNL r22, r22
NNL r23, r23
NNL r24, r24
NNL r25, r25

We also use NLU for the permutation layer – this time in linear mode. We first write the expressions for the bit permutations as follows (where $Y_i$ is the output bit and $A_j$ is the input bit in big-endian format):

\[
\begin{align*}
Y_{63...56} &= A_{63,59,55,51,47,43,39,35} \\
Y_{55...48} &= A_{31,27,23,19,15,11,7,3} \\
Y_{47...40} &= A_{62,58,54,50,46,42,38,34} \\
Y_{39...32} &= A_{30,26,22,18,14,10,6,2} \\
Y_{31...24} &= A_{61,57,53,49,45,41,37,33} \\
Y_{23...16} &= A_{29,25,21,17,13,9,5,1} \\
Y_{15...8} &= A_{60,56,52,48,44,40,36,32} \\
Y_{7...0} &= A_{28,24,20,16,12,8,4,0}
\end{align*}
\]

In the next step, we convert these expressions into matrix form (shown only for the first output byte):

\[
\begin{bmatrix}
y_{63} \\
y_{62} \\
y_{61} \\
y_{60} \\
y_{59} \\
y_{58} \\
y_{57} \\
y_{56}
\end{bmatrix}
= \begin{bmatrix}
10000000 \\
00010000 \\
00001000 \\
00000100 \\
00000000 \\
00000000 \\
00000000 \\
00000000
\end{bmatrix} \oplus \cdots \oplus
\begin{bmatrix}
a_{63} \\
a_{62} \\
a_{61} \\
a_{60} \\
a_{59} \\
a_{58} \\
a_{57} \\
a_{56}
\end{bmatrix}
\begin{bmatrix}
00000000 \\
00000000 \\
00000000 \\
00000000 \\
00000000 \\
10000000 \\
00000000 \\
00000000
\end{bmatrix}
\]

Writing this for all the bytes, we get:
As can be seen in the matrix form, an output byte is obtained by the sum of multiplications of four input bytes with four different binary matrices, where each matrix is a 2-row shifted version of the neighboring matrix. In total, we need 16 different matrices, which can be obtained from each other in a sequence of row shifts. In the case of NLU, each byte written into the configuration register corresponds to left shift of the coefficients by one byte, which in fact is a 1-row shift upwards. Therefore, we start with multiplication of the rightmost byte and continue towards the leftmost byte by consecutive multiply-and-adds. However, since each byte pair uses the identical four matrices, we can also make use of this property by performing the rightmost multiplications of two bytes. We then continue with the multiply-and-adds in pairs, using the second register output from the accumulator FIFO. The resultant assembler code for the permutation layer is given as follows (only initialization and permutation for the first two bytes are shown):

\[
\begin{align*}
Y_7 &= M_{00}A_7 \oplus M_{01}A_6 \oplus M_{02}A_5 \oplus M_{03}A_4 \\
Y_6 &= M_{00}A_3 \oplus M_{01}A_2 \oplus M_{02}A_1 \oplus M_{03}A_0 \\
Y_5 &= M_{10}A_7 \oplus M_{11}A_6 \oplus M_{12}A_5 \oplus M_{13}A_4 \\
Y_4 &= M_{10}A_3 \oplus M_{11}A_2 \oplus M_{12}A_1 \oplus M_{13}A_0 \\
Y_3 &= M_{20}A_7 \oplus M_{21}A_6 \oplus M_{22}A_5 \oplus M_{23}A_4 \\
Y_2 &= M_{20}A_3 \oplus M_{21}A_2 \oplus M_{22}A_1 \oplus M_{23}A_0 \\
Y_1 &= M_{30}A_7 \oplus M_{31}A_6 \oplus M_{32}A_5 \oplus M_{33}A_4 \\
Y_0 &= M_{30}A_3 \oplus M_{31}A_2 \oplus M_{32}A_1 \oplus M_{33}A_0 
\end{align*}
\]
As can be seen in the code, it takes 16 cycles in average for the permutation of two bytes (or 8 cycles per byte), which is only the half of the implementation without NLU instructions. After applying a similar use of NLU for the key expansion block as well, we end up with 6017 clock cycles and 406 bytes of program memory to complete the encryption of a single 64-bit input block. The reduction in both code space and execution time is considerable with respect to a raw assembly implementation.

9.4.2 CLEFIA Software Implementation

CLEFIA is based on a 4-branch generalized Feistel structure, which uses two different diffusion functions and two different 8-bit Sboxes. On the key schedule part, the same diffusion functions are used to generate the intermediate key, $L$, from the input key, $K$. Both keys, $K$ and $L$ are then expanded to round keys using a permutation called DoubleSwap function. The fastest and most compact version of the cipher, the 128-bit key version, uses 18 rounds for both encryption and decryption.

In our comparison, we only implemented the fast and compact versions of the 128-bit key encryption function with minimal optimization in both standard assembly and NLU ISE cases. The 8-bit Sboxes are most efficiently implemented as lookup tables in either case. However, the situation is the opposite for the diffusion and permutation functions. Since both of them are based on linear combinations of shifted bytes, they are perfect examples to utilize the linear mode operation of the NLU.

For example, the basic operation of the diffusion function is a multiplication by 2 over finite field GF($2^8$) defined by the primitive polynomial $p(x) = x^8 + x^4 + x^3 + x^2 + 1$. It is repeated several times inside both diffusion operations. In the standard assembly, it is implemented as a subroutine, which takes 15 or 17 cycles (depending on the most significant bit of the input byte) including the subroutine calls. It is also possible to reduce the cycle count to 7 or 9 at the expense of repeating the corresponding program code several times, adding hundreds of bytes to the occupied program memory.
On the other hand, the whole multiplication is only a single NMU instruction for NLU following a one-time initialization of the internal product matrix (via NLD instruction). Similar approach is also used for the DoubleSwap function. The resultant NLU-based code is 1912 bytes vs. 2170 and 3046 bytes in the compact and fast standard codes, respectively. Execution time is reduced to 15268 cycles from 28684 cycles in the fast version and 42124 cycles in the subroutine-based compact version.

### 9.4.3 Serpent Software Implementation

Serpent is an interesting case study for several reasons: It is the cipher from which PRESENT has originated. Therefore, they have several similarities. However, Serpent is defined for bitsliced implementations. Therefore, its code can be optimized for different targets: Either for code space (compactness) or for execution time. In the compact case, Sboxes are implemented as lookup tables, which requires un-bitslicing and re-bitslicing the state bits before and after Sbox operations. Such an implementation is much slower than a bitsliced implementation.

Since NLU instruction set is optimized for ANF-based implementations, it is not directly applicable to a bitsliced cipher. However, in the case of Serpent, un-bitslicing operation is equivalent to the permutation layer of PRESENT, only with a larger block size – 128 bits instead of 64. Therefore, PRESENT permutation layer code can be reused together with the substitution layer codes. Finally, re-bitslicing has to be done, which is the inverse permutation of PRESENT and can be implemented using multiply-and-adds.

We have applied this strategy in re-writing the Serpent code, resulting in substantial gains both in code space and execution time. We have also re-written the bitsliced linear layer of Serpent using NLU linear operations, which provided nominal gain. As a result, we came up with a 10% faster software implementation, which has a less code size than the half of the fastest Serpent’s code size. It is also faster more than twice the most compact implementation, but slightly larger in code size.

### 9.4.4 AES Software Implementation

Since AES is the most widely-used and still unbroken block cipher, we have also investigated the possibility of implementing AES using NLU instructions. In this case, as in CLEFIA, we were not able to implement the Sboxes more efficiently than lookup tables. However, there is space for improving the implementation of MixColumns diffusion operator.

In an optimized implementation of the AES cipher, MixColumns operation takes 117 instructions, 16 of which are 3-cycle program memory load instructions, while all the others are single cycle instructions. Therefore, total execution time for the MixColumns diffusion operator is 149 cycles, whereas it requires a total of 492 bytes of program memory – 236 bytes for 118 instructions and 256 bytes for an auxiliary lookup table.

In our implementation using the NLU ISE, the same operation requires only 112 instructions (224 bytes of program memory), which also corresponds to 112 cycles of execution time. In order to implement MixColumns via NLU instructions, we use the linear mode of NLU.

We first recall the expressions for MixColumns:
9.4. Applications

\[ Y_0 = \{02\}A_0 \oplus \{03\}A_1 \oplus A_2 \oplus A_3 \]
\[ Y_1 = A_0 \oplus \{02\}A_1 \oplus \{03\}A_2 \oplus A_3 \]
\[ Y_2 = A_0 \oplus A_1 \oplus \{02\}A_2 \oplus \{03\}A_3 \]
\[ Y_3 = \{03\}A_0 \oplus A_1 \oplus \{03\}A_2 \oplus \{02\}A_3 \]

This can also be expressed in matrix form as follows (shown only for the first output \( Y_0 \)):

\[
\begin{bmatrix}
  y_7 \\
  y_6 \\
  y_5 \\
  y_4 \\
  y_3 \\
  y_2 \\
  y_1 \\
  y_0
\end{bmatrix} =
\begin{bmatrix}
  01000000 \\
  00100000 \\
  00010000 \\
  10010000 \\
  10001000 \\
  00001000 \\
  10000100 \\
  10000000
\end{bmatrix} \oplus
\begin{bmatrix}
  11000000 \\
  01100000 \\
  00110000 \\
  10011000 \\
  10001100 \\
  00000110 \\
  10000011 \\
  10000001
\end{bmatrix} \oplus
\begin{bmatrix}
  a_7 \\
  a_6 \\
  a_5 \\
  a_4 \\
  a_3 \\
  a_2 \\
  a_1 \\
  a_0
\end{bmatrix}
\]

The matrix form representation allows us to implement it using the linear mode of NLU. The most expensive part is the initializations of the matrices with coefficients for multiplication by 2 and 3. Re-initialization with two matrices (\( \times 2 \) and \( \times 3 \)) for every column corresponds to eight initializations, which is optimized by reusing the last initialized matrix. Only the initialization for the first column requires both matrices, while the other columns require only a single matrix initialization. The resultant assembly code is very similar to that of CLEFIA.

; temporary results are in r16 - r19

; first column: state is in r0 - r3 (32 bits)

; initialization for multiplication by 2
NLD 0, 0x40
NLD 0, 0x20
NLD 0, 0x10
NLD 0, 0x88
NLD 0, 0x84
NLD 0, 0x02
NLD 0, 0x81
NLD 0, 0x80

; multiplication by 2
NMU r16, r0
NMU r17, r1
NMU r18, r2
NMU r19, r3

; initialization for multiplication by 3
NLD 0, 0x00
NLD 0, 0x60
NLD 0, 0x30
NLD 0, 0x98
NLD 0, 0x8C
NLD 0, 0x06
NLD 0, 0x83
NLD 0, 0x81

; multiplication by 3 and accumulation
NMA 0, r16, r1
NMA 0, r17, r2
NMA 0, r18, r3
NMA 0, r19, r0

; multiplication by 1 and accumulation
EOR r16, r2
EOR r16, r3
EOR r17, r0
EOR r17, r3
EOR r18, r0
EOR r18, r1
EOR r19, r1
EOR r19, r2

; transfer temp results to state registers
MOVW r17:r16, r1:r0
MOVW r19:r18, r3:r2

; second column: state is in r4 - r7 (32 bits)

; multiplication by 3
...

; initialization for multiplication by 2
...

; multiplication by 2 and accumulation
...

; multiplication by 1 and accumulation
...

; transfer temp results to state registers
...
Our implementation using NLU instructions reduces execution time of the MixColumns operation by 24%, and its code size by 55%. In the overall AES encryption and decryption, this corresponds to a total reduction of 12% (2406 vs. 2739 cycles) and 10% (3246 vs. 3579 cycles) in execution time, respectively. The program memory size reduction for the combined encryption and decryption is 11% (1402 vs. 1570 bytes).
### Table 9.2: Performance comparison of ciphers implemented with/without NLU

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Number of Clock Cycles</th>
<th>Memory Utilization</th>
<th>Time-Area Product (TAP) (cycles-bytes)</th>
<th>TAP Gain (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESENT (LUT)</td>
<td>10792</td>
<td>660 bytes</td>
<td>$7.1 	imes 10^6$</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT (NLU)</td>
<td>6017</td>
<td>406 bytes</td>
<td>$2.4 	imes 10^6$</td>
<td>66</td>
</tr>
<tr>
<td>CLEFIA (compact)</td>
<td>42124</td>
<td>2170 bytes</td>
<td>$91.4 	imes 10^6$</td>
<td>0</td>
</tr>
<tr>
<td>CLEFIA (fast)</td>
<td>28684</td>
<td>3046 bytes</td>
<td>$87.4 	imes 10^6$</td>
<td>4</td>
</tr>
<tr>
<td>CLEFIA (NLU)</td>
<td>15268</td>
<td>1912 bytes</td>
<td>$29.2 	imes 10^6$</td>
<td>68</td>
</tr>
<tr>
<td>Serpent (ANF)</td>
<td>49314</td>
<td>7220 bytes</td>
<td>$356.0 	imes 10^6$</td>
<td>0</td>
</tr>
<tr>
<td>Serpent (LUT)</td>
<td>106338</td>
<td>2620 bytes</td>
<td>$278.6 	imes 10^6$</td>
<td>22</td>
</tr>
<tr>
<td>Serpent (NLU)</td>
<td>45431</td>
<td>2960 bytes</td>
<td>$134.5 	imes 10^6$</td>
<td>62</td>
</tr>
<tr>
<td>AES (LUT)</td>
<td>3159</td>
<td>1570 bytes</td>
<td>$4.96 	imes 10^6$</td>
<td>0</td>
</tr>
<tr>
<td>AES (NLU)</td>
<td>2826</td>
<td>1402 bytes</td>
<td>$3.96 	imes 10^6$</td>
<td>20</td>
</tr>
</tbody>
</table>

### 9.5 Results and Discussion

The performance that can be achieved via an NLU unit on an AVR microcontroller is shown in Table 9.2 with a comparison to the software implementations of PRESENT, CLEFIA, Serpent, and AES without the utilization of NLU.

PRESENT is first implemented in a LUT-based fashion, which is slower and larger than the NLU-based implementation. In our NLU-based approach, we have 44% less clock cycles and we use 39% less Flash memory. In the case of CLEFIA, we compare our results to a fast implementation and a compact implementation. Both use lookup for Sboxes and logic operations for diffusion layers. The main difference is the subroutines in the compact version, which are replaced by repeated code in the fast version. We were able to optimize both versions using the NLU in linear mode to implement diffusions and permutations. Our execution time reduction is between 46-64%, while the code space reduction is 37-12% against fast and compact versions, respectively.

For Serpent, we have two different implementations: One is using ANF representations for the substitution part and is implemented in bitsliced form, and the other is again implemented in bitsliced form except for the Sbox operation. In the latter, we have a LUT-based substitution layer as in the case of PRESENT. Our NLU-based Serpent design is even faster than the ANF-based Serpent implementation and uses approximately the same amount of memory with the LUT-based implementation. In order to compare our AES results, we use an optimized software implementation, which uses lookup tables for both Sboxes and MixColumns operation. We were only able to optimize the MixColumns operation, which alone reduced both the code size and execution time. Depending on the implementation (encryption/decryption or encryption-only), code size reduction is between 10-18%, while execution time reduction is around 10% for both cases.
9.6 Conclusion and Future Directions

We also list the time-area (execution time and program memory) products for each case in order to highlight the NLU ISE advantage. The execution times are given as average times. RAM utilization is independent of the implementation in all four ciphers.

9.6 Conclusion and Future Directions

In this work, we were able to improve the software implementations of cryptographic block ciphers on small CPUs. We came up with a non-linear/linear ISE NLU, which can implement non-linear and linear operations of these ciphers. Using the four new NLU instructions, we show that we achieve time-area product reductions of 20-68% for the widely-used ciphers PRESENT, Serpent, CLEFIA, and AES. The area overhead for the extension unit is only 1.8 kGE – almost the same as the area of a parallel PRESENT core designed in the same fashion.

In future, the extension unit design can be extended to different (16-bit, 32-bit, 64-bit, etc.) microcontroller architectures. Furthermore, more comparisons and cipher support can be added to the current and future ISE.
Chapter 10

Software-efficiency – Going Further: A Lightweight Block Cipher PRIDE

This chapter addresses the software performance bottleneck of block cipher implementations by proceeding on the cipher design side. In order to achieve an efficient cipher, we focus on the “software-costly” SPN core component “linear layer”. We first briefly present a general methodology to construct good, sometimes optimal, linear layers allowing for a large variety of trade-offs. Then, we search for efficient representatives of such linear layers regarding software performance. Our search is performed on a reconfigurable hardware platform (FPGA), which accelerates the process via a carefully-designed hardware search architecture. The value of the linear layer construction is underlined by using the resulting software-efficient linear layer in our new “software-oriented” block cipher proposal. PRIDE is optimized for 8-bit microcontrollers and significantly outperforms all academic solutions both in terms of code size and cycle count.

Contents of this Chapter

| 10.1 Introduction | 117 |
| 10.2 The Interleaving Construction | 119 |
| 10.3 The Search for “Software-friendly” Linear Layers | 121 |
| 10.4 The Software-oriented Lightweight Block Cipher PRIDE | 126 |
| 10.5 Conclusion | 133 |

10.1 Introduction

As already discussed in Chapter 8, the dominant metric according to which the vast majority of lightweight ciphers have been optimized is the chip area, which contradicts with the fact that we also need to consider other design metrics depending on the application. Software-efficiency is one of these important metrics (as mentioned in Chapter 9), as low-end embedded processors actually dominate the world of embedded systems and dedicated hardware is a comparably small fraction. Hence, most of the existing “hardware-friendly”-“software-unfriendly” cryptographic solutions are implemented on those devices in software. Unfortunately, programming in embedded systems is a skill game that heavily depends on the personal skills and experience of the programmer. Some specific applications, such as our focus lightweight cryptography,
especially require specialized knowledge and specially-tailored designs in order to get efficient software implementations. Considering this fact, it is quite puzzling that software-efficient solutions for lightweight crypto applications on low-cost processors have been (nearly) disregarded for so long. Certainly, there were a few exceptions: Besides the practical examples that we already presented in Chapter 9, several theoretical studies have been published in this field. There have been attempts to come up with ciphers that are (partially) tailored for low-cost processors \[ \text{SMMK12, SPG06, WZ11, GNL11, BSS}^{+}13, \text{KDH13}. \] Of these, execution times of both SEA and ITUbee are rather high, mostly due to the high number of rounds. Furthermore, ITUbee uses 8-bit Sboxes, which occupy a vast amount of program memory storage. However, SPECK, which is the lightweight cipher proposal of US National Security Agency (NSA) \(\text{together with SIMON} \) \[ \text{BSS}^{+}13 \], seems to be an excellent lightweight software cipher in terms of both speed and program memory.

It is obvious that there are quite some challenges to be overcome in this relatively untouched area of lightweight software cryptography. A software cipher for embedded devices should not only be compact in terms of program memory, but also be relatively fast in execution time. It should clearly be secure and, preferably, its security should be easily analyzed and verified. The latter can possibly be achieved by building on conservative structures, which are conventionally costly in software implementations, thereby posing even harder challenges. One major component influencing all or at least most of those criteria outlined above is the linear layer. Thus, it is important to have general constructions for linear layers that allow to explore and make optimal use of the possible trade-offs.

The general design approaches for linear layers are already given in Section 2.2.1. Out of these, the wide-trail strategy usually results in simple and strong security arguments. While the wide-trail strategy does provide a powerful tool for arguing about the security of a cipher, it does not help in actually designing an efficient linear layer (or the corresponding linear code) with a suitable number of active Sboxes. Here, with the exception of early designs in \[ \text{Dae95} \] and later PRINCE and mCrypton, most ciphers following the wide-trail strategy simply choose an \[ \text{MDS} \] matrix as the core component. This might guarantee an (partially) optimal number of active Sboxes, but usually comes at the price of a less efficient implementation. The only exception here is that, in the case of \[ \text{MDS} \] matrices, the authors of PHOTON and LED made the observation that implementing such matrices in a serialized fashion improves hardware-efficiency. This idea was further generalized in \[ \text{SDMS12, WWW12} \], and more recently in \[ \text{AF14} \].

It is our belief that, in many cases, it is advantageous to use a near-\[ \text{MDS} \] matrix (or in general a matrix with a sub-optimal branch number) for the overall design. Furthermore, it is, in our opinion, utmost surprising that there are virtually no general constructions or guidelines that would allow an \[ \text{SPN} \] design to benefit from security vs. efficiency trade-offs.

10.1.1 Our Contribution

As a part of this work, we take steps towards a better understanding of possible trade-offs for linear layers. In the following sections, we briefly present a general construction that allows to combine several strong linear mappings on a few number of bits into a strong linear layer for

\[ \text{Note that SIMON and SPECK was announced during the course of our work and they were not published in academic proceedings/journals. We therefore claim that our results are superior to the “academic” proposals.} \]
10.2 The Interleaving Construction

a larger number of bits\(^2\). From a coding theory perspective, this construction corresponds to a construction known as \textit{block-interleaving} (see [LC04], pages 131–132). While this idea is rather simple, its applicability is powerful. Implicitly, a specific instance of our construction is already implemented in [AES]. Furthermore, special instances of this construction are recently used in [BNN+10] and [GLSV14].

We show that this construction leads to very strong linear layers with respect to efficiency on embedded 8-bit microcontrollers. For this, we adopt a search strategy from [UCI+11] to find the most efficient linear layer possible within our constraints. We implemented this search as a carefully-designed hardware architecture on an FPGA platform in order to overcome the big computational effort involved and have the advantage of reconfigurability. Note that our target platform is Atmel’s AVR microcontroller, as it is dominating the market along with PIC; and many implementations in the literature are also implemented using AVR 8-bit instruction set. We therefore opted for this platform in order to provide a better comparison to other ciphers – especially to SIMON and SPECK.

Following the search, we use the resulting efficient linear layers in the design of a new \textit{software-oriented} block cipher named PRIDE which significantly outperforms all existing block cipher proposals of similar key sizes – with the exception of SIMON and SPECK. One of the key points here is that our construction of strong linear layers is nicely in line with a bitsliced implementation of the Sbox layer. Our cipher is comparable, both in speed and memory size, to the new NSA block ciphers SIMON and SPECK, dedicated for the same platform. We would like to note that while in this work we focus on \textit{SPN} ciphers, most of the results translate to the design of Feistel ciphers as well.

Note that this work is the output of a collaboration and was published at CRYPTO in 2014. In this chapter, we only focus on the parts that the author was involved in; therefore, we refer to the original publications [ADK+14a, ADK+14b] where necessary.

10.2 The Interleaving Construction

Following the wide-trail strategy described in Chapter 2, we build linear layers by constructing \((2^n, 2^n)\) additive codes with minimal distance \(d\) over \(\mathbb{F}_2^n\). The code needs to have a generator matrix \(G\) in standard form, i.e.,

\[
G = [I \mid L^T]
\]

where the submatrix \(L\) is invertible, and corresponds to the linear layer we are using.

Hence, the main question is how to construct “efficient” matrices \(L\) with a given branch number. Our construction allows to combine small matrices into bigger ones. We hereby drastically reduce the search-space of possible linear layers. This in turn makes it possible to construct efficient linear layers for various trade-offs.

For further information on interleaving construction and an illustrative example of our approach, we refer to our original publications [ADK+14a, ADK+14b].

10.2.1 The General Construction

In order to give a formal description of our approach, we first define the following isomorphism.

\(^2\)As this part is not the personal contribution of the thesis author, we refer to the original publications [ADK+14a, ADK+14b] for further details of this construction.
\[ P_{b_1, \ldots, b_k}^n : \left( \mathbb{F}_2^{b_1} \times \mathbb{F}_2^{b_2} \times \cdots \times \mathbb{F}_2^{b_k} \right)^n \rightarrow \left( \mathbb{F}_2^{b_1} \right)^n \times \left( \mathbb{F}_2^{b_2} \right)^n \times \cdots \times \left( \mathbb{F}_2^{b_k} \right)^n \]

\[ (x_1, \ldots, x_n) \mapsto \left( \left( x_1^{(1)}, \ldots, x_n^{(1)} \right), \ldots, \left( x_1^{(k)}, \ldots, x_n^{(k)} \right) \right) \]

where \( x_i = \left( x_1^{(1)}, \ldots, x_n^{(k)} \right) \) with \( x_j^{(j)} \in \mathbb{F}_2^{b_j} \).

This isomorphism performs the transformation of mapping Sbox outputs to our small linear layers \( L_i \).

Note that, for our purpose, there are in fact many possible choices for \( P \). In particular, we may permute the entries within \( \left( \mathbb{F}_2^{b_i} \right)^n \). Given this isomorphism we can now state our main theorem. The construction of \( P \) follows the idea of a diffusion-optimal mapping as defined in [DR01, Definition 5].

**Theorem 10.2.1** Let \( G_i = [I \mid L_i^T] \) be the generator matrix for an \( \mathbb{F}_2 \)-linear \((2n, 2^n)\) code with minimal distance \( d_i \) over \( \mathbb{F}_2^{b_i} \) for \( 0 \leq i < k \). Then the matrix \( G = [I \mid L^T] \) with

\[
L = \left( P_{b_1, \ldots, b_k}^n \right)^{-1} \circ (L_0 \times L_1 \times \cdots \times L_{k-1}) \circ P_{b_1, \ldots, b_k}^n
\]

is the generator matrix of an \( \mathbb{F}_2 \)-linear \((2n, 2^n)\) code with minimal distance \( d \) over \( \mathbb{F}_2^{b_i} \) where

\[
d = \min_i d_i \quad \text{and} \quad b = \sum_i b_i.
\]

For the proof, please see our publications [ADK+14a, ADK+14b].

A special case of the construction above is implicitly already used in [AES]. In the case of [AES] it is used to construct a \([8, 4, 5]\) code over \( \mathbb{F}_2^2 \) from 4 copies of the \([8, 4, 5]\) code over \( \mathbb{F}_2^3 \) given by the MixColumns operation. In the Superbox view on [AES] the ShiftRows operation plays the role of the mapping \( P \) (and its inverse) and MixColumns corresponds to the mappings \( L_i \).

In the following, we use this construction to design efficient linear layers. Besides the differential and linear branch number, we hereby focus mainly on three criteria:

1. Maximize the diffusion (via ensuring high dependency)
2. Minimize the density of the matrix
3. Software-efficiency

In this thesis, we will explain only the third bullet-point, software-efficiency, in detail and refer to the original publications [ADK+14a, ADK+14b] for the details of the first two criteria.

The strategy we employ is as follows. We first find candidates for \( L_0 \), i.e., \((2n, 2^n)\) additive codes with minimal distance \( d_0 \) over \( \mathbb{F}_2^{b_0} \). In this stage, we ensure that the branch number is \( d_0 \) and our efficiency constraints are satisfied. We then apply permutations to \( L_0 \) to produce \( L_i \) for \( i > 0 \). This stage maximizes diffusion.
10.3. The Search for “Software-friendly” Linear Layers

10.2.2 Searching for $L_0$

The following lemma (which is a rather straightforward generalization of Theorem 4 in [WWW12]) gives a necessary and sufficient condition that a given matrix $L$ has branch number $d$ over $\mathbb{F}_2$.

**Lemma 10.2.2** Let $L$ be a $b \times b$ binary matrix, decomposed into $b \times b$ submatrices $L_{i,j}$.

$$L = \begin{pmatrix} L_{0,0} & L_{0,1} & \cdots & L_{0,n-1} \\ L_{1,0} & L_{1,1} & \cdots & L_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ L_{n-1,0} & L_{n-1,1} & \cdots & L_{n-1,n-1} \end{pmatrix}$$  \hspace{1cm} (10.1)

Then, $L$ has differential branch number $d$ over $\mathbb{F}_2$ if and only if all $i \times (n - d + i + 1)$ block submatrices of $L$ have full rank for $1 \leq i < d - 1$. Moreover, $L$ has linear branch number $d$ if and only if all $(n - d + i + 1) \times i$ block submatrices of $L$ have full rank for $1 \leq i < d - 1$.

Based on Lemma [10.2.2] we may instantiate various search algorithms, which we will describe in the following sections. In our search we focus on cyclic matrices, i.e. matrices where row $i > 0$ is constructed by cyclic shifting row 0 by $i$ indices. These matrices have the advantage of being efficient both in software and hardware. Furthermore, since these matrices are symmetric, considering the dual code $C^\perp$ to $C = [I \mid L^T]$ is straightforward.

10.3 The Search for “Software-friendly” Linear Layers

In this section, we give the details of our hardware-based search in order to find efficient linear layers for our “software-oriented” cipher. We explain our search constraints and present our hardware architecture design in detail.

We are, unsurprisingly, making use of the construction given in Theorem [10.2.1]. We decided on a linear layer with high dependency and a linear and differential branch number of 4. One key-observation is that the construction of Theorem [10.2.1] fits naturally with a bitsliced implementation of the cipher, in particular with the Sbox layer. As a bitsliced implementation of the Sbox layer is advantageous on 8-bit microcontrollers, in any case this is a nice match.

A natural choice in terms of Theorem [10.2.1] is to choose $k = 4$ and $b_1 = b_2 = b_3 = b_4 = 1$. Thus, the task reduces to finding four $16 \times 16$ matrices forming one $64 \times 64$ matrix (to permute the whole state) of the following form.

$$\begin{pmatrix} L_0 & 0 & 0 & 0 \\ 0 & L_1 & 0 & 0 \\ 0 & 0 & L_2 & 0 \\ 0 & 0 & 0 & L_3 \end{pmatrix}$$

Each of these four $16 \times 16$ matrices should provide branch number 4 and together achieve high dependency with the least possible number of instructions. Instead of searching for an efficient implementation for a given matrix, we decided to search for the most efficient solution fulfilling our criteria. However, we need to simplify this search process in order to make it realizable.
In a software implementation, the part of the state, which is stored in two 8-bit registers $X$ and $Y$, is multiplied with each $L_i$ as a part of the linear layer application. Hence, we can split each $16 \times 16$ matrix as in the following in order to come up with a structured search.

$$
\begin{pmatrix}
X' \\
Y'
\end{pmatrix} =
\begin{pmatrix}
A & B \\
C & D
\end{pmatrix}
\begin{pmatrix}
X \\
Y
\end{pmatrix}
$$

where

$$L_i =
\begin{pmatrix}
A & B \\
C & D
\end{pmatrix}.$$

Note that $X, Y$ are 8-bit registers and $A, B, C, D$ are all $8 \times 8$ cyclic matrices. In the following, we explain our search effort to find instructions resulting in “good” $8 \times 8$ binary matrices, i.e., $A, B, C, D$, on a hardware-based search architecture.

### 10.3.1 Proposed Search Model

Our proposed search architecture is highly-parallel and operates on a limited set of instructions pertinent to implementation of linear layer diffusion matrices. A predetermined number of instructions is executed in a pipelined manner and the resultant output register contents are checked either for match to a target matrix or for certain cryptographic properties. As already mentioned, we are looking for $8 \times 8$ binary matrices in this search; however, we will combine the resulting matrices in the end in order to come up with $16 \times 16$ diffusion matrices.

Similar previous attempts targeted to reduce the number of instructions for substitution layer [UCI+11]. In their work, the authors wanted to come up with the least number of instructions (in turn clock cycles) for 4-bit Sboxes. However, their search was run on a software platform, which significantly slowed down the search process. In our work, we focus on the linear layer and try to find diffusion matrices giving the least possible number of instructions. To this end, we make use of a hardware platform (Xilinx Virtex-6 ML605 Evaluation Board), which inherently possesses both parallelism and pipelining properties that significantly accelerate the search process.

First of all, in order to have an idea of the number of instructions that we need, we started our search effort with analyzing the best figures in the literature, i.e., cycle counts of NSA lightweight ciphers SIMON and SPECK. A naive back-the-envelope calculation of a possible software implementation revealed that we have to come up with a linear layer of 7-9 instructions/byte for a 64-bit block cipher. Our manual attempts provided us with $BN = 4$ diffusion matrices of 9-11 instructions/byte. Clearly, in order to reach our targets, these figures had to be reduced.

A natural idea is to search through all instruction combinations (for a limited set and number of instructions) and see if we end up with good diffusion matrices. In other words, the problem can be summarized as follows: Given $N$ instructions$^3$ (say $N = 8$) and only linear operations (exclusive OR, rotate, shift, move, and masking) with 2-3 temporary registers, which instruction

---

$^3$More importantly, clock cycles. It can actually differ from the number of instructions, but each instruction of our limited instruction set costs 1 clock cycle. So, the number of instructions are the same with the clock cycle count in our case.
combinations would result in good matrices? And more importantly, how can we go through all \( N \)-instruction combinations?

In our search model, the starting (initial) value is the identity matrix, i.e., \( Y = A \). We execute instructions step by step, similar to a software implementation. Each “linear” instruction will act as a matrix multiplied with the initial matrix, i.e., \( Y = M \times A \), which will finally yield a certain value in the output register. After all instructions are executed, the quality of this \( M \) will be checked. Here, by quality, we mean the branch number and non-singularity – the cryptographic properties – of the matrix. This scheme is depicted in Figure 10.1.

Figure 10.1: Instruction flow with quality check – an example with 8 instructions

As mentioned before, our search is limited only to instructions that can result in a “linear” operation, which are listed in Table 10.1.

In the overall structure, there should be a counter that would go through all possible combinations for each one of the targeted \( N \) instructions. Depending on these counter value(s), we have to decide on the instruction’s equivalent matrix, the source and destination registers; and then apply them.

An important question is the number of source/destination registers. Clearly, one register is needed as the IO, and 2-3 registers are needed for temporary storage. Although manual coding attempts revealed that 1-2 registers are sufficient, we use 3 temporary registers in our design.

The next question is if we need all 4 registers at every stage. The answer is actually no, as we can skip some of the registers depending on the instruction order. This scheme is depicted in Figure 10.2. Such a reduction of course means a drop in the time complexity, as we try less number of possibilities.

This scheme means, we should define five different instruction module types, one for each of 1\textsuperscript{st}, 2\textsuperscript{nd}, 3\textsuperscript{rd}, regular (intermediate), and last instructions. Furthermore, not every instruction is possible at every cycle. For example, the first instruction cannot be EOR, since a second

Figure 10.2: Overall instruction flow – showing source and destination skips at each stage
Table 10.1: Limited instruction set

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Operation</th>
<th>Description</th>
<th>Input → Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOR</td>
<td>$D \leftarrow D \oplus S$</td>
<td>Multiply destination matrix with identity matrix ($I$) and then add (EOR) source matrix.</td>
<td>-</td>
</tr>
<tr>
<td>MOV</td>
<td>$D \leftarrow S$</td>
<td>Copy source matrix to destination.</td>
<td>-</td>
</tr>
<tr>
<td>MOVW</td>
<td>$D + 1 : D \leftarrow S + 1 : S$</td>
<td>Copy two source matrices to two destination matrices.</td>
<td>-</td>
</tr>
<tr>
<td>SWAP</td>
<td>$D \leftarrow D_{3..0}, D_{7..4}$</td>
<td>Swap upper and lower nibbles of destination.</td>
<td>$x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0 \rightarrow$ $x_3 x_2 x_1 x_0 x_7 x_6 x_5 x_4$</td>
</tr>
<tr>
<td>ROL</td>
<td>$D \leftarrow D \ll 1$</td>
<td>Rotate destination by 1-bit to left using carry register.</td>
<td>$x_7 x_6 x_5 x_4 x_3 x_2 y_1 x_0 c \rightarrow$ $x_0 x_3 x_4 x_5 x_6 x_7 y_1 x_0 c$</td>
</tr>
<tr>
<td>ROR</td>
<td>$D \leftarrow D \gg 1$</td>
<td>Similar to ROL, in right direction instead.</td>
<td>$x_7 x_6 x_5 x_4 y_3 x_2 x_1 x_0 c \rightarrow$ $x_0 x_3 x_4 x_5 x_6 x_7 x_8 x_9 c$</td>
</tr>
<tr>
<td>LSL</td>
<td>$D \leftarrow D \ll 1$</td>
<td>Logical shift left, 0 is pushed in instead of carry.</td>
<td>$x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0 c \rightarrow$ $x_0 x_3 x_4 x_5 x_6 x_7 y_1 x_0 c$</td>
</tr>
<tr>
<td>LSR</td>
<td>$D \leftarrow D \gg 1$</td>
<td>Logical shift right, LSL in right direction.</td>
<td>$x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0 c \rightarrow$ $x_0 x_3 x_4 x_5 x_6 x_7 x_8 x_9 c$</td>
</tr>
<tr>
<td>ASR</td>
<td>$D \leftarrow D \div 2$</td>
<td>Arithmetic shift right, divide destination by 2.</td>
<td>$x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0 c \rightarrow$ $x_0 x_3 x_4 x_5 x_6 x_7 x_8 x_9 c$</td>
</tr>
<tr>
<td>CLC</td>
<td></td>
<td>Clear carry.</td>
<td>$x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0 c \rightarrow$ $x_0 x_3 x_4 x_5 x_6 x_7 x_8 x_9 c$</td>
</tr>
<tr>
<td>CLR</td>
<td></td>
<td>Clear register.</td>
<td>$x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0 c \rightarrow$ $0000000000c$</td>
</tr>
</tbody>
</table>

Operand does not exist yet. Also, there should not be any bit destructing last instruction, like LSL, LSR, ASR. All of these criteria further reduce the time complexity of our search model.

10.3.2 Overall Architecture

Our resultant reconfigurable architecture is shown in Figure [10.3]. We have implemented different types of blocks to check the properties of the generated linear layer matrices, such as non-singularity (validity) and linear/differential branch numbers (cryptographic properties) of the matrix.

Non-singularity is checked via the determinant. For binary matrices, a non-zero determinant guarantees non-singularity. Finding the determinant is usually a problem for matrices over $4 \times 4$. However, since we are interested in the determinant of a binary matrix, we can simply use $U$ of $LU$-decomposition. For that, we turn the matrix into an upper triangle. If it can be done, then the matrix is simply non-singular. This operation is realized via line-by-line elimination in a total of 8 steps. On the other hand, the branch number can be found by exact realization of “minimum of the sum of input and output Hamming distances” algorithm in hardware, implemented as a $256 – to – 1$ binary tree.
10.3.3 Evaluation of Search Results

We performed our search using the Xilinx ML605 Evaluation Board, which has a Virtex-6 XC6VLX240T FPGA. We tried different combinations of instructions from 7 to 11 instructions. The resulting matrices are checked for the mentioned properties.

Using our parallel architecture, we were able to find several good permutation layer matrices starting from 7 instructions with branch number 4. We were able to search up to 11 instructions and cover matrices with branch number 6 as well.
For further design, time complexity reduction, and reconfigurable hardware architecture details, we refer to our publication [KLY13], as the topic is too broad to cover within the scope of this chapter.

Note that, following the search results, we combined the $8 \times 8$ matrices manually to form $16 \times 16$ matrices. During this effort, we made further optimizations. For example, we used MOVW instead of MOV where necessary, in order to come up with a better number of instructions/cycle count. Such a replacement resulted in one more temporary register; however, this was not a problem as we had enough registers left.

### 10.4 The Software-oriented Lightweight Block Cipher PRIDE

In this section, we describe our new lightweight software-cipher PRIDE, a 64-bit block cipher that uses a 128-bit key. It has an efficient linear layer and an involution Sbox as well as a low-cost key schedule. Note that the key schedule design and security analysis of the cipher are realized in cooperation with the co-authors of the original publications [ADK+14a, ADK+14b]. The author of this thesis, as mentioned before, has contributed to the efficient linear layer search, Sbox selection, and implementations of the cipher.

#### 10.4.1 The Extremely Efficient Linear Layer

As a result of the search explained in previous sections, we achieved an extremely efficient linear layer. The cheapest solution provided by our search needed 36 cycles for the complete linear layer, which is what we opted for. The optimal matrices forming the linear layer are given in the appendix (Part V). Of these four matrices, $L_0$ and $L_3$ are involutions with the cost of 7 instructions (in turn, clock cycles), while $L_1$ and $L_2$ require 11 and 13 instructions for true and inverse matrices, respectively. The assembly codes are given in the following.

```assembly
; State s0, s1, s2, s3, s4, s5, s6, s7
; Temporary registers t0, t1, t2, t3

; Linear Layer and Inverse Linear Layer: L0
movw t0, s0 ; t1:t0 = s1:s0
swap s0
swap s1
eor s0, s1
eor t0, s0
mov s1, t0
eor s0, t1

; Linear Layer: L1
swap s3
movw t0, s2 ; t1:t0 = s3:s2
movw t2, s2 ; t3:t2 = s3:s2
```

126
10.4. The Software-oriented Lightweight Block Cipher PRIDE

lsl t0
rol t2
lsr t1
ror t3
eor s2, t3
mov t0, s2
eor s2, t2
eor s3, t0

; Inverse Linear Layer: L1
movw t0, s2 ; t1:t0 = s3:s2
movw t2, s2 ; t3:t2 = s3:s2
lsr t0s
ror t2
lsr t1
ror t3
eor t3, t2
eor s3, t3
swap s3
mov s2, t3
lsr t3
ror s2
eor s2, t2

; Linear Layer: L2
swap s4
movw t0, s4 ; t1:t0 = s5:s4
movw t2, s4 ; t3:t2 = s5:s4
lsl t0
rol t2
lsr t1
ror t3
eor s4, t3
mov t0, s4
eor s4, t2
eor s5, t0

; Inverse Linear Layer: L2
movw t0, s4 ; t1:t0 = s5:s4
movw t2, s4 ; t3:t2 = s5:s4
lsr t0
ror t2
lsr t1
ror t3
eor t3, t2
Chapter 10. Software-efficiency – Going Further: A Lightweight Block Cipher PRIDE

```assembly
eor s5, t3
mov s4, t3
lsr t3
ror s4
eor s4, t2
swap s4

; Linear Layer and Inverse Linear Layer: L3
movw t0, s6 ; t1:t0 = s7:s6
swap s6
swap s7
eor s6, s7
eor t1, s6
mov s7, t1
eor s6, t0
```

Comparing to linear layers of other \textit{SPN} based ciphers clearly demonstrated the benefit of our approach. However, note that these comparisons have to be taken with care as not all linear layers operate on the same state size and do not offer the same security level. The linear layer of PRESENT costs 144 cycles (derived from the total cycle count given in \cite{EKM13}). MixColumns operation of \texttt{AES} costs 117 instructions (but 149 cycles because of 3-cycle data load instruction utilizations, as MixColumns constants are implemented as lookup table – which means additional 256 bytes of memory, too) \cite{AVR}. Note that ShiftRows operation was merged with the lookup table of Sbox in this implementation, so we take only MixColumns cost as the linear layer cost. The linear layer of CLEFIA (again 128-bit cipher) costs 146 instructions and 668 cycles. The bitslicing-oriented design Serpent (128-bit cipher) linear layer costs 155 instructions and 158 cycles. Other lightweight proposals, KLEIN and mCrypton linear layers cost 104 instructions (100 cycles) and 116 instructions (342 cycles), respectively \cite{EGG12}. Finally, the linear layer cost of PRINCE is 357 instructions and 524 cycles\footnote{It is of course not fair to compare a 128-bit cipher with a 64-bit cipher. However, we provide \texttt{AES} numbers as a reference due to the fact that it is a widely-used standard cipher and its cost is much better compared to many lightweight ciphers.}, which is even worse than \texttt{AES}. One of the reasons for this high cost is the non-cyclic $4 \times 4$ matrices forming the linear layer. The other reason is the ShiftRows operation applied on 4-bit state words, which makes coding much more complex than that of \texttt{AES} on an 8-bit microcontroller.

\subsection*{10.4.2 Sbox Selection}

For our bitsliced design, we decided to use a very simple – in terms of software-efficiency – 10-instruction Sbox (which makes $10 \times 2 = 20$ instructions in total for the whole state). The formulation for this Sbox is given below.

\footnote{Result taken from Chapter \ref{chapter:software-efficiency}}
\[
A = c \oplus (a \& b) \\
B = d \oplus (b \& c) \\
C = a \oplus (A \& B) \\
D = b \oplus (B \& C)
\]

It is at the same time an involution Sbox, which prevents the encryption/decryption overhead. Besides being very efficient in terms of cycle count, this Sbox is also optimal with respect to linear and differential attacks. The maximal probability of a differential is \(1/4\) and the best correlation of any linear approximation is \(1/2\). The PRIDE Sbox is depicted in the following.

<table>
<thead>
<tr>
<th>(x)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S[x])</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>F</td>
<td>1</td>
<td>5</td>
<td>E</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>D</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

The corresponding assembly code for the Sbox is as follows.

```assembly
; State s0, s1, s2, s3, s4, s5, s6, s7
; Temporary registers t0, t1, t2, t3

; Substitution Layer
movw t0, s0
movw t2, s2
and s0, s2
eor s0, s4
and s2, s4
eor s2, s6
and s1, s3
eor s1, s5
and s3, s5
eor s3, s7
movw s4, s0
movw s6, s2
and s4, s6
eor s4, t0
and s6, s4
eor s6, t2
and s5, s7
eor s5, t1
and s7, s5
eor s7, t3
```
10.4.3 Description of PRIDE

Similar to PRINCE, the cipher makes use of the FX-construction [Bir05]. A pre-whitening key $k_0$ and post-whitening key $k_2$ are derived from one half of $k$, while the second half serves as basis $k_1$ for the round keys, i.e.,

$$k = k_0||k_1 \quad \text{with} \quad k_2 = k_0.$$ 

Moreover, in order to allow an efficient bitsliced implementation, the cipher starts and ends with a bit-permutation. This clearly does not influence the security of PRIDE in any way. Note that in a bitsliced implementation, none of the permutations $P$ nor $P^{-1}$ used in PRIDE has to be actually implemented explicitly.

The cipher has 20 rounds, of which the first 19 are identical. Subkeys are different for each round, i.e., the subkey for round $i$ is given by $f_i(k_1)$. We define

$$f_i(k_1) = k_{10}||g_i^{(0)}(k_{11})||k_{12}||g_i^{(1)}(k_{13})||k_{14}||g_i^{(2)}(k_{15})||k_{16}||g_i^{(3)}(k_{17})$$

as the subkey derivation function with four byte-local modifiers of the key as

$$g_i^{(0)}(x) = (x + 193i) \mod 256, \quad g_i^{(1)}(x) = (x + 165i) \mod 256, \quad g_i^{(2)}(x) = (x + 81i) \mod 256, \quad g_i^{(3)}(x) = (x + 197i) \mod 256,$$

which simply add one of four constants to every other byte of $k_1$. The overall structure of the cipher is depicted in the following.

```
\text{The round function } R \text{ of the cipher shows a classical SPN. The state is XORed with the round key, fed into 16 parallel 4-bit Sboxes and then permuted and processed by the linear layer.}
```
The difference between $R$ and $R'$ is that in the latter no more diffusion is necessary, therefore the last round ends after the substitution layer. With the software-friendly matrices we have found, the linear layer is defined as follows (cf. Theorem 10.2.1):

$$L := P^{-1} \circ (L_0 \times L_1 \times L_2 \times L_3) \circ P \quad \text{where} \quad P := P_{1,1,1,1}^{16}.$$

The test vectors for the cipher are provided in the appendix (Part V).

### 10.4.4 Performance Analysis

As already mentioned, one round of our proposed cipher PRIDE consists of a linear layer, a substitution layer, a key addition, and a round constant addition (key update). In a software implementation of PRIDE on a microcontroller, we also perform branching in each round of the cipher in addition to the previously listed layers. Adding up all these costs gives us the total implementation cost for one round of the cipher. The total cost can roughly be calculated by multiplying the number of rounds with the cost of each round. Note that we should subtract the cost of one linear layer from the overall cost, as PRIDE has no linear layer in the last round.

The software implementation cost of the round function of PRIDE on Atmel AVR ATmega8A 8-bit microcontroller is presented in the following.

<table>
<thead>
<tr>
<th></th>
<th>Key update</th>
<th>Key addition</th>
<th>Sbox Layer</th>
<th>Linear Layer</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (cycles)</td>
<td>4</td>
<td>8</td>
<td>20</td>
<td>36</td>
<td>68</td>
</tr>
<tr>
<td>Size (bytes)</td>
<td>8</td>
<td>16</td>
<td>40</td>
<td>72</td>
<td>136</td>
</tr>
</tbody>
</table>

Comparing PRIDE to existing ciphers in literature, we can see that it outperforms many of them significantly both in terms of cycle count and code size. Note that we are not using any lookup tables in our implementation, in turn no RAM. The comparison with existing implementations is given in Table 10.2.

In the table, the first row is the time (performance) in clock cycles, the second row is the code size in bytes, and the third row is the equivalent rounds. The third row expresses the number of rounds for the given ciphers that would result in a total running time similar to PRIDE.

Note that we do not list the RAM utilization for the ciphers under comparison in the table. In the implementation of PRIDE, our target was to be fast and at the same time compact. Note that we do not exclude data & key read and data write back as well as the whitening steps in our results (these are omitted in SIMON and SPECK numbers). Although the given numbers are just for encryption, decryption overhead is also acceptable: It costs 1570 clock cycles and 282 bytes.

A cautionary note is indicated for the above comparison for several reasons. AES, Serpent, CLEFIA, and NOEKOEN are working on 128-bit blocks; so, for a cycle per byte comparison, their cycle count has to be divided by a factor of two. Moreover, the ciphers differ in the claimed

---

6 Which has the additional advantage of increased resistance against cache-timing attacks.
security level and key-size. PRIDE does not claim any resistance against related-key attacks (and actually can be distinguished trivially in this setting) and also generic time-memory trade-offs are possible against PRIDE in contrast to most other ciphers. Besides those restrictions, the security margin in PRIDE in terms of the number of rounds is sufficient.

One can see that PRIDE is comparable to SPECK-64/96 and SPECK-64/128 (members of NSA’s software-cipher family), which are based on a Feistel structure and use modular additions as the main source of non-linearity.

In addition to the above table, the recent work of Grosso et al. [GLSV14] presents LS-Designs. This is a family of block ciphers that can systematically take advantage of bitslicing in a principled manner. In this work, the authors make use of lookup tables. Therefore, a direct comparison with PRIDE is not fair as the use of lookup tables does not minimize the linear layer cost. However, to have an idea, we can try to estimate the cost of the 64-bit case of this family. They suggest two options: The first uses a 4-bit Sbox with 16-bit Lbox, and the second uses 8-bit Sbox with 8-bit Lbox. The first option has 8 rounds, which results in 64 non-linear operations, 128 XORs, and 128 table lookups in total. The second one has 6 rounds, which takes 72 non-linear operations, 144 XORs, and 48 table lookups. For linear layer cost, we consider the XOR cost together with table lookups. Unfortunately, it is not easy to estimate the overall cost of the given two options on AVR platform as the table lookups take more than one cycle compared to the non-linear and linear operations. Another important point here to mention is that the use of lookup tables result in a huge memory utilization.

In order to observe the software performance of PRIDE on different microcontrollers, it is also implemented in C (a reference implementation) and its runtime, code size, and energy consumption is evaluated on the evaluation board EnergyMicro EF32GG-STK3700. As the compiler, we selected SimplicityStudio IDE as in the case of PRINCE. The results are given in Table 10.3.

---

Table 10.2: Performance comparison of PRIDE to other block ciphers

<table>
<thead>
<tr>
<th>Cipher</th>
<th>AES-128</th>
<th>PRESENT-128</th>
<th>CLEFIA-128</th>
<th>SEA-96</th>
<th>NOEKEON-128</th>
<th>PRESENT-128</th>
<th>SIMECK-64/128</th>
<th>BSS-128</th>
<th>PRIDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(cyc)</td>
<td>5159</td>
<td>10792</td>
<td>25648</td>
<td>17745</td>
<td>23517</td>
<td>3614</td>
<td>2607</td>
<td>2000</td>
<td>266</td>
</tr>
<tr>
<td>bytes</td>
<td>1570</td>
<td>7220</td>
<td>660</td>
<td>3046</td>
<td>386</td>
<td>364</td>
<td>716</td>
<td>182</td>
<td>186</td>
</tr>
<tr>
<td>eq.r.</td>
<td>5/10</td>
<td>1/32</td>
<td>4/31</td>
<td>1/18</td>
<td>8/92</td>
<td>1/16</td>
<td>5/12</td>
<td>34/26</td>
<td>1514</td>
</tr>
</tbody>
</table>

---

*Our C code was run on this microcontroller by Jan Zimmer.*
Table 10.3: Performance of PRIDE on 32-bit ARM CORTEX-M3

<table>
<thead>
<tr>
<th></th>
<th>Run time (clock cycles) [Enc/Dec]</th>
<th>Code size (bytes)</th>
<th>Energy (µJ) [Enc/Dec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIDE</td>
<td>3921/4267</td>
<td>863</td>
<td>19.1/20.3</td>
</tr>
</tbody>
</table>

Note that this is not the most efficient 32-bit design as it is a direct mapping of the 8-bit assembly implementation of PRIDE to C. We therefore come up with an optimized 32-bit implementation on another 32-bit ARM platform, STM32F4DISCOVERY\[8\]. The results are given in Table 10.4.

Table 10.4: Performance of PRIDE on 32-bit STM32F4DISCOVERY

<table>
<thead>
<tr>
<th></th>
<th>Run time (clock cycles) [Enc/Dec]</th>
<th>Code size (bytes) [Enc/Dec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINCE</td>
<td>1477/1541</td>
<td>354/440</td>
</tr>
</tbody>
</table>

Finally, we note that, despite its target being software implementations, PRIDE is also efficient in hardware. It can be considered a hardware-friendly design, due to its cheap linear and Sbox layers.

10.4.5 Security Analysis

The security analysis of PRIDE is not in the scope of this thesis. Due to the interdisciplinary nature of this work, the author was focused on the search and design of efficient building blocks for the cipher, together with the implementations. Therefore, we leave the security discussion of PRIDE to the original publications [ADK+14a, ADK+14b], in which the security claims for PRIDE and a security evaluation of its general construction regarding classical attacks (linear, differential) are provided.

It is worth mentioning that the performed security analysis together with third-party analyses have not revealed any critical and/or practical weaknesses of PRIDE so far; however, further analysis is of course encouraged.

10.5 Conclusion

In this work, we presented a framework, which can be extended in the future, in order to construct linear layers for block ciphers that allows to trade security against efficiency. Then, we proposed a hardware architecture to search for very efficient examples of this construction.
especially for software implementations of cryptographic block ciphers. Using this proposed highly-parallel and pipelined hardware search architecture, we found diffusion matrices with smallest possible cycle counts and code sizes. Moreover, we presented a new “software-oriented” cipher PRIDE dedicated for 8-bit microcontrollers, which makes use of the resulting linear layer diffusion matrices. Our proposal PRIDE significantly outperforms all academic solutions both in terms of code size and cycle count.

In addition to our software-efficient cipher proposal, our work – to the best of our knowledge – is the first attempt to search for efficient cipher linear layers for software implementations using a hardware architecture. Our highly parallel architecture made it possible to search for various parameters in a very short amount of time, which would not be possible on a software search platform.

For the hardware-powered search for linear layers, we can extend our work to cover a larger spectrum of microcontroller instruction sets, including the highly popular PIC and ARM [ARM] instruction sets, as a natural result of using [FPGA].

Finally, regarding PRIDE, we obviously encourage further cryptanalysis.
Part IV

Conclusion
Chapter 11
Concluding Remarks

In this thesis, we have presented efficient implementations of lightweight primitives on different platforms and proposed new lightweight block ciphers targeting different metrics. In the following, we summarize our results and discuss the contributions of this thesis.

Contents of this Chapter

11.1 Implementations and Applications of Lightweight Primitives . . . . . . . . . 137
11.2 Lightweight Block Cipher Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138
11.3 ISE Design for (Lightweight) Crypto Primitives . . . . . . . . . . . . . . . . . 138

11.1 Implementations and Applications of Lightweight Primitives

In this thesis, we have presented efficient implementations of some prominent lightweight symmetric-key primitives and hash functions and their alternative implementations on different platforms for better performance. Furthermore, we have also shown some special use case examples (i.e., IPSecco) of lightweight cryptography.

Firstly, we evaluated area, power, and energy consumption of lightweight block cipher implementations and different AES architectures. We discussed the differences in dynamic power consumption in relation to the area. Our results showed that the dynamic power consumption has an important effect on the energy and power consumption of these cryptographic implementations.

Following that, we switched our implementation focus to FPGAs, as they become more and more attractive alternatives for lightweight security applications. We made use of the existing RAMs on the FPGA in order to get a better slice count. We presented two different RAM-based implementations of the lightweight cipher algorithm PRESENT. With their lowest reported slice counts, both implementations are suitable for lightweight applications.

Then, we performed a suitability analysis of the SHA-3 finalists for lightweight applications. We limited ourselves to reach lightweight figures for area, which also results in good numbers for average power consumption in most applications; and tried to get the lowest possible recorded gate counts for all five finalists. We avoided the use of block memories in order to be compatible with different platforms. We successfully reached our target of lowest gate count, and presented very compact implementations of SHA-3 finalists.
We later combined some results of Chapter 5 and 6 and presented an example lightweight cryptography application. We implemented lightweight building blocks for the execution of the popular IPsec protocol on reconfigurable hardware. We showed that even a complex protocol suite, which requires several cipher standards and modes of operation, can be implemented even on very low-cost and energy-efficient FPGA. We selected algorithms that are generally designed for efficient realization in hardware and adapted them to the specification of our target device. We were therefore able to achieve a significant reduction in terms of area utilization compared to related work. Our results show that standardized security can be achieved on reconfigurable hardware with only a modest investment on slice resources. This leaves us with more space for other functionality on the target hardware.

11.2 Lightweight Block Cipher Design

This thesis proposes two novel “lightweight” ciphers, namely PRINCE and PRIDE.

We first presented the new low-latency and low-cost cipher PRINCE, together with a variety of its implementations on different platforms, ranging from ASICs and FPGA to software implementations. PRINCE has a very small footprint and achieves the lowest latency compared to the existing lightweight solutions – without compromising security. The compactness of PRINCE is the result of an extensive search on the substitution and permutation layers of the cipher. Potential SCA-resistant constructions for PRINCE are also considered; however, we observed that such countermeasures increase the cipher area significantly.

Later, we proceeded in the “software-efficiency” direction. We started with presenting a framework to construct linear layers for block ciphers that allows to trade security against efficiency. Then, we proposed a hardware architecture to search for very efficient examples of this construction especially for software implementations of cryptographic block ciphers. Using our proposed highly-parallel and pipelined hardware search architecture, we found diffusion matrices with smallest possible cycle counts and code sizes. Furthermore, we presented a new “software-oriented” cipher PRIDE dedicated for 8-bit microcontrollers, which uses these resulting linear layer diffusion matrices. Our proposal PRIDE significantly outperforms all academic solutions both in terms of code size and cycle count. In addition to our software-efficient cipher proposal, our work – to the best of our knowledge – is the first attempt to search for efficient cipher linear layers for software implementations using a hardware architecture. Our highly parallel architecture made it possible to search for various parameters in a very short amount of time, which would not be possible on a software search platform.

11.3 ISE Design for (Lightweight) Crypto Primitives

There have been many different ISE proposals for cryptographic applications so far. However, most of the previous works have been on the implementation of cipher-specific instructions. Furthermore, the previous works also do not target (lightweight) cryptography for resource-constrained devices. In this thesis, we were able to improve the software implementations of cryptographic block ciphers on small CPUs through a new crypto ISE proposal. We came up with a non-linear/linear ISE NLU which can implement non-linear and linear operations of block ciphers. Using the four new NLU instructions, we show that we achieve time-area product
reductions of 20-68% for the widely-used ciphers PRESENT, Serpent, CLEFIA, and AES with a very low area overhead for the hardware extension unit.
Chapter 12
Directions for Future Research

This chapter points out the open problems, topics that might need further investigation, and ideas for future research based on the problems and results of the research carried out during the course of this thesis.

Contents of this Chapter

12.1 Further Lightweight Crypto Implementations and Applications . . . . . . 141
12.2 Potential Lightweight Cipher Design Ideas . . . . . . . . . . . . . . . . . . . 142
12.3 Further (Lightweight) Crypto ISE Ideas . . . . . . . . . . . . . . . . . . . . 142

12.1 Further Lightweight Crypto Implementations and Applications

Within the scope of this thesis, we evaluated a number of well-known lightweight ciphers. An extended evaluation including any other existing and upcoming (lightweight) block and stream ciphers would definitely be more beneficial to serve as a design/selection guide for cryptographers. In our evaluation, we used industrial tools in order to generate dynamic power consumption results; however, one can also investigate the HDL implementation of a cipher and estimate the dynamic power consumption through measuring the toggle activity. The main idea of this approach is to measure the total number of bit toggles that happen during the encryption of a single block of a given algorithm. Such a metric can be especially helpful when we consider energy. The energy consumption in CMOS circuits is dominated by the dynamic power dissipation and it is directly proportional to the number of bit toggles. Hence, it provides a good prediction for the energy consumption of a cipher. In the future, the relation between “the number of toggles from an HDL implementation” and “the power reports from the synthesis of this implementation” can be investigated.

For our RAM-based PRESENT implementations on FPGA, further capabilities can be proposed. For instance, our on-slice Sbox and on-RAM Sbox designs are good candidates for the application of the shared Sbox technique in [PMK+11] and the block memory content scrambling technique given in [GM11] for SCA resistance. In an extended version of this study, these SCA countermeasures can be applied and their effects on the area can be observed together with their resistance to SCA attacks.

We have used some of our proposed implementations in an example application, IPSecco, in this thesis. We can apply SCA-resistant techniques also to IPSecco. Generic protection
layers [GM1] can be integrated in future as well as countermeasures that are specifically designed to use in lightweight applications. Another interesting topic would be the investigation of different algorithms and their combinations, such as CLEFIA, Keccak, HIGHT, etc., and also the primitives that can make use of resource-sharing. One common solution for efficient resource-sharing is partial reconfiguration, which is only available on more powerful, expensive, and larger FPGAs. Therefore, further effort can be put into the integration and sharing of components between different ciphers (e.g., Sbox lookup tables, block RAM, or registers for keys) in order to further reduce the size of our design.

Furthermore, similar to the case of IPSecco, our other designs and implementations can also contribute to different applications. Moreover, we can also come up with realizations of different applications and prove the real-life use cases of lightweight cryptography.

12.2 Potential Lightweight Cipher Design Ideas

In our work, we identified some “unaddressed” metrics, such as low-latency and software-efficiency, in lightweight cryptography and proposed new lightweight block ciphers in order to provide solutions for these gaps. However, more design metrics can be identified and novel cryptographic primitives and design techniques can be suggested, also making use of the ideas and techniques that we suggested in our work. Especially, for the case of PRIDE, our framework for linear layer construction can be extended in the future. For the hardware-powered search for linear layers, we can extend our work to cover a larger spectrum of microcontroller instruction sets, including the highly popular PIC and ARM instruction sets, as a natural result of using FPGAs.

Nevertheless, further implementations of PRINCE and PRIDE on different hardware and software platforms are encouraged together with further cryptanalysis.

12.3 Further (Lightweight) Crypto ISE Ideas

As a future work, we can address the necessary compiler modifications for the AVR instruction set in order to include NLU. This would be beneficial for a fully-running “AVR with NLU” toolchain. We can also propose a very simple custom microcontroller including NLU in its instruction set. In this case, we can come up with our own toolchain, which includes NLU in its compiler from scratch.

The NLU ISE design can also be extended to different (4-bit, 16-bit, 32-bit, 64-bit, etc.) microcontroller architectures. 32-bit ARM instruction set can be a good starting point due to its wide utilization.

Moreover, our proposed and future NLU instructions can be used to implement many other (lightweight) primitives. An extended evaluation of a large set of crypto primitives using NLU would be very beneficial in order to observe time-area product reductions.
PRINCE Diffusion Matrices

In this appendix, we present the two $64 \times 64$ diffusion matrices ($M'$ and $M$) used in PRINCE middle layer and round linear layers.
Matrix $M'$
Matrix $M$

$$M = \begin{pmatrix}
0000000000000000000000000000000000000000000000000000000000000000
\end{pmatrix}$$
PRINCE Test Vectors

In this appendix, we present five different test vectors for PRINCE encryption/decryption, with intermediate values.
<table>
<thead>
<tr>
<th>Description</th>
<th>Test Vector 1</th>
<th>Test Vector 2</th>
<th>Test Vector 3</th>
<th>Test Vector 4</th>
<th>Test Vector 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cipherpart 1</td>
<td>8186660000002f4</td>
<td>9e156e3f3ed524</td>
<td>78ac6c57b6dfc</td>
<td>604e86c3b2c0a</td>
<td>wa2583a58f9ef</td>
</tr>
<tr>
<td>Key (k)</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>Key (k)</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>k_prime</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>Round key</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 1 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 2 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 3 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 4 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 5 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 6 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 7 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 8 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 9 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 10 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 11 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 12 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 13 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 14 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>After Round 15 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th>Test Vector 6</th>
<th>Test Vector 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decryption</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 1 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 2 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 3 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 4 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 5 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 6 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 7 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 8 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 9 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 10 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 11 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 12 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 13 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 14 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
<tr>
<td>After Round 15 key add</td>
<td>0000000000000000</td>
<td>ffffffffffffffff</td>
</tr>
</tbody>
</table>
PRIDE Diffusion Matrices

In this appendix, we present PRIDE’s software-friendly four $16 \times 16$ matrices $L_0$, $L_1$, $L_2$, $L_3$ and their inverses.
PRIDE Diffusion Matrices

\[ L_0 \text{ and } L_0^{-1} = \begin{pmatrix}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 
\end{pmatrix} \]

\[ L_1 = \begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 
\end{pmatrix} \]

\[ L_1^{-1} = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 
\end{pmatrix} \]
### PRIDE Diffusion Matrices

**$L_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}**

**$L_2^{-1} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}**

**$L_3 \& L_3^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}**
PRIDE Test Vectors

In this appendix, we present five different test vectors for PRIDE encryption/decryption, with intermediate values.
**Encryption**

<table>
<thead>
<tr>
<th>Test Vector 1</th>
<th>Test Vector 2</th>
<th>Test Vector 3</th>
<th>Test Vector 4</th>
<th>Test Vector 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Vector 1</td>
<td>158</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key (k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After Pre-whitening</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After Fk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Round 1 key</td>
<td>0x10a50510010c5</td>
<td>0x10a50510010c5</td>
<td>0x10a50510010c5</td>
<td>0x10a50510010c5</td>
</tr>
<tr>
<td>Round 2 key</td>
<td>0x00400010a0a5</td>
<td>0x00400010a0a5</td>
<td>0x00400010a0a5</td>
<td>0x00400010a0a5</td>
</tr>
<tr>
<td>Round 3 key</td>
<td>0x8586208a15f8</td>
<td>0x8586208a15f8</td>
<td>0x8586208a15f8</td>
<td>0x8586208a15f8</td>
</tr>
<tr>
<td>Round 4 key</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>Round 5 key</td>
<td>0x00f8c46c4f5f6f</td>
<td>0x00f8c46c4f5f6f</td>
<td>0x00f8c46c4f5f6f</td>
<td>0x00f8c46c4f5f6f</td>
</tr>
<tr>
<td>After Round 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After Round 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After Round 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After Round 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After Round 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PRIDE Test Vectors**

- **Test Vector 1**: 73c58a40244c29d9
- **Test Vector 2**: 498617a4673ddfc9
- **Test Vector 3**: 4d1f4309cf378772
- **Test Vector 4**: 0000000000000000
- **Test Vector 5**: d3d9f9eabd65e25c

**Table**

- **Test Vector 1**: 73c58a40244c29d9
- **Test Vector 2**: 498617a4673ddfc9
- **Test Vector 3**: 4d1f4309cf378772
- **Test Vector 4**: 0000000000000000
- **Test Vector 5**: d3d9f9eabd65e25c
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>ANF</td>
<td>Algebraic Normal Form</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>CHES</td>
<td>International Workshop on Cryptographic Hardware and Embedded Systems</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CTR</td>
<td>Counter (mode of operation)</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>ECC</td>
<td>Elliptic Curve Cryptography</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Erasable Programmable Read-Only Memory</td>
</tr>
<tr>
<td>EmSec</td>
<td>Embedded Security</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out (memory)</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standard</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HGI</td>
<td>Horst Görtz Institute for IT-Security</td>
</tr>
<tr>
<td>HMAC</td>
<td>Hash-based Message Authentication Code</td>
</tr>
<tr>
<td>HW</td>
<td>Hamming Weight</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>LFSR</td>
<td>Linear Feedback Shift Register</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
</tbody>
</table>
Abbreviations

PC  Personal Computer
RAM  Random Access Memory
RISC  Reduced Instruction Set Computer
RFID  Radio Frequency IDentification
RSA  Rivest Shamir and Adleman
RUB  Ruhr University Bochum
SCA  Side-Channel Attacks
SHA  Secure Hash Algorithm
SRAM  Static Random Access Memory
USB  Universal Serial Bus
XOR  Exclusive OR
ACM  Association for Computing Machinery
ACNS  International Conference on Applied Cryptography and Network Security
AFRICACRYPT  Annual International Conference on the Theory and Applications of Cryptology
AH  Authentication Header
ALU  Arithmetic Logic Unit
APN  Almost Perfect Non-linear
ARX  Addition-Rotation-XOR
ARITH  IEEE Symposium on Computer Arithmetic
ASAP  IEEE International Conference on Application-specific Systems Architectures and Processors
ASIACRYPT  Annual International Cryptology Conference
ASP  Application-Specific Processor
CARDIS  Smart Card Research and Advanced Application Conference
CIS  International Conference on Computational Intelligence and Security
CMS  IFIP TC-6 TC-11 Conference on Communications and Multimedia Security
CRYPTO  Annual International Conference on the Theory and Application of Cryptology and Information Security
CT-RSA  RSA Conference Cryptographers’ Track
DATE  Design, Automation & Test in Europe
DSD  EUROMICRO Conference on Digital System Design
EUROCRYPT Annual International Conference on the Theory and Applications of Cryptographic Techniques

ESP Encapsulating Security Payload

FPL International Conference on Field Programmable Logic and Applications

FPU Floating Point Unit

FSE International Workshop on Fast Software Encryption

GCM Galois/Counter Mode

GE Gate Equivalence

GF Galois Field

IACR International Association for Cryptologic Research

ICCD International Conference on Computer Design

ICCSA International Conference on Computational Science and Its Applications

IEEE Institute of Electrical and Electronics Engineers

IKE Internet Key Exchange

IMA Institute of Mathematics and its Applications

INDOCRYPT International Conference on Cryptology in India

IO Input/Output

IP Internet Protocol

IPDPS IEEE International Parallel and Distributed Processing Symposium

IPSec Internet Protocol Security

ISE Instruction Set Extension

LightSec International Workshop on Lightweight Cryptography for Security and Privacy

LNCS Lecture Notes in Computer Science

MDS Maximum Distance Separable

MIPS Million Instructions per Second

MIT Massachusetts Institute of Technology

NESSIE New European Schemes for Signatures, Integrity, and Encryption

NLU Non-linear/Linear Unit

NRE Non-recurring Engineering

NSA US National Security Agency

ReConFig International Conference on Reconfigurable Computing and FPGAs
Abbreviations

**RFIDSec** International Workshop on RFID Security and Privacy

**SAC** International Workshop on Selected Areas in Cryptography

**SecPriWiMob** IEEE International Workshop on Security and Privacy in Wireless and Mobile Computing, Networking, and Communications

**SECSI** Secure Component and System Identification Workshop

**SLR** Shift Logical Right

**SP** Substitution-Permutation

**SPL** Southern Programmable Logic Conference

**SPN** Substitution-Permutation Network

**SSL** Secure Sockets Layer

**UBI** Unique Block Iteration

**VLSI** Very Large Scale Integration

**WAIFI** International Workshop on the Arithmetic of Finite Fields

**WISA** International Workshop on Information Security Applications

**WiSec** ACM Conference on Wireless Network Security

**XNOR** Exclusive NOR
Bibliography
Bibliography


AVRAES: AVRAES: The AES block cipher on AVR controllers. [AVR]
http://point-at-infinity.org/avraes/


Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak Sponge Function Family. Submission to NIST SHA-3 Competition, 2011. [BDPA11]


[Cad] Cadence. NC-Verilog Simulator.

Bibliography


[Fri] Limor Fried. PIC vs. AVR. http://www.ladyada.net/library/picvsavr.html


Bibliography


Bibliography


[Nan] NanGate. The NanGate 45nm Opencell Library. \url{http://www.nangate.com/?page_id=2325}.


[NTR] NTRU. Quantum-resistant Cryptography. \url{http://tbuktu.github.io/ntru/}.


Bibliography


List of Figures
## List of Figures

1.1 Development of computing devices over the last 50 years ........................................... 4
1.2 The gaps in lightweight cryptography ................................................................. 6

2.1 Branches of cryptology and cryptographic primitives .............................................. 10
2.2 Illustration of SPN ................................................................. 10

3.1 Xilinx Virtex-6 ML605 Evaluation Board ............................................................. 16
3.2 STM32F4DISCOVERY ................................................................. 18

4.1 Round-based implementation of a generic block cipher ......................................... 23
4.2 Round-based implementation of a generic block cipher – no key schedule ............. 23

5.1 PRESENT data flow scheme in the RAM-based design for a 3-round example ... 31
5.2 PRESENT permutation scheme ................................................................. 32
5.3 PRESENT cycle count for the state processing phases of each round (on-slice Sbox version) ................................................................. 33
5.4 PRESENT cycle count for the first state processing phase of each round (on-RAM Sbox version) ................................................................. 34
5.5 PRESENT cycle count for the key expansion phase of each round (on-slice Sbox version) ................................................................. 34
5.6 PRESENT cycle count for the key expansion phase of each round (on-RAM Sbox version) ................................................................. 35
5.7 PRESENT cycle count for the final round (same in both on-slice and on-RAM Sbox versions) ................................................................. 35
5.8 PRESENT block diagram (key expansion module shown on the right) for on-slice Sbox version ................................................................. 36
5.9 PRESENT block diagram (key expansion module shown on the right) for on-RAM Sbox version ................................................................. 36

6.1 BLAKE compression function ................................................................. 41
6.2 One round of BLAKE and the underlying $G_i$ function ......................................... 41
6.3 BLAKE serial architecture ................................................................. 42
6.4 $G_i$ half function ................................................................. 42
6.5 BLAKE serial data flow ................................................................. 43
6.6 BLAKE timing diagram ................................................................. 44
6.7 Grostl compression function ................................................................. 44
6.8 Grostl construction function $f$ (left) and output function $\Omega$ (right) ............. 45
6.9 $P$ and $Q$ permutations ................................................................. 45
6.10 Grostl serial architecture ................................................................. 46
6.11 Details of $P/Q$ block .................................................. 47
6.12 Data flow for $4 \times 4$ toy version .................................. 47
6.13 Grostl timing diagram ............................................... 48
6.14 JH compression function ............................................ 49
6.15 Structure of $F_8$ compression function (left) and $E_8$ function (right) 49
6.16 Three layers of round function ..................................... 50
6.17 JH serial architecture .............................................. 50
6.18 JH serial flow .......................................................... 51
6.19 JH timing diagram .................................................... 52
6.20 Sponge construction of Keccak ..................................... 53
6.21 Keccak-$f$ function and steps of the function ..................... 53
6.22 Keccak serial architecture ........................................... 54
6.23 Keccak timing diagram .............................................. 55
6.24 Keccak data flow ...................................................... 55
6.25 Skein normal hashing scheme ...................................... 56
6.26 Four rounds of ThreeFish-512 ..................................... 57
6.27 Skein serial architecture ............................................ 57
6.28 Skein serial flow ...................................................... 58
6.29 Skein timing diagram ................................................ 59
6.30 Interface model ....................................................... 59
7.1 Integrating IPSecco .................................................... 65
7.2 Architecture of the HMAC core ..................................... 67
7.3 Architecture of the PRESENT/GCM core ......................... 69
7.4 Architecture of the RAM-based PRESENT implementation ........ 70
7.5 Architecture of the serial PRESENT-80 implementation .......... 71
7.6 Architecture of the serial PRESENT-128 implementation ......... 72
7.7 Architecture of the ECC core ....................................... 73
8.1 Sbox distributions with respect to gate counts ..................... 89
8.2 Round-based implementation of PRINCE .......................... 92
9.1 Overall structure of NLU ............................................... 102
9.2 Non-linear unit ....................................................... 104
9.3 Matrix multiplication ................................................ 105
9.4 Linear unit – detailed circuit ...................................... 106
10.1 Instruction flow with quality check – an example with 8 instructions 123
10.2 Overall instruction flow – showing source and destination skips at each stage 123
10.3 Reconfigurable “optimal linear layer” search architecture ........ 125
List of Tables
## List of Tables

4.1 Performance and energy numbers of various \text{AES} realizations all using 128-bit key lengths ................................................................. 26
4.2 Performance and energy numbers of lightweight block ciphers implementations ................................................................. 26
5.1 Performance comparison ........................................................................... 37
6.1 BLAKE specifications ............................................................................. 41
6.2 Grostl specifications .............................................................................. 45
6.3 JH specifications .................................................................................... 49
6.4 Keccak specifications ............................................................................. 52
6.5 Skein specifications .............................................................................. 56
6.6 Comparison of our work with previous works .......................................... 60
7.1 IPSecco configurations ........................................................................... 66
7.2 Characteristics of our implementation for Xilinx Spartan-3 .................. 74
7.3 Characteristics of our implementation for Xilinx Spartan-6 .................. 75
8.1 All Sboxes for the PRINCE-family up to affine equivalence ................. 86
8.2 Area/power comparison of unrolled versions of PRINCE and other ciphers 91
8.3 Extrapolated area of unrolled versions of other ciphers against PRINCE 91
8.4 Performance comparison of round-based versions of PRINCE and other ciphers 92
8.5 Performance of round-based versions of other ciphers against PRINCE 92
8.6 Performance comparison of unrolled versions of PRINCE and other ciphers on FPGA ........................................................................... 93
8.7 Performance comparison of round-based versions of PRINCE and other ciphers on FPGA ................................................................. 93
8.8 Performance of PRINCE on 32-bit ARM CORTEX-M3 ...................... 94
8.9 Performance of PRINCE on 8-bit Atmel ATmega128 .......................... 94
9.1 \text{NLU} instructions .............................................................................. 102
9.2 Performance comparison of ciphers implemented with/without NLU ....... 114
10.1 Limited instruction set ....................................................................... 124
10.2 Performance comparison of PRIDE to other block ciphers .................. 132
10.3 Performance of PRIDE on 32-bit ARM CORTEX-M3 ...................... 133
10.4 Performance of PRIDE on 32-bit STM32F4DISCOVERY ................... 133
About the Author

Author information as of February 2015.

Personal Information

Elif Bilge Kavun
Born in İzmir, Turkey on 22/12/1986

Education

■ 06/2011–02/2015: Ph.D. (Dr.-Ing.) in Electrical Engineering and Information Technology, Ruhr-Universität Bochum, Germany

■ 08/2008–09/2010: M.Sc. in Cryptography, Middle East Technical University, Ankara, Turkey

■ 09/2004–07/2008: B.Sc. in Electronics and Telecommunications Engineering, İzmir Institute of Technology, Turkey

■ 01/2008–06/2008: Erasmus Exchange Student at Université Joseph Fourier - Grenoble, France


Professional and Academic Experience

■ 12/2014–Present: Digital Design Engineer, Infineon Technologies AG, Munich, Germany

About the Author

■ 03/2014–05/2014: Interim Engineering Intern, Product Security Group, Qualcomm Technologies, Inc., San Diego, CA, USA
■ 01/2011–05/2011: Research Assistant, Hardware Security Group, Fraunhofer - AISEC (former Fraunhofer - SIT), Munich, Germany
■ 03/2008–06/2008: Intern, TIMA Laboratory, Institut Polytechnique de Grenoble, France

Academic Awards

■ 06/2011: Middle East Technical University, M.Sc. Best Thesis Award
Author’s Publications

Author’s publications as of February 2015.

Peer-Reviewed Publications in Journals


Peer-Reviewed Publications in the Proceedings of International Conferences and Workshops


- **E. B. Kavun** and T. Yalcın. RAM-Based Ultra-Lightweight FPGA Implementation of PRESENT. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig’11)*. IEEE Computer Society, 2011.


Other Publications


Accepted Posters and Presentations at International Conferences and Workshops (without proceedings)


Invited Talks

■ **25/05/2012**: Lightweight Cryptography – What do we really mean by “Lightweight”?. IT-Sicherheits-Stammtisch, Industrie- und Handelskammer Mittleres Ruhrgebiet, Bochum, Germany.

■ **17/05/2012**: Lightweight Cryptography – What do we really mean by “Lightweight”? 5th International Conference on Information Security and Cryptology (ISCTurkey’12), Ankara, Turkey.

Attended Conferences and Workshops During Ph.D.

■ CRYPTO’14, Santa Barbara, CA, US (August 2014)
■ CrossFyre’14, Bochum, Germany (July 2014)
■ ReConFig’13, Cancún, Mexico (December 2013)
■ CHES’13, Santa Barbara, CA, US (August 2013)
■ CRYPTO’13, Santa Barbara, CA, US (August 2013)
■ RFIDSec’13, Graz, Austria (July 2013)
■ CrossFyre’13, Leuven, Belgium (June 2013)
■ ARITH 21, Austin, Texas, US (April 2013)
■ ASIACRYPT’12, Beijing, China (December 2012)
■ ICACM, Ankara, Turkey (October 2012)
■ CHES’12, Leuven, Belgium (September 2012)
■ RFIDSec’12, Nijmegen, the Netherlands (July 2012)
■ CrossFyre’12, Eindhoven, the Netherlands (June 2012)
■ ISCTurkey’12, Ankara, Turkey (May 2012)
■ 3rd SHA-3 Conference, Washington, D.C., US (March 2012)
■ SHARCS’12, Washington, D.C., US (March 2012)
■ ReConFig’11, Cancún, Mexico (December 2011)
■ ECRYPT Workshop on Lightweight Cryptography, Louvain-la-Neuve, Belgium (November 2011)
■ CARDIS’11, Leuven, Belgium (September 2011)
■ CryptArchi’11, Bochum, Germany (June 2011)