Resource-constrained and Resource-efficient
Modern Cryptosystem Design

Aydin Aysu

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Patrick R. Schaumont, Chair
Leyla Nazhandali
Cameron D. Patterson
Danfeng Yao
Yaling Yang

June 16, 2016
Blacksburg, Virginia

Keywords: Embedded Systems, Lightweight Cryptography, Post-quantum Cryptography, Physical Unclonable Functions

Copyright 2016, Aydin Aysu
In the context of a system design, resource-constraints refer to severe restrictions on allowable resources, while resource-efficiency is the capability to achieve a desired performance and, at the same time, to reduce wasting resources. To design for low-cost platforms, these fundamental concepts are useful under different scenarios and they call for different approaches, yet they are often mixed. Resource-constrained systems require aggressive optimizations, even at the expense of performance, to meet the stringent resource limitations. On the other hand, resource-efficient systems need a careful trade-off between resources and performance, to achieve the best possible combination. Designing systems for resource-constraints with the optimizations for resource-efficiency, or vice versa, can result in a suboptimal solution.

Using modern cryptographic applications as the driving domain, I first distinguish resource-constraints from resource-efficiency. Then, I introduce the recurring strategies to handle these cases and apply them on modern cryptosystem designs. I illustrate that by clarifying the application context, and then by using appropriate strategies, it is possible to push the envelope on what is perceived as achievable, by up to two orders-of-magnitude.

In the first part of this dissertation, I focus on resource-constrained modern cryptosystems. The driving application is Physical Unclonable Function (PUF) based symmetric-key authentication. I first propose the smallest block cipher in 128-bit security level. Then, I show how to systematically extend this design into
the smallest application-specific instruction set processor for PUF-based authentication protocols. I conclude this part by proposing a compact method to combine multiple PUF components within a system into a single device identifier.

In the second part of this dissertation, I focus on resource-efficient modern cryptosystems. The driving application is post-quantum public-key schemes. I first demonstrate energy-efficient computing techniques for post-quantum digital signatures. Then, I propose an area-efficient partitioning and a Hardware/Software code-sign for its implementation. The results of these implemented modern cryptosystems validate the advantage of my approach by quantifying the drastic improvements over the previous best.
Mihály Csíkszentmihályi has an exceptional book on human psychology, in which he defines flow as an ecstatic state of creation where one feels intrinsic motivation with a great inner clarity and with an immense sense of serenity. My entire PhD journey was a total and utter “flow”, and for that, I have to sincerely thank to my advisor, Dr. Patrick Schaumont. His guidance and encouragement always kept me at flow. None of this work would be possible without his devotion to my research and to my development as a researcher.

I acknowledge my committee members Dr. Leyla Nazhandali, Dr. Cameron Patterson, Dr. Yaling Yang, and Dr. Danfeng Yao, for their valuable feedback on my dissertation. I also express my gratitude to Dr. Patterson, for his help during my first ever paper published at Virginia Tech, which is now becoming a foundational paper on designing compact arithmetic units for post-quantum arithmetic, and to Dr. Nazhandali, for her contributions to the SIMON project.

I feel myself privileged to have co-authored papers with extremely talented researchers. Their effort made this contribution much more significant. Thank you Ege Gulcan, Nahid Farhady Ghalaty, Bilgiday Yuce, Zane Franklin, Moein Pahlavan Yali, Harsha Mandadi, Shravya Gaddam, Carol Pinto, Luke Wegryn, Dr. Daisuke
Moriyama and Dr. Moti Yung. I also thank National Science Foundation and Cisco Systems Inc. for funding my research projects, and to Dr. Eric Xu Guo for the wonderful summer internship experience at Qualcomm Inc.

I am indebted to Dr. İlker Hamzaoğlu, my undergraduate and MS advisor, for accepting to advise an average GPA student who skipped his class time and again to perform at concerts with one of his numerous bands. I am grateful to Dr. Erkay Savaş for introducing me to the amazing world of cryptography and to Murat Sayinta for teaching me the nuts and bolts of hardware design.

Completing this work would be much harder without the limitless support of my family and friends. The long Starcraft sessions with my friends was a great way to recharge. Although, I have been losing increasingly more as the years go by due to lack of practice, I look forward to giving them again a run for their money.

I do not think this section would be complete without acknowledging coffee for keeping me up, and bourbon and beer for calming me down.
## Contents

1. **Introduction**  
   1.1 The Two Ends of Computing Spectrum  
   1.2 Brave New World of Challenges for Embedded Security  
   1.3 Bringing in the Context  
   1.3.1 Resource-constrained vs. Resource-efficient Design  
   1.4 Three Optimization Strategies for Low-cost Platforms  
   1.4.1 Serialization  
   1.4.2 Precomputation  
   1.4.3 Integration with Reuse  
   1.5 Contributions / Organization

I. **Resource-constrained Cryptosystem Design**

2. **Lightweight Encryption Core**
2.1 Introduction .................................................. 26
2.2 Basics of Symmetric-key Block Ciphers .................. 28
  2.2.1 SPN Block Ciphers ........................................ 28
  2.2.2 Feistel Block Ciphers .................................... 30
2.3 SIMON: A Feistel Based Block Cipher ..................... 30
2.4 Dimensions of Parallelism for Block Ciphers .......... 32
  2.4.1 Parallelism of Encryptions ............................... 35
  2.4.2 Parallelism of Rounds .................................... 35
  2.4.3 Parallelism of Bits ....................................... 36
2.5 Bit-serial Design for FPGAs ............................... 36
  2.5.1 Datapath .................................................. 37
  2.5.2 Control ................................................... 38
  2.5.3 Bit-serializing SIMON .................................... 40
2.6 Hardware Architecture ...................................... 40
2.7 Implementation Results ..................................... 47
2.8 Lessons Learned and Looking into the Future .......... 50
  2.8.1 Feistel Goes Well with FPGAs .......................... 51
  2.8.2 Correlate Operations with LUT-input Size ............ 51
  2.8.3 Think About Slices, not Only LUTs or Registers ....... 52
4.1.1 Hardware Attestation with Fusion PUF .......................... 92
4.1.2 Contributions and Outline of the Chapter ...................... 93
4.2 Parameter Definition in PUF Protocols ............................ 94
  4.2.1 Background on Silicon PUF Technology — Part 2 ............. 95
  4.2.2 Protocol Parameter Design Flow .............................. 97
4.3 PUF Fusion ....................................................... 99
  4.3.1 Protocol-level Fusion ......................................... 99
  4.3.2 Hardware-level Fusion ....................................... 101
  4.3.3 Circuit-level Fusion ......................................... 104
4.4 Implementation Results ............................................. 104
  4.4.1 Parameter Extraction for Fusion PUF .......................... 105
  4.4.2 Comparison with Previous Work .............................. 108
4.5 Conclusions and Future Extensions ................................. 109

II Resource-efficient Cryptosystem Design .......................... 111

5 Energy-efficient Post-quantum Signatures ........................... 118
  5.1 The Need for Alternative Public-key Cryptosystems .............. 119
  5.2 Precomputation for Post-quantum Embedded Systems .......... 121
5.2.1 Precomputation as a Re-emerging Topic in Cryptographic Computing ........................................ 123
5.2.2 Moving Towards the Precomputed Execution ................................................................. 124
5.2.3 Contributions and Organization ....................................................................................... 128
5.3 Precomputation of Hash-based Signature Schemes ............................................................. 129
5.3.1 Hash-based Signatures ..................................................................................................... 129
5.3.2 Precomputation for Hash-based Signatures ..................................................................... 131
5.4 Target Platform .................................................................................................................. 135
5.5 Implementation Results ....................................................................................................... 137
5.5.1 Implementation Parameters ........................................................................................... 138
5.5.2 Impact of the Precomputation on Run-time Performance ............................................. 140
5.5.3 Impact of the Precomputation on Availability ................................................................. 142
5.5.4 Comparison with Related Work ....................................................................................... 144
5.6 Conclusions ....................................................................................................................... 145

6 Area-efficient Post-quantum Signatures 146

6.1 Introduction ....................................................................................................................... 147
6.1.1 Novelty ......................................................................................................................... 148
6.2 The Impact of Proposed Methods on Efficiency ............................................................... 149
6.3 Lattice-based Signatures .................................................................................................... 152
List of Figures

1.1 The computing spectrum of electronic devices 3
1.2 (a) An example design-space with 4 possible solutions 8
1.3 (a) Principle, (b) example of serialization 12
1.4 (a) Principle, (b) example of precomputation 14
1.5 (a) Principle, (b) example of integration with reuse 19

2.1 The architecture of (a) SPN and (b) Feistel Network 29
2.2 (a) Feistel round and (b) key generation of SIMON for the 128/128 configuration 31
2.3 Representation for the dimensions of parallelism with several example designs 32
2.4 Generic block diagram for the parallelism of encryptions and rounds 34
2.5 (a) Bit-parallel datapath and (b) bit-serial datapath 37
2.6 (a) Control with up-counters, (b) control with ring counters, and (c) control of nested loops 38
2.7 Bit-serialized SIMON (a) round function and (b) key generation . . . 42
2.8 Block diagram of round operation for the bit-serialized architecture . 43
2.9 Schedule of the bit-serialized architecture . . . . . . . . . . . . . . . . 44
2.10 Block diagram of key generation for the bit-serialized architecture . . 46
2.11 Implementation results of SIMON, and comparison with the previous
work on low-cost block cipher and stream cipher implementations . . 48

3.1 Response of different devices to the same challenge . . . . . . . . . . 58
3.2 PUF protocol operations on the constrained device . . . . . . . . . . 59
3.3 The basic principle of (a) fuzzy extractors and (b) reverse fuzzy ex-
tractors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
3.4 Symbolic presentation for the RFE-2 protocol and the AGMSY’15
protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
3.5 Top-level block diagram of the ASIP . . . . . . . . . . . . . . . . . . 67
3.6 The structure of instructions . . . . . . . . . . . . . . . . . . . . . . . 69
3.7 Comparison of design methods for integration . . . . . . . . . . . . . 72
3.8 In the bit-serialized architecture $K_{i+1}$ simply becomes a 48-bit shift
register which is skipped for SIMON 96/96 . . . . . . . . . . . . . . . 74
3.9 The hardware architecture of the cryptographic operations . . . . . 75
3.10 Applying the offset coding on ASIP . . . . . . . . . . . . . . . . . . 80
3.11 Implementation results and comparison of PUF-based protocol implementations .................................................. 86

4.1 Hardware attestation using Fusion PUF .................................................... 92

4.2 Distribution of the Intra and Inter HD of n-bit PUF responses ........ 95

4.3 Design flow of PUF-based protocols .................................................. 97

4.4 Representation of the four fusion methods ...................................... 99

4.5 Different combinations of $n_1, n_2, t$ for concatenation based fusion and the associated residual entropies ......................... 107

4.6 Examples of low-latency signature generation: (a) structural-health monitoring system, (b) cyber infrastructure system for roadside electronics .................................................. 113

5.1 Energy profiles of (a) battery-operated and (b) energy-harvesting systems .................................................. 122

5.2 The typical energy levels on the supercapacitor and availability for two consecutive days .................................................. 126

5.3 Precomputation of intermediate nodes for the W-OTS .................. 132

5.4 Operation flow of (a) Mono, (b) Mono-opt, (c) Precomp, and (d) Precomp-opt execution .................................................. 134

5.5 The picture of the setup, photographed by the author .................. 135

5.6 The block diagram of the research platform .................................. 136
5.7 Energy on supercapacitor vs. availability ($\omega = 8$) .......................... 143

6.1 Resource allocation for (a) Monolithic and (b) Precomputed (c) Monolithic-par-allel (d) Precomputed-parallel ................................. 150

6.2 Practical lattice-based digital signature schemes ................................. 153

6.3 The principle of operation for the (a) monolithic and the (b) precom-puted execution of GLP signature generation ................................. 160

6.4 Partition of signature generation to hardware and software operations 162

6.5 Polynomial multiplication of $t = sc$ where $c$ is a sparse polynomial .. 163

6.6 (a) Design dimensions and (b) the required operations for polynomial multiplication ...................................................... 164

6.7 The simplified generic top-level block diagram of signature generation architecture ......................................................... 168

6.8 Hardware architecture for the polynomial multiplier of design config-uration A ................................................................. 169

6.9 Hardware architecture for the polynomial multiplier of design config-uration C ................................................................. 171

6.10 Probability of success vs. number of trials ................................. 172

A.1 The modified reverseFE protocol ........................................ 187

A.2 The AGMSY’15 protocol ........................................ 198
## List of Tables

1.1 Organization of the dissertation .............................................. 21  
2.1 SIMON — area vs. throughput ............................................. 50  
3.1 Instruction set of the ASIP .................................................. 70  
3.2 Integrating with modes-of-operation ..................................... 73  
3.3 Resource usage breakdown on the XC6SLX-2TQG144 .................. 82  
3.4 Benchmark of secure PUF-based constructions on FPGAs .......... 84  
4.1 Table of glossary for PUF-based authentication protocols ........ 94  
4.2 Experimental setup .......................................................... 105  
4.3 The results of parameter extraction for $d_1 = 0.4838$, $d_2 = 0.4867$, $e_1 = 0.0522$, $e_2 = 0.0128$ .......................... 106  
4.4 Comparison with previous work .......................................... 108  
4.5 The asymmetry of the two computing devices ........................ 115
5.1 Security reductions of pre- and post-quantum era . . . . . . . . . . . . 120
5.2 Symbols of the precomputation methodology . . . . . . . . . . . . . . 125
5.3 Energy and execution time results of hash-based signatures . . . . . . 139
5.4 Run-time hash operation comparison . . . . . . . . . . . . . . . . . . 141

6.1 Signature scheme parameters . . . . . . . . . . . . . . . . . . . . . . . 157
6.2 The implementations of the lattice-based cryptography . . . . . . . . 158
6.3 FPGA implementation results and comparisons with previous work . 174
6.4 Resource usage breakdown . . . . . . . . . . . . . . . . . . . . . . . . . 176

7.1 A summary of the chapters . . . . . . . . . . . . . . . . . . . . . . . . . 181

A.1 ReverseFE mapping on ASIP . . . . . . . . . . . . . . . . . . . . . . . 191
A.2 AGMSY’15 mapping on ASIP . . . . . . . . . . . . . . . . . . . . . . . 197
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>ASIP</td>
<td>Application-specific Instruction Set Processor</td>
</tr>
<tr>
<td>BLISS</td>
<td>Bimodal Lattice Signature Scheme</td>
</tr>
<tr>
<td>BRAM</td>
<td>Block Random Access Memory</td>
</tr>
<tr>
<td>CBC</td>
<td>Cipher Block Chaining</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>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage Frequency Scaling</td>
</tr>
<tr>
<td>ECB</td>
<td>Electronic Code Book</td>
</tr>
<tr>
<td>ECC</td>
<td>Elliptic Curve Cryptography</td>
</tr>
<tr>
<td>ECDSA</td>
<td>Elliptic Curve Digital Signature Algorithm</td>
</tr>
<tr>
<td>FAR</td>
<td>False Accept Ratio</td>
</tr>
<tr>
<td>FLOPS</td>
<td>Floating-point Operations per Second</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>Abbr.</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>FRR</td>
<td>False Reject Ratio</td>
</tr>
<tr>
<td>Gbps</td>
<td>Giga Bits per Second</td>
</tr>
<tr>
<td>GCM</td>
<td>Galois Counter Mode</td>
</tr>
<tr>
<td>GHz</td>
<td>Giga Hertz</td>
</tr>
<tr>
<td>GLP</td>
<td>Güneysu-Lyubashevsky-Pöppelmann</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Units</td>
</tr>
<tr>
<td>HD</td>
<td>Hamming distance</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet-of-Things</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>ISE</td>
<td>Integrated Synthesis Environment</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output</td>
</tr>
<tr>
<td>Kb</td>
<td>Kilobit</td>
</tr>
<tr>
<td>KB</td>
<td>Kilobyte</td>
</tr>
<tr>
<td>KHz</td>
<td>KiloHertz</td>
</tr>
<tr>
<td>LE</td>
<td>Logic Element</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-up-table</td>
</tr>
<tr>
<td>LWE</td>
<td>Learning-with-Errors</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>MHz</td>
<td>Mega Hertz</td>
</tr>
<tr>
<td>ms</td>
<td>milliseconds</td>
</tr>
<tr>
<td>MQ</td>
<td>Multivariate Quadratic</td>
</tr>
<tr>
<td>NFC</td>
<td>Near Field Communication</td>
</tr>
<tr>
<td>NSA</td>
<td>National Security Agency</td>
</tr>
</tbody>
</table>

xx
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTT</td>
<td>Number Theoretic Transform</td>
</tr>
<tr>
<td>PRF</td>
<td>Pseudo-random Function</td>
</tr>
<tr>
<td>PRNG</td>
<td>Pseudo-random Number Generation</td>
</tr>
<tr>
<td>PUF</td>
<td>Physical Unclonable Function</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio Frequency Identifier</td>
</tr>
<tr>
<td>RO</td>
<td>Ring Oscillator</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-only Memory</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest-Shamir-Adleman</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>S-box</td>
<td>Substitution Box</td>
</tr>
<tr>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>SHM</td>
<td>Structural Health Monitoring</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>SPN</td>
<td>Substitute-Permutation Network</td>
</tr>
<tr>
<td>TRNG</td>
<td>True Random Number Generator</td>
</tr>
<tr>
<td>UHF</td>
<td>Ultra High Frequency</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>W-OTS</td>
<td>Winternitz One-time Signature</td>
</tr>
<tr>
<td>WISP</td>
<td>Wireless Identification and Sensing Platform</td>
</tr>
<tr>
<td>XST</td>
<td>Xilinx Synthesis Technology</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive Or</td>
</tr>
</tbody>
</table>
This work is derived from the corpus of publications on cryptographic engineering by the author Aydin Aysu. In addition to the work discussed in this dissertation, during the course of his PhD., the author also wrote papers on design metrics for embedded security, lightweight PUF designs, PUF-based protocol design and dimensioning, and fault analysis of symmetric-key block ciphers.

A list of the peer-reviewed publications of the author is provided below in reverse chronological order.

**Peer-reviewed Journal Articles and Conference Proceedings:**


[14] A. Aysu, C. Patterson, P. Schaumont, Low-Cost and Area-Efficient FPGA Implementations of Lattice-Based Cryptography, IEEE Int. Symposium on Hardware-Oriented Security and Trust (HOST), Austin, TX, USA 2013


[19] A. Aysu, O.C. Ulusel and I. Hamzaoglu, Adaptive H.264 Multiple Reference Frame Motion Estimation Hardware, Gml Sistemler ve Uygulamalar Sempozyumu (GOMSIS), Istanbul, Turkey 2010 (Turkish)
Chapter 1

Introduction

Gone are the days where desktop computers dominate the personal computing medium. Nowadays, a variety of computing devices including tablet computers, smartphones, smartwatches, Radio Frequency Identifier (RFID) and Near Field Communication (NFC) tags, and so on, are widely adopted, and the cooperation of these numerous devices enable a myriad of brand-new applications. Ubiquitous computing, pervasive computing, Internet-of-Things (IoT) or Internet-of-Everything are all similar terms coined to describe this phenomenon.

On the other hand, cryptanalysis tools also develop over time. Recently, Stevens et al. announced the practical break of the full Secure Hash Algorithm-1 (SHA-1) [20]. This work follows a decade-long series of cryptanalytic progress and implements the attack on a cluster of graphics processing units (GPUs). The attack also has obvious implications on the later standard, SHA-2 hash function, which has a similar construction.

Within this changing ecosystem, the distributed low-cost devices that cooperate
over a physical, remote, and non-secure medium should be protected against new
types of attack vectors. Fortunately, for the case of hash functions, the cryptographic
research community have predicted the attack possibility and developed their latest
standard, SHA-3, with different characteristics. Hence, the systems with SHA-1
and SHA-2 can switch to SHA-3 and remain secure at the moment. SHA-3 is also
standardized after a careful evaluation of its performance on a number of different
target platforms, which makes the switch possible. However, there are many more
emerging cryptographic constructions to defend this new battleground. Therefore,
the suitability of such security mechanisms, especially on the low-cost devices of the
‘Things’, is essential for their impending real-world implementations. In this chapter,
we will first elaborate on the changes in the computing landscape and on how these
changes affect cryptographic tools for embedded security. Then, we will define the
key problems for this new era of cryptosystems and highlight the contributions of
this thesis.

1.1 The Two Ends of Computing Spectrum

The computing devices of today span a diverse range of capability. Figure 1.1 shows
a spectrum of these devices. At the one end, there are passively-powered or energy-
harvested RFID and NFC tags with severe area and power limitations. These de-
vices are extremely resource-constrained and it is difficult to implement complex
operations on them. For instance, the Wireless Identification and Sensing Platform
(WISP) Ultra High Frequency (UHF) RFID sensors use a low-power MSP430 micro-
controller [21]. The 16-bit microcontroller is clocked at 4 Mega Hertz (MHz) and it
provides a performance at merely a few Kilo Floating Point Operations Per Second
Figure 1.1: The computing spectrum of electronic devices

(FLOPS). Such devices can be scaled at a reasonable cost and be distributed to implement smartdust type of applications.

Land and aerial drones enable various up-and-coming commercial applications like automated express delivery. Such machines typically include a more capable microcontroller like the 32-bit Arm Cortex-M3 [22] or even a microcomputer like the
Raspberry Pi [23]. These devices run at higher clock frequencies (80-800 MHz) and can achieve Mega FLOPS. The challenge for this set of devices is to efficiently use the available resources (e.g., memory and energy) while maximizing performance.

Wearable or handheld computers such as a smartwatch, smartglass and smartphone are used by billions on a daily basis. These devices can be a local hub of surrounding less capable devices, and can collect and perform meaningful computation based on the received data. They typically use a high performance multi-processor clocked over 1 Giga Hertz (GHz) and they provide over a Giga FLOPS.

For computation-intensive applications, improving performance is possible by using massively parallel units. Vectorized multi-processors and GPUs incorporate such architectures to accelerate multimedia applications. The recent version of these machines like the Sony PlayStation-4 [24] and Nvidia GeForce GTX Titan Z [25] can achieve more than a Tera FLOPS.

Supercomputers perform complex algorithms like weather forecasting and molecular dynamics modelling that have to process massive amounts of data. As of date, July 2016, the fastest supercomputer, Tianhe-2 (Milky-Way-2), has 3,120,000 cores and could peak 33.86 peta FLOPS with a theoretical limit at 54.90 Peta FLOPS [26]. This value represents the current computation limit of a single computing infrastructure.

At the other end of the computing spectrum, the near future holds the promise of large-scale quantum computers. Such a quantum computer does not yet exists in the open literature\footnote{Although, Snowden Revelations unveiled that some government agencies have been funding millions of dollars to develop quantum computers [27].}. The true performance potential of a powerful quantum computer

is unknown, but their inherently parallel elements, quantum bits, are proven to achieve exponential improvements in the complexity of certain types of problems. Unfortunately (or depending on your standpoint, fortunately), these problems are the very same problems that we rely on to protect our digital security.

1.2 Brave New World of Challenges for Embedded Security

New challenges for embedded security arise from the ends of the computing spectrum. Figure 1.1 shows the target platforms to defend and new machines that can attack them. The nature of these computing devices reveals new challenges for the embedded security, and cryptography addresses these problems with new constructions. These new cryptographic solutions are the driving applications of my dissertation.

On the attackers side, quantum computers with quantum cryptanalysis crack the foundations of security assumptions. Indeed, they are proven to break the majority of current cryptosystems [28], [29]. Therefore, post-quantum cryptography evolved as a new field to investigate alternative schemes that rely on fundamentally different principles. Such constructions will become the primary shelter in case of a breakthrough in quantum computing or also in traditional cryptanalysis.

On the defenders side, low-cost mobile platforms like RFID/NFC tags and microcontrollers operate in remote locations, and interact with the physical world. Hence, there is a need for physical assurance on such devices and a mechanism to secure their communications. However, it is difficult to incorporate complex security mechanisms on such systems. Cryptography handles these problems through
two new branches of study. First, Physical Unclonable Functions (PUFs) propose promising methods to form a low-cost hardware-root-of-trust and to provide a physical proof. Second, lightweight cryptography studies low-complexity algorithms to perform cryptographic operations.

*How to design modern cryptosystems for low-cost devices?* As a result of these changes, in recent years, a lot of new cryptographic primitives regarding post-quantum cryptography, lightweight cryptography, and PUFs have appeared in the literature. In an IoT ecosystem, these components have to be mapped into low-end devices under severe resource and performance limitations. Moreover, they should be optimized depending on the requirements of the application domain and the capabilities of the target device. For example, passively powered RFID tags are area- and power-constrained with relaxed performance specifications, while energy-harvested and real-time systems are bounded both by energy resources and latency requirements. This complexity calls for a structured approach that clarifies the application context and distinguishes the different types of target platforms.

### 1.3 Bringing in the Context

So far, much of the cryptographic engineering work on modern cryptosystems has been of the feasibility kind. The questions of interests were, for example, ‘can we map a post-quantum signature on a low-cost microcontroller?’, ‘can we realize a stable key generation from PUF constructions’, ‘can we implement a lightweight encryption module in less than 1000 gate equivalent?’, and so on.

However, this feasibility work has to evolve into a different study. The questions
for this new research are ‘can we compute real-time post-quantum signatures on driving cars?’, ‘can we secure a Structural Health Monitor (SHM) on a bridge running of solar energy?, ‘can we implement a full authentication protocol with PUFs on an IoT node?’). The difference between these two sets of questions is context. Context is present, especially for embedded computing systems, whenever we consider real-world integration of cryptographic engineering.

Hence, feasibility-driven designs mature into context-driven designs. The context for low-cost platforms promotes resource-constrained and resource-efficient designs. Establishing this context yields a specialized design for that very context and enables achieving significant improvements over feasibility-driven generic designs. Next, we discuss the two key concepts in more details.

### 1.3.1 Resource-constrained vs. Resource-efficient Design

The blue background in Figure 1.2 shows a design-space with two dimensions (performance and resource usage) and with 4 possible solutions ($A$, $B$, $C$, and $D$). How can we rank these solutions? Obviously, $A$ is a better option than $D$ as it is both smaller and faster. In all other pairwise comparisons, one is not strictly a better solution than the other and being better depends on the requirements of the target application. Resource-constrained refers to being restricted by low resources. For example, IoT applications on RFID/NFC tags are resource-constrained because they require a design that is as small as possible while the execution time can be long [30]. $B$ is the resource-constrained solution making it the viable candidate among 4 alternatives in such a scenario.

On the other hand, some applications like low-latency encryption [31] are bounded
both with the allowable resources and performance. A designer then has to strive for resource-efficiency, finding the best combination of resource and performance. One method to test resource-efficiency is to measure the distance of the solutions from the origin, which reveals $A$ as the most resource-efficient solution.

Finally, some applications like video processing are performance-constrained; the system has to process a certain number of video frames in real-time to be compliant with the high-definition multimedia standards [15]. Therefore, the design has to meet a high performance but it can use much more resources. In that setting, $C$ becomes a more suitable solution compared to others.

Before proceeding further, we should clarify that there is indeed a confusion...
about constraints vs efficiency in modern cryptosystem design. This is even evident in the top-level publications in this field. To highlight the severity of this confusion on efficiency and constraints, below we provide several examples only from the last 2 years of publications at the workshop of Cryptographic Hardware and Embedded Systems (CHES), which is the flagship conference in cryptographic engineering.

- CHES 2015, Chakraboti et al. [32], “TriviA: A Fast and Secure Authenticated Encryption Scheme”, this paper first indicates a fast design for performance-constrained applications, however, it becomes clear in the results section that it is not actually fast but the most area-efficient design to date.

- CHES 2015, Liu et al. [33], “Efficient Ring-LWE Encryption on 8-bit AVR Processors”, the authors in fact propose high-speed (HS) and memory-efficient (ME) implementations. However, there is no analysis on efficiency and furthermore, the ME implementation is less memory-efficient than the HS implementation. Obviously, ME was targeted for memory-constrained not memory-efficient systems.

- CHES 2014, Roy et al. [34], “Compact Ring-LWE Cryptoprocessor”, the authors suggest a compact design for resource-constrained domains. However, the results show that it is more than 4× bigger with respect to the previous work. Therefore, the design was not intended for area-optimizations but instead for area-efficiency.

The list goes on. It is not a coincidence that all these papers target recently developed cryptographic constructions. This motivates our application domain even further.
The scope of this work is resource-constrained and resource-efficient designs. Since we now distinguished these two different cases, we can introduce the recurring optimization strategies behind them. Then, we demonstrate how to apply those strategies on real-world systems. As the driving application domain, we will focus on modern cryptosystems. This domain is especially motivating because its design space has not been extensively studied. We show that by clarifying the application context, and then by using the appropriate strategies, we can push the envelope on what is perceived as achievable, by up to two orders-of-magnitude.

1.4 Three Optimization Strategies for Low-cost Platforms

We have observed three recurring optimization strategies to design for low-cost platforms: serialization, precomputation, and integration with reuse. We argue the use of serialization to address constraints, precomputation to provide efficiency, and integration with reuse, to meet both constraints and efficiency requirements.

It is important to highlight these strategies because they extend the shelf life of this work. The cryptographic schemes we focus in this dissertation are very young and hence they are subject to change. However, as we also show throughout this dissertation, we can apply the same strategies on the future iterations of the target algorithms and still reasonably expect a similar optimization level. In this section, we describe these strategies and highlight what makes them appealing for modern cryptographic constructions.
1.4.1 Serialization

Serialization is a generic strategy that reuses the computing resources by time multiplexing the data. Figure 1.3(a) shows a basic visualization of serialization, and its impact on area and execution time. Before serialization, it takes $A_1$ area and $t_1$ time to compute an operation. After serializing, the area reduces to $A_2$ and the execution time increases to $t_2$. The green and blue rectangles in the figure reflects the area-efficiency (smaller is better), which corresponds to Execution Time $\times$ Area. Serialization changes the area-efficiency from $A_1 t_1$ to $A_2 t_2$.

Figure 1.3(b) demonstrates a simple example of serialization. Both of these architectures compute $e = a + b + c + d$. The parallel design on the left uses an adder tree with 3 adders and stores the result in a register. This circuit will compute the value of $e$ in a single clock cycle. The serialized architecture on the right achieves a smaller area by using 1 adder, but it is slower. It requires adding $a, b, c, d$ sequentially at one value per clock cycle (aka, time multiplexing). Therefore, it will take a total of 4 clock cycles to compute the value of $e$. The area of the serialized architecture is slightly more than 1/3 (due to added multiplexers and control) of the parallel version but the execution time is 4× longer. Hence, area-efficiency of the parallel architecture is $3A \times t$, which is better than the efficiency of serialized architecture of $A \times 4t$. Note that, the serialized architecture can be optimized to compute $e$ in 3 clock cycles but the area will still be more than 1/3 of the parallel one, due to added control logic and multiplexers. Therefore, it will still be less area-efficient than the parallel architecture.

As we can infer from the example, serialization requires:

- Serializing the input and, in most cases, also the output. This is also suitable
Figure 1.3: (a) Principle, (b) example of serialization

for serial Input/Output (I/O) interfaces like RS-232.
- Memory elements to move data serially within the architecture.
- Serialized control for time-multiplexing.

What makes Serialization Suitable for Cryptography?

The key of effectively applying serialization is to realize its limit, which boils down to the utilized non-linear operator. Cryptographic constructions use non-linearity to
decorrelate the output from the input. For example, Feistel ciphers use non-linear functions, SP-networks use Substitution boxes (S-box) and public-key schemes use modular, elliptic, or polynomial multiplication. Serializing beyond the non-linear operator backfires with a diminishing return. Serialization is not free, it comes with the extra control overhead (sel control in Figure 1.3(b)). Reducing the complexity of non-linear operation is one of the main motivation behind lightweight cryptography. Advanced Encryption Standard (AES) uses S-boxes and each S-box implements a non-linear operation that maps an 8-bit input to an 8-bit output. Therefore, the smallest AES designs use 8-bit architectures [35]. The major area advantage of the PRESENT lightweight block cipher was proposed as its 4-bit S-boxes which reduces the limit of serialization from 8-bits to 4-bits. But why should we stop there? A recent block cipher does not even use an S-box, instead it simply uses AND gates to provide non-linearity. This makes it an excellent case study. We show that this block cipher, SIMON [36], can be very effectively serialized down to 1-bit level, which is referred to as bit-serialization [37]. Section 2.5 elaborates on the bit-serial design and section 2.6 discusses fine-tuning of bit-serialization on the SIMON block cipher.

1.4.2 Precomputation

We adopt precomputation as a strategy to reduce the amount of computations required in response to the application inputs, by partitioning the underlying operations in an offline part, computed before the inputs are available, and an online part, computed in response to the actual input. This strategy is suitable for real-time embedded systems instead of batch processing databases. We quantify that precomputation can bring efficiency by saving run-time latency and energy requirements of modern cryptosystems at the expense of reasonable extra storage.
Figure 1.4(a) demonstrates the principle of precomputation, and its impact on execution time and energy. Without precomputation, the system performs all operations on-the-fly upon observing the input and generates the output in $t$ time by using $\epsilon$ energy. In contrast, the precomputed execution has an offline part that produces a set of input-independent variables called coupons and stores them in a memory.

Figure 1.4: (a) Principle, (b) example of precomputation
With the real-time input, the system transitions to the online part and generates the output using the precomputed coupon and the input. This transformation will save the amount of run-time computations by moving the input-independent variable computations to the offline part. Indeed, the total execution time and energy of precomputing coupons ($t_{pre}, \epsilon_{pre}$), storing coupons ($t_{str}, \epsilon_{str}$), and later spending them with the input to generate the output ($t_{on}, \epsilon_{on}$), can be more than computing everything on-the-fly ($t, \epsilon$). However, the run-time latency and energy consumption ($t_{on}, \epsilon_{on}$) will be smaller.

Figure 1.4(b) gives a simple example of precomputation. Again, both architectures compute $e = a + b + c + d$. This time, let us assume that for the target application, inputs $a$, $b$, and $c$ do not change often, but the value of $d$ changes for every execution. The circuit on the left uses an adder tree with 3 adders and stores the result in a register. This circuit will evaluate the function by using all four input values $a$, $b$, $c$, and $d$, and compute the value of $e$ in a single clock cycle. To generate $e$, the value at $d$, which changes for every evaluation, has to propagate through two adders.

The architecture on the right uses precomputation for this application. The precomputed coupon in this case is the sum $a+b+c$ which is stored in a register. With the real-time input $d$, the circuit reads the coupon and computes the output $e$. Now, to generate $e$, the value at $d$ has to propagate through a single adder. Therefore, the latency and the run-time energy will be lower than the circuit on the left. Although the two architectures use the same number of computation elements (ie. adders), the precomputation will improve the performance of the implementation.

For simplicity, in this example, we assumed that there is only one precomputed coupon. In reality, the circuit can compute several coupons and store it in a larger
memory element. The extra memory of the precomputed execution depends on the application. For example, if the values of $a$, $b$, $c$ change for every 10 iteration and the system on average performs 100 iterations in a single day, at the end of the day, it can precompute and store 10 coupons to be executed the next day. The more memory results in more coupons and less chance to deplete all available coupons. The designer also has to define what the system does in case of a coupon depletion: either to switch to an on-the-fly computation mode, or to return back to the offline part to generate more coupons. Section 5.2 elaborates and formulates precomputation in the context of energy harvesting systems.

As we can deduce from the example, precomputation requires:

- Analyzing the input and output characteristics of the system.
- Partitioning the operations into an offline part and an online part.
- Determining the amount of required extra memory storage and the feasibility of providing it.

What makes Precomputation Suitable for Cryptography

To effectively apply precomputation, a designer has to analyze the workload that can be handled before the real-time input and minimize the operations that have immediate dependency on it. In cryptographic applications, the run-time operation is typically an input information that is to be encrypted or signed. Precomputed operations, depending on the application, can be generating keys, accumulating an entropy pool, selecting random numbers, and computing message independent variables. Among these, processing key-related material is particularly important because a majority of cryptographic functions start from a secret key and prepares a processed key material which does not depend on the input message. Indeed, block
ciphers use round keys, stream ciphers create key streams and public-key ciphers generate key pairs.

A contribution of this dissertation is to demonstrate that post-quantum cryptosystems can efficiently utilize precomputation. In particular, we have applied precomputation on the two most promising family of post-quantum schemes: lattice-based encryption and hash-based encryption. Hash-based ciphers use precomputable hash-chains based on the processed key material. Moreover, these random keys are typically one-time (ie. they have to change for every new message), making hash-based ciphers inherently suitable for precomputation. Likewise, lattice-based ciphers use random masking polynomials derived from the secret keys to encrypt or sign input messages.

Section 5.3 demonstrates precomputation methods to improve the energy-efficiency of hash-based signatures on an energy-harvesting microcontroller platform. Section 6.4 proposes a precomputed partition of lattice-based signature operations into offline/online part and a smart allocation for an area-efficient HW/SW co-design.

1.4.3 Integration with Reuse

We adopt integration with reuse as a strategy to design optimized systems starting from standalone components, by finding the common elements among various components and combining them into a single optimized unit. Figure 1.5(a) describes the basic principle of integration with reuse. Suppose that we are designing a system that has to implement two operations $A$ and $B$, and the optimization goal is to minimize the area. If we pick the smallest possible architecture $A_1$ and $B_1$ that respectively implements $A$ and $B$, it is not guaranteed to achieve the smallest solution. Indeed,
if $A_1$ and $B_1$ do not share any operations, then there can be two solutions $A_2$ and $B_2$ that are bigger than $A_1$ and $B_1$, but can reuse some area so that their combination is smaller than of $A_1$ and $B_1$. Hence, the components of the combined system may be less optimized than the standalone modules but the integration with reuse could result in a design that is a global minimum. Thus, the result would be better than the trivial combination of locally optimized composing elements.

Figure 1.5(b) exemplifies the advantages of integration with reuse. This time, assume that our system has to compute $f = (a - b) + (c - d)$ in addition to $e = a + b + c + d$. The design on the left uses two separate architectures each implementing one of these two operations. Hence, the design has a total of 4 adders and 2 subtractors, each optimized for a single use-case. The integrated design on the right merges these two operations with reuse. The adders will now implement both addition and subtraction with 2’s complement. This design will require additional multiplexers, inverters and carry-in logic for the adders making it less optimized for computing just $e$ or just $f$, but for computing both $e$ and $f$, it will be smaller than the design that handles the two operations separately.

As we can conclude from the example, integration with reuse requires:

- Enumerating a breakdown of system operations.
- Analyzing the resource sharing opportunities of the composing elements.
- Validating that the resulting architecture is smaller than the sum of orthogonal, standalone optimized blocks.
What makes Integration Appealing for Cryptosystems?

A common pitfall of hardware research in cryptographic engineering is to focus only on the standalone isolated building blocks. However, modern cryptosystems may require various types of arithmetic, cryptographic, and encoding operations. One of the
major contribution of this dissertation is to draw attention to the cross-component optimizations of modern cryptographic systems by applying integration with reuse.

More recently, cryptographic engineering community have started to realize the importance of compact system designs for low-cost applications. Hutter et al. recently published a compact Application Specific Instruction-set Processor (ASIP) design for end-to-end public-key authenticated encryption [30]. Even though there exist smaller alternatives (e.g. AES-GCM), the design uses relatively larger Salsa20 and Poly1305 cryptographic blocks for authenticated encryption because their composing elements also enables public-key encryption operations.

We demonstrate that serialization coupled with integration with reuse enables very compact modern cryptosystems. While serialization reduces all operations into a linear feedback shift register (LFSR) or a non-linear feedback shift register (NLFSR), integration allows sharing those shift registers and common logic operations. For instance, new authentication solutions with PUFs need error correction and encryption. We show that (in Section 3.3) a serialized BCH-based error correction, which is essentially an LFSR, can be integrated very efficiently with a bit-serialized block cipher which is an NLFSR. Another integration with reuse that we applied in the same section is to bring multi-purpose crypto functionality required for authentication protocols like encryption, hashing, and pseudo-random number generation (PRNG) by integrating different mode-of-operations with a single block cipher core. These cryptosystem are significantly more optimized than previous best in literature and they validate the advantages of designing cryptosystems by integration with reuse.
Table 1.1: Organization of the dissertation

<table>
<thead>
<tr>
<th>Reference</th>
<th>Design Target</th>
<th>Used Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2</td>
<td>Resource-constraints</td>
<td>Serialization</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>Resource-constraints</td>
<td>Serialization, Integration</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Resource-constraints</td>
<td>Integration</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Resource-efficiency</td>
<td>Precomputation</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Resource-efficiency</td>
<td>Precomputation, Integration</td>
</tr>
</tbody>
</table>

1.5 Contributions / Organization

Table 1.1 shows the organization of the dissertation. The rest of the dissertation is organized in two parts. The first part focuses on resource-constrained cryptosystem design, and the second part studies resource-efficient cryptosystem design. Each chapter shows a particular illustration of resource-constrained or resource-efficient design using at least one of the three recurring strategies. Chapter 2, 3, and 4 comprise the first part, and Chapter 5, and 6 form the second part. Then, Chapter 7 concludes the dissertation and envisages the future extensions of this work.

The driving application of Part I is PUF-based symmetric-key cryptosystems.

Chapter 2 studies the fundamental building block of symmetric-key cryptosystems: block ciphers. We present the design of the smallest block cipher in the literature, which was approximately 3× smaller than its nearest competitor at the time of publication. To achieve this architecture systematically, we first define the design space of block ciphers. Then, we employ a recent promising block cipher and
apply serialization to minimize the area. We also provide insights towards future lightweight proposals and discuss its possible limitations from an implementation perspective.

Chapter 3 extends our lightweight block cipher design into a compact ASIP that can implement lightweight PUF-based authentication protocols. Again, by employing serialization, we first implement compact error coding and other arithmetic operations. Then, by integrating with reuse, we minimize the area-cost. The integration of cryptographic operations like hashing, encryption, and pseudo-random number generation is based on using the serialized cipher design through lightweight modes-of-operations. The ASIP design also allows mapping multiple theoretical protocol constructions into an optimized architecture and to get real-world performance figures. We demonstrate the capability of the processor by implementing the secure PUF protocols in the literature, and benchmark their execution time, memory footprint, communication overhead, and power/energy consumption. We quantify that the proposed ASIP is smaller than all previous solutions.

Chapter 4 proposes a compact fusion technique that integrates multiple PUF components within a complex system into a single, unified device identity. We investigate new capabilities of the adversary under our fusion scenario and we formulate the conditions to achieve such fusion without degrading the security level. Our method can reduce the error coding complexity and the cost of interfacing multiple PUF components into an authentication protocol.

The driving application of Part II is post-quantum public-key cryptosystems.

Chapter 5 and Chapter 6 investigate precomputation strategies for energy-efficient and area-efficient post-quantum cryptosystems, respectively. We use pre-
and we show that post-quantum public-key cryptosystems favor pre-computation with a smart partitioning. We optimize full signature schemes of the two most promising family of post-quantum systems: lattice- and hash-based signatures. Chapter 5 demonstrates energy-efficient software realizations that can save over an order-of-magnitude run-time energy to provide the same throughput. The software implementation is on an energy-harvested low-cost microcontroller and it introduces an alternative way of thinking about optimizations in an energy-harvested system.

Chapter 6 describes an area-efficient hardware/software codesigned cryptosystem that is up to two orders-of-magnitude faster than the previous work on similar platforms. After partitioning the operations for pre-computation, we employ a hardware/software codesign that allocates complex precomputation operations into a compact microprocessor and that accelerates timing-critical operations with dedicated hardware. We showed that such a hardware/software codesign technique results in area-efficient designs because it can implement several offline operations by integration with reuse through a compact microprocessor architecture and it frees the remaining resources for run-time optimizations.

Chapter 7 revisits the key contributions, summarizes the optimizations and its impact, comments on possible future extensions of our work, and concludes the dissertation.
Part I

Resource-constrained Cryptosystem Design
Chapter 2

Lightweight Encryption Core

While AES is extensively in use in a number of applications, its area cost limits its deployment in resource-constrained platforms. In this chapter, we unveil the design secrets of the smallest block cipher ever proposed in the literature of cryptographic engineering. The design is based on SIMON, a recent promising low-cost alternative of AES on reconfigurable platforms. The Feistel network, the construction of the round function and the key generation of SIMON, enables bit-serial hardware architectures which can significantly reduce the cost. Moreover, encryption and decryption can be done using the same hardware. The results show that with an equivalent security level, SIMON is 86% smaller than AES, 70% smaller than PRESENT (a standardized lightweight AES alternative), and its smallest hardware architecture only costs 36 slices (72 Look-up-tables (LUTs), 30 registers). To our best knowledge, this work sets the new area records as we propose the hardware architecture of the smallest block cipher ever published on Field Programmable Gate Arrays (FPGAs) at 128-bit level of security.
2.1 Introduction

FPGAs bring together the power and area-cost advantages of hardware [38] with the flexibility of software [39], making them very suitable for resource-constrained applications like RFID tags [40]. Apart from security related logic, these nodes should include various other IP cores to implement multiple functionalities. Communicating with RF front-end, receiving and processing data of multiple sensors, implementing access control, managing power supply and many more extensions could be imposed on the target device. Therefore, our aim is to minimize the cost of security on these type of platforms, to enable the implementation of all required functionalities within a single low-cost FPGA.

To implement security in such embedded systems, we rely heavily on block ciphers and utilize them in security protocols [41] to store critical information [42], to authenticate identities [43], even to generate pseudo-random numbers [44]. The most popular block cipher is Rijndael which serves as the AES. AES is extensively used and it constitutes the core of many security systems.

Unfortunately, the cost of AES limits its deployment for resource-constrained embedded system platforms. To overcome this disadvantage, several different lightweight block ciphers have been proposed in the recent years. These block ciphers try to reduce the area-cost of AES. PRESENT [45], XTEA [46], ICEBERG [47], SEA [48], HIGHT [45] and CLEFIA [49] are among the implementations on FPGA that aim to achieve a lightweight block cipher.

Recently, National Security Agency (NSA) published two families of block ciphers, SIMON and SPECK, that can be very promising alternatives to AES for low-cost applications [36]. SIMON is tailored for efficient hardware implementa-
tions and SPECK is tuned for efficient software realizations. The agency argues that their Application Specific Integrated Circuit (ASIC) implementation of SIMON is the smallest available. Moreover, the authors also report that they have used a bit-serialized implementation, but there are no further details about the hardware architecture, other than the implementation results. To find out the FPGA resource utilization, we have implemented SIMON, the hardware optimized block cipher, and we also give details of our low-cost bit-serialized hardware architecture. Our results revealed that SIMON can break area records for block ciphers on FPGAs. We also show that our implementation is even smaller than previously published stream cipher implementations.

To put the area cost in perspective, the smallest Spartan-3 FPGA has 768 slices while the smallest AES design costs 264 equivalent slices (124 slices and 2 Block Random Access Memories (BRAMs)) [35] which constitutes to 34\% of the FPGA area. That leaves the remaining 66\% of available resources to be utilized for other purposes. Our SIMON block cipher implementation provides the same level of security as AES, but only uses 36 slices, which constitutes 4.7\% of the total area.

The rest of the chapter is organized as follows. Section 2.2 discusses the design fundamentals of block ciphers. Section 2.3 gives a brief overview of SIMON. Section 2.4 explores the dimensions of parallelism for block ciphers. Section 2.5 describes the principles of bit-serialization. Section 2.6 specifies the details of the low-cost bit-serialized hardware architecture. Section 2.7 shows the implementation results and its comparison to previous work. Section 2.8 summarizes the lessons learned from this study and gives implementation insights for future block ciphers. Section 2.9 concludes the chapter.
2.2 Basics of Symmetric-key Block Ciphers

Symmetric-key block ciphers perform a series of deterministic operations on a block of input plaintext using the key. This process results in a block of ciphertext. Formula 2.1 shows how to generate this ciphertext, where $x$ is the input $n$-bit plaintext, $c$ is the output $n$-bit ciphertext, $F$ is the round function, $k_i$ ($i \in 1, 2, ..., m$) is the round key derived from the input key for the round $i$, and $m$ is the number of total rounds. Block ciphers essentially apply the round function $F$, $m$-times recursively, using the key $k_i$ that we have derived (recursively) for that round. The first round of this recursion is set to use the input plaintext $x$ with the key $k_1$.

$$c = F(F(...F(F(x, k_1), k_2), k_{m-1}), k_m) \quad (2.1)$$

Depending on the construction of the round function $F$, and the recursion, block ciphers are divided into two groups: Substitute-Permutation network (SPN) and Feistel based block ciphers.

2.2.1 SPN Block Ciphers

SPN block ciphers perform a series of substitution operation followed by a permutation operation on the input plaintext. Figure 2.1(a) shows a generic SPN cipher. Substitute is an operation where $n$-bit input is transformed into $n$-bit output. S-Box performs this operation. S-boxes are designed to have non-linearity and there is a significant amount of research conducted to develop secure S-boxes. Permutation is a simple rewiring operation in hardware where the order of the bits are mixed to
Figure 2.1: The architecture of (a) SPN and (b) Feistel Network.

obfuscate the message further. AES, the current encryption standard, is an SPN cipher.
2.2.2 Feistel Block Ciphers

The most famous example of Feistel is the legacy cipher of Data Encryption Standard (DES), the predecessor of AES, developed by International Business Machines (IBM) [50]. Figure 2.1(b) shows a generic Feistel block cipher. Feistel ciphers divide the input plaintext into two blocks. Then, the cipher processes one of these blocks and the output is evaluated with the other block with an exclusive or (XOR) operation. This operation repeats \( m \)-times where \( m \) is the number of rounds. Feistel round function \( F \) can be defined with simple shift operators and logic gates.

The key difference between Feistel based and SPN cipher is the size of the processed plaintext block within a round. Feistel ciphers typically operate on one half of the block at each round whereas SPN ciphers process the complete block. Therefore, Feistel ciphers usually require more rounds than SPN ciphers to compute the same amount of output bits with the same security level. The advantage of Feistel ciphers is that the non-linear operation can be simpler and the decryption can be done using the same hardware by only reversing the key schedule.

In Section 2.8, we will elaborate more on the low-cost FPGA implementation suitability of Feistel vs. SPN ciphers.

2.3 SIMON: A Feistel Based Block Cipher

In this work, we will focus on the 128/128 configuration of SIMON which has a security level equivalent to AES-128. This configuration uses 128-bit input plaintext and 128-bit key to generate 128-bit ciphertext in 68 rounds. The readers can refer to [36] for further details of this algorithm.
Figure 2.2: (a) Feistel round and (b) key generation of SIMON for the 128/128 configuration.

Figure 2.2 (a) shows the round operation of SIMON. The round function performs logic operations on the most significant 64-bits (the upper half block) and the result is XOR-ed with the least significant 64-bits (the lower half block) and the 64-bit round key $k_i$. At the end of each round, the contents of the upper block is transferred to the lower block as the new generated values are written back into the upper block. The full SIMON round operation consists of three 64-bit circular shift operators (circular left shift one, circular left shift two, and circular left shift eight), three 64-bit XOR operators, and one 64-bit AND operator.

Figure 2.2 (b) shows the key generation of SIMON. The key generation function performs logic operations on the most significant 64-bits and the result is XOR-ed with the least significant 64-bits and the 64-bit round constant $z_i$. The round constant is a design-time constant value that is uniquely tuned for each configuration. The full key generation consist of two 64-bit circular shift operators (circular right shift three and circular right shift four) and three 64-bit XOR operators.
2.4 Dimensions of Parallelism for Block Ciphers

In the design space of block ciphers, several parallelism dimensions exist and a designer can opt for different parallelism choices within each dimension. Figure 2.3 illustrates a 3-d representation of these dimensions where x, y, and z axis respectively stands for the dimensions of parallelism for rounds, encryptions, and bits. We also show several example designs from the literature and this work. In the first dimension, shown in z axis, while computing a round operation, inputs of the operators of the round function can be at any length between one-bit to n-bits, where n is the block-length processed in one round (e.g., SIMON 128/128 has block length of n = 64 while AES-128 has a block length of n = 128). If the operands are one-bit, we call
the resulting implementation a bit-serialized architecture. In the second dimension, shown in $x$ axis, we can parallelize the number of round operations. This number can range from one round to $m$-rounds, where $m$ is the total number of rounds required to complete the block cipher (e.g., SIMON 128/128 has $m = 68$ rounds). In the third dimension, shown in $y$ axis, we can have multiple copies of the encryption engine. Depending on our target throughput, we can use one copy to $e$-copies of the encryption engine where $e$ is the maximum number of copies we can fit into our FPGA.

Depending on the parallelism choices at each dimension, the processing capability of the hardware implementation can range from $e$-parallel encryptions per clock cycle to one-bit of one round of one encryption per clock cycle. We will give brief descriptions of these choices and their effect on throughput and cost, but for the scope of this chapter, we only will investigate the hardware architecture for bit-serialized implementations because it is the most suitable option for area-constrained applications.
Figure 2.4: Generic block diagram for the parallelism of encryptions and rounds
2.4.1 Parallelism of Encryptions

Parallelizing the encryption engine is useful, if there is enough space on the FPGA and the target throughput is not met. Henzen et al. uses 4 AES engines in parallel in AES-Galois Counter Mode (GCM) to achieve the target throughput of 100 Giga bits per second (Gbps) [51]. These architectures are suitable for performance-constrained applications like quantum-key distribution to reach the utmost performance where the cost is not the primary optimization factor. Figure 2.4 shows the block diagram of such hardware architectures that consists of $e$ parallel engines. If the throughput of one engine is $t$ encryptions per clock cycle, then the throughput of $e$ parallel engines is $t \times e$ encryptions per clock cycle.

2.4.2 Parallelism of Rounds

Within an encryption engine, the parallelism choice ranges from a single round operation per clock cycle, up to $m$-rounds in parallel. Figure 2.4 shows the block diagram of such hardware architectures that consist of $r$ parallel rounds. If we select to implement more than a single round in parallel, we first unroll the round functions and associated key generations, then we map each unrolled round operation (together with the corresponding key generation) to a hardware block, pipeline all blocks and set them to run at the same time. Therefore, at each pipeline stage, we can encrypt an independent message block. If we use $r$-parallel blocks of round operation and if the corresponding key generation and the cipher requires $m$-rounds ($r \leq m$), these implementations can achieve a throughput of $\frac{r}{m}$ ciphertext per clock cycle. Using such an architecture with $m$ parallel round operations, we can achieve a throughput of one ciphertext per clock cycle per engine. Qu et al. shows that unrolling all round
operations of the AES hardware could increase the throughput to 73.7Gbps, while using 22994 slices on a Virtex-5 FPGA [52].

2.4.3 Parallelism of Bits

Within a round operation, inputs of an operator can be one-bit up to \( n \)-bits where \( n \) is the block size. Using these operators, if we can process \( b \)-bits in one clock cycle, the computation of one round takes \( \frac{n}{b} \)-cycles. The parallelism choice of bits can have a significant impact on the cost of the resulting architecture.

Resource-constrained hardware architectures are the topic of interest of Part II. These architectures are used in applications where the performance is of secondary importance. Therefore, the challenge is being able to implement the functionality of the block cipher by using the least possible amount of resources. In order to minimize the cost, we used a bit-serialized architecture in which all operators are defined in one bit. To implement SIMON in a bit-serialized fashion, we have to first bit-serialize the round function and key generation. Next, we discuss the bit-serial design strategy on FPGAs and bit-serialization of SIMON.

2.5 Bit-serial Design for FPGAs

As we discussed in Section 1.4.1, the strategy to systematically reduce the area of a circuit is through serialization; dividing operations in time and reusing the same resources for similar computations. In our design, we have applied bit-serialization [37], a serialization strategy that processes one output-bit at a time. We have adapted and applied this strategy with an architecture optimization using shift register logic.
2.5.1 Datapath

Figure 2.5 illustrates an example where the datapath computes $c = a \oplus b$ by XORing two 16-bit registers $a$ and $b$, and generates the 16-bit output $c$. In this example, the datapath uses the same value of $a$ multiple times while the value of $b$ changes. If all the bits are processed in the same clock cycle (Figure 2.5(a)), the datapath produces all bits of $c$ in parallel. This datapath utilizes 48 registers (to store $a$, $b$, and $c$) and 16 LUTs (to compute 16 XOR operations of $c = a \oplus b$). We can map these elements into 24 Spartan-3 FPGA slices [53].

If we bit-serialize the entire datapath (Figure 2.5(b)), the resulting hardware will produce one output bit in one clock cycle. The 16-bit register blocks can now be mapped to SRL-16 logic and the output of $a$ and $b$ can be XORed using a single XOR gate. To keep the value of $a$, SRL-16 $a$ should have a feedback from its output.
Figure 2.6: (a) Control with up-counters, (b) control with ring counters, and (c) control of nested loops

to input. Thus, the resulting hardware architecture will consist of 5 LUTs (3 SRL-16 LUTs to store \(a, b,\) and \(c\), 1 LUT to compute the XOR operation and 1 LUT to apply the feedback via a multiplexer). Now, the datapath can be mapped to a total of 3 slices, which is one-eight of the size of the size of the bit-parallel implementation.

2.5.2 Control

Bit-serialization comes with control overhead. If not dealt carefully, this can counteract the area gain of the datapath. In bit-serial designs, to identify when to start and end loading shift registers, and when to finish operations, we need to keep track of the bit positions during computations. In the example, since the value of \(a\) is fixed for a number of \(c = a \oplus b\) executions, the control needs to determine the value of the select signal at the input multiplexer of SRL-16.\(a\). It will select ‘0’ while \(a_{in}\) is
loaded, otherwise it will select ‘1’. Usually, this is implemented with counters and comparators. Figure 2.6 (a) shows a 4-bit counter with a corresponding comparator. In each clock cycle, the counter value increments by one and four registers update their values in parallel. A comparator checks the counter value and returns ‘1’ when the check condition occurs. This architecture consists of 5 LUTs (4 LUTs for counter and 1 LUT for comparator) and 4 registers.

Instead of using an up-counter, the same functionality can be realized using a ring counter. Ring counters consist of circular shift registers. Figure 2.6 (b) shows a 16-bit ring counter. After 16 clock cycles, the output of this counter will return 1 indicating that 16 cycles have passed. The control unit can use a single LUT (SRL-16) to implement the ring counter which is less than one-fifth of a counter-based control mechanism. If the control signal has to remain 1 after 16 clock cycles, the controller can use an edge detector which costs an extra LUT and register, to check when a transition from 1 to 0 occurs.

Managing the hierarchy of control is also simpler using ring counters and edge detectors. Consider an example with two nested loops both counting up to 16. Figure 2.6 (c) shows the implementation of this nested loop with two ring counters and an edge detector. The outer (SRL-16outer) loop may count the number of rounds while the inner (SRL-16inner) loop counts the number of bits. The Edge Detector will convert the start pulse into a continuous enable signal which will keep SRL-16inner active until a positive edge is detected at the output of SRL-16outer. Once the SRL-16inner is active, its output will be 1 every 16 clock cycles and enable the SRL-16outer for a single clock cycle. This control unit can be realized with 4 LUTs and 3 register (2 LUTs for SRL-16, 2 LUT and 3 register for the Edge Detector).
2.5.3 Bit-serializing SIMON

The datapath of SIMON is serialized similar to the example. The bit-parallel operations are converted into bit-serial ones and the necessary data elements are stored in SRL-16. The serialization of the control flow is achieved by using ring counters and edge detectors. The ring counters control the internal signals when there is a data transmission with the host system. The I/O structure is also simplified using bit-serial design strategy. The data input and output of the design are single bit ports which makes it very suitable for standard serial communication interfaces.

Figure 2.7 (a) shows the details for bit-serialization of the SIMON round. The bits processed at the first clock cycles are also highlighted. The round operation has two components, compute and transfer. Compute requires reading three single-bits from the upper block, one bit from the lower block, and performing logic operations on them using also a single-key bit. The one-bit output of compute is written into the upper block of the next state. Transfer transmits one bit from the upper block of the current state to the lower block of the next state.

Figure 2.7 (b) shows the details for bit-serialization of the SIMON key generation. The format follows the round function and the transfer component is exactly the same. The difference is that Compute requires reading two single-bits from upper block instead of three, and that this time it is right shifted values.

2.6 Hardware Architecture

As we discussed in Section 2.5, FPGA resources are well suited to bit-serialized implementations. LUTs can be efficiently configured as bit-serial memory elements that
can support these hardware architectures. On Spartan-3 FPGAs we can configure LUTs \(^1\) to act as a 16x1 shift register element (SRL-16). In SIMON architecture, we refer to this structure as a First In First Out (FIFO) element, because it does not allow to read or write into its intermediate nodes. Instead, it is only possible to write into the first node and read out the last node. On Spartan-6 FPGAs, due to the increased size of LUT inputs, we can configure LUTs to act as a 64x1 FIFO. Another advantage of FIFO is the ability to reading from and writing into it at the same clock cycle. Using this property, we overlap compute and transfer operations, and complete a round operation in 64 clock cycles.

\(^1\)These are actually special LUTs that are in SLICE_M type of slices
Figure 2.7: Bit-serialized SIMON (a) round function and (b) key generation. \( k_{ij} \) and \( z_{ij} \) respectively stands for the key bit and round constant bit for the \( i \) – \( th \) round and \( j \) – \( th \) bit of the state.
Figure 2.8: Block diagram of round operation for the bit-serialized architecture.
Figure 2.9: Schedule of the bit-serialized architecture

Figure 2.8 shows the details of our FIFO based hardware architecture for round operation. The shift operator causes a challenge for bit-serialized architectures due to its circular access pattern. For example, bits #56,#62,#63 are required from upper-block to calculate the result of bit #0. However, the result of bit #0 will only be used at the next round because its current value is needed to calculate the value of #1 which will happen in the next clock cycle. Figure 2.9 gives the detailed schedule of the FIFO based round operation. To enable the one bit per clock cycle processing, four input bits of the LUT are read at the same clock cycle. On these bits, LUT performs the logic operations of SIMON round function. The 64-bit upper block is stored in 8 shift registers to enable parallel access, followed by a 56x1 sized FIFO. In order to eliminate the bubbles of scheduling that occur at the beginning of each round, we implement two ping-pong register sets, each consisting of 8 registers. At
even rounds, the output of the LUT is written into the lower shift registers (SRD), and at odd rounds, it is written into the upper shift registers (SRU). Similarly, to enable circular access pattern, at even rounds, the output of the FIFO is written back into the SRU, and at odd rounds, it is written back into the SRD. FIFO\textsubscript{2} input is connected to FIFO\textsubscript{1} output to perform transfer operation.
Figure 2.10: Block diagram of key generation for the bit-serialized architecture
Figure 2.10 highlights the details of our bit-serialized FIFO based hardware architecture for key generation. The implementation of the key generation uses the operating principle of the round function, but, this time we have to use right-shifted values at the inputs of LUT. Therefore, the registers are placed after the FIFO. Instead of the ping-pong register architecture that was utilized at round generation, we used an extra register set since we only have to store the first four output values separately. While the 4 original flip-flops store the least significant 4 bits of the upper word in the first round, the 4 extra registers store them for the rest of the rounds. The operation of these additional registers are controlled using an enable signal. These registers are enabled only at the first four clock cycles of a round. During these clock cycles, the output of the LUT is sent to these additional registers and the output of these registers are sent back into the FIFO input.

2.7 Implementation Results

The proposed hardware architecture is implemented in Verilog Hardware Description Language (HDL). The Verilog HDL Register Transfer Level (RTL) codes are synthesized to the Spartan-6 slx4 FPGA with a speed grade of -3 and the Spartan-3 s50 FPGA with a speed grade of -5, using Xilinx Synthesis Technology (XST). The resulting netlists are placed to the same FPGA using Planahead and routed in Integrated Synthesis Environment (ISE) 14.6.
Figure 2.11: Implementation results of SIMON, and comparison with the previous work on low-cost block cipher and stream cipher implementations

Area Comparison Methodology

For the scope of this work, rather than quantifying the actual silicon area for the proposed designs, we are interested in providing a relative comparison with the related work. We refer the interested readers to the work of Kuon et al. [54] for a detailed analysis on the silicon area equivalent of FPGA-based designs.

We will use the number of occupied slices as the area comparison metric. This choice is driven with practicality concerns, almost all the previous work report the slice usage of their designs. A slice essentially consists of LUTs, registers, and in-
terconnect fabric, and the number of these elements depends on the target FPGA technology. Therefore, to provide a meaningful comparison, it is important to use the FPGA technologies with similar slice infrastructure. If the reported results in previous work do not indicate the slice count but only identify the number of registers and LUTs, we assume that these resources are mapped optimally into slices; the register and LUT resources of slices are ideally placed to minimize the slice count. Our comparison methodology also converts the memory footprint of a design into slices by quantifying its distributed RAM cost (as in Good et al. [35]). For Xilinx Spartan-3 FPGA technology, a LUT can implement a 16-bit distributed RAM, hence a slice with two LUTs corresponds to a 32-bit memory element. Likewise, for Xilinx Spartan-6 FPGA technology, a LUT can implement a 64-bit distributed RAM, hence a slice with four LUTs corresponds to a 256-bit memory element.

Comparison with Previous Work

To make a fair comparison with the earlier work, we target a commonly used FPGA, but we also map our implementation on the lowest-cost Spartan-6 FPGA to show the cost on recent low-cost FPGA platforms. Figure 2.11 gives the implementation results and its comparison to previous work. The proposed bit-serialized implementation is the smallest ever reported in the literature. The area result of the closest block cipher is HIGHT, and it uses 91 slices whereas our implementation costs 36 slices. The proposed hardware architecture is also smaller than all published low-cost stream ciphers [55], [56]. With the same security level of 128-bits, the cost of the smallest stream cipher is GRAIN [56], and it uses 48 slices. The proposed hardware architecture of SIMON is 86% smaller than AES, 60% smaller than the smallest block cipher and even 25% smaller than the closest stream cipher. To our best knowledge,
Table 2.1: SIMON — area vs. throughput

<table>
<thead>
<tr>
<th>Design</th>
<th>Area (Slice)</th>
<th>Max. Frequency (MHz)</th>
<th>Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1,1)</td>
<td>36</td>
<td>136</td>
<td>3.6</td>
</tr>
<tr>
<td>(1,1,128)</td>
<td>399</td>
<td>135</td>
<td>216.8</td>
</tr>
<tr>
<td>(1,2,128)</td>
<td>766</td>
<td>135</td>
<td>392.4</td>
</tr>
</tbody>
</table>

Table 2.1 shows the area/performance trade-off for our design. The systematic exploration of parallelism yields a trade-off in performance of a factor of 109 times for a variation in area of 21 times. These factors are much larger than what can be obtained from typical automatic design space exploration using FPGA tools, and it motivates our approach. This trade-off reveals that a round-serial design is more efficient than a bit-serial design and hence more suitable for resource-efficient systems that can include a larger encryption core. However, recall that the context of Part I is resource-constrained systems, not resource-efficient ones.

### 2.8 Lessons Learned and Looking into the Future

In this section, based on our experience from this study, we give insights towards block cipher design from an implementation perspective.
2.8.1 Feistel Goes Well with FPGAs

Compared to SPN based block ciphers, Feistel networks with simple non-linear operators suit better to area-constrained applications. SPN uses S-boxes and the lowest-cost designs follow an s-bit serial architecture where s is the bit-length of the S-box I/O [35]. Therefore, bit-serial implementations are not feasible for SPN based block ciphers. In contrast, Feistel ciphers with a simple non-linear operator can be bit-serialized. A single AND2 gate provides non-linearity for SIMON, which allows bit-serialization. FPGA LUTs can efficiently realize both FIFO and single-bit output RAM memory elements and therefore FPGAs are compatible with bit-serialized implementations.

2.8.2 Correlate Operations with LUT-input Size

For efficient implementations, it is better if the round function and key generation operations correlate with the LUT-input size. For a single-bit, the round operation of SIMON consists of 3 shift operators, 3 XORs and 1 AND gate, whereas key generation consists of 2 shift operators and 3 XOR gates. Therefore, for a single-bit output, SIMON key generation can be implemented using a 4-input LUT, but the round operation requires a 5-input LUT. Spartan-6 FPGAs can implement both of these operations using a single LUT but the round operation requires 2 LUTs on Spartan-3 FPGAs. This might become problematic and can lead to area-inefficient implementations especially for bit-parallel hardware architectures.
2.8.3 Think About Slices, not Only LUTs or Registers

The ultimate reported area-cost is the number of utilized slices. It is not registers or LUTs. Therefore, during design and implementation, it is important to try to balance the number of occupied register and LUTs. Especially FIFO based implementations can utilize this phenomena. The selection of mapping FIFOs to LUTs or registers depends on the resource utilization balance between these resources. This selection also depends on the FPGA architecture. Spartan-3 slices have two LUTs and two registers, but the Spartan-6 slices contain four LUTs and eight registers favouring register heavy implementations choices.

2.8.4 Optimize Beyond Tool Automations

The ultimate reported area cost is the number of utilized slices! Tool automation tends to minimize LUT count and increase the number of utilized slices. During our experiments, we have observed that even using Xilinx Smartexplorer with an area optimization target, the tool automation follows the same trend. Therefore, custom back-end engineering efforts are required to further reduce the slice count. To achieve the lowest cost, we had to hand-pick our design elements using Xilinx PlanAhead. However, reducing the number of slices might come at the expense of operating clock frequency. Routing might not be optimal between the LUTs that a designer picks, and it might be a very time consuming process to find the optimal clock frequency for the minimum number of utilized slices. Moreover, this optimal clock frequency will likely be smaller than the tool optimized version.
2.8.5 Recent Platforms Significantly Reduce the Total Cost

We generally expect to achieve higher throughputs for the same architecture as we move towards smaller technology nodes. We also expect our circuit area to shrink. Due to the increase in the size of LUT-input from four bits to six bits, the total number of LUTs reduce. Our implementation results showed that when we move from Spartan-3 to Spartan-6 FPGAs, the LUT count reduces from 72 to 40, resulting in an area reduction of 45%. If the input bits of the LUT grow further in future technology nodes, we can also expect to have lower LUT count for the same design.

2.9 Conclusion

We demonstrated that by using bit-serialization on a modern block cipher, we can indeed set the new area records for encryption modules on FPGAs. The total area count of the resulting design is 36 slices on a Spartan-3 FPGA and 13 slices on a Spartan-6 FPGA. SIMON claims to have approximately a 50% of area reduction over AES for its ASIC implementation, we validate that with a reduction of 86%, SIMON is an even stronger alternative to AES for resource-constrained FPGA applications. This encryption core is a new enabler for resource-constrained symmetric-key cryptosystems. The following section takes the next step towards that direction.
Chapter 3

Lightweight ASIP for PUF-based Protocols

So far, the effort in building resource-constrained cryptographic solutions have mainly focused on designing single-purpose modules that can implement a single functionality. In this chapter, we propose a compact ASIP for PUF-based authentication protocols. The ASIP can perform several cryptographic, encoding, and arithmetic operations which enable the implementation of multiple authentication protocols. The key idea of the proposed ASIP is applying serialization and integration with reuse for area-optimization of the combination of its multiple components.

Our contribution is twofold. First, we provide a novel ASIP that can efficiently execute PUF-based authentication protocols. Second, we demonstrate the capability of our ASIP by mapping three secure PUF-based authentication protocols and benchmark their execution time, memory footprint, communication overhead, and power/energy consumption. Our results demonstrate the advantage of ASIP over
dedicated architectures and also as opposed to general-purpose programming on an MSP430. The results further demonstrate various efficiency metrics that can be used to compare PUF-based protocol implementations.

3.1 Introduction

PUFs are a promising technology that can derive entropy from the physical characteristics of an implementation. PUF-based authentication protocols utilize the properties of PUFs and provide novel identification mechanisms. A promising feature of these systems is to provide lightweight root-of-trust for the physical authentication solutions on resource-constrained applications like the IoT [13]. So far, there have been several protocol attempts with PUFs ([57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [5]), yet a recent analysis reveals that only a minority of them are regarded as secure [75]. Hence, the search is far from over and in the near future, we expect to see more contributions to the literature of PUF protocols with improved properties.

The primary objective of this chapter is designing a flexible solution with multiple PUF protocol elements and optimizing the complete system for resource-constrained applications. This is indeed the first attempt towards such a goal. Unfortunately, the current research on PUFs follows two distinct trends, (1) theoretical formulation and security evaluation of PUF-based protocols or (2) practical construction and characterization of single-purpose building blocks. There are a few existing connections between the two concepts that implement a full PUF-based protocol [72], [73], [5]. However, these are ad-hoc designs that can implement a single protocol (which is bound to change often) and they have limited optimizations
towards resource-constrained applications. Moreover, some of their performance results are either missing or vary considerably depending on the specific PUF instance of the proposed design.

We take on the daunting task of designing an ASIP for our goal. We start by observing existing PUF protocols, analyze the security and efficiency of their composing elements, and finally provide an optimized, low-area and low-power processor that can execute multiple lightweight PUF protocols. We design a novel ASIP because it allows utilizing both serialization and integration with reuse; ASIP is a processor customized for a specific application and it offers the flexibility to implement multiple protocols by reusing a specialized datapath. Compared to a general-purpose processor, it can significantly reduce the area-cost and power consumption.

Moreover, our work gives an insight to theory-focused researchers on the building blocks of efficient protocols. Using our ASIP’s API, they can observe the suitability of their protocol and quickly quantify the real-world implementation results that are optimized towards lightweight applications. The practice-oriented researchers can also use our ASIP and plug-in their latest optimized PUF or other component to see its impact on a complete protocol instantiation.

The rest of the chapter is organized as follows. Section 3.2 introduces the PUF technology, investigates PUF protocols and motivates the selected protocols of this work. Section 3.3 describes the design of ASIP: high-level design principles, hardware architecture, and the instruction set of the ASIP. Section 3.4 and Appendix A explains the mapping of the reference protocols onto the ASIP. Section 3.5 presents the implementation results and its comparison to the previous work. Section 3.6 concludes the chapter and comments on possible future extensions.
3.2 PUF-based Protocols for Secure and Lightweight Authentication

In this section, we make a very brief introduction to Silicon PUF Technology to cover the basic principles of PUFs. The next subsection is entitled as Part 1 because later in Chapter 4, we cover more topics on PUF regarding quality metrics for identification. We refer interested readers to [76] for an in-depth discussion of PUFs.

3.2.1 Background on Silicon PUF Technology — Part 1

Process variation is a well-known phenomena that occurs due to variations during the fabrication processes of electronic devices. It affects physical attributes like transistor drive strength and delay characteristics, and PUFs exploit these random variations to generate device-unique digital fingerprints. Just like human fingerprints, since it is hard to exactly model the variations, it is also hard to predict the actual fingerprints before they emerge. And once it does, the entropy becomes static, the variations ensure a certain amount of difference between two distinct fingerprints. This fingerprint information is a type of hardware root-of-trust which can test something that the device physically possess. Therefore, silicon PUF technology has received a significant amount of interest for security-critical applications like Intellectual Property (IP) protection [77], tamper-evident key storage [78], and device counterfeiting [79].

Figure 3.1 describes the basic model of a PUF with Challenge/Response mechanism. The Challenge is a distinct input and a method to trigger the PUF and the Response is the corresponding output. Depending on the obtainable Challenge Response pairs (CRPs), PUFs are categorized into two groups: Weak-PUFs and
Strong-PUFs. Weak-PUFs have a few number of challenge response pairs in a smaller silicon area while the Strong-PUFs offer an exponentially larger CRP space at the expense of an increased area-cost [74]. Therefore, the selection between using Weak-PUFs or Strong-PUFs depends on how many CRPs are sufficient to implement the PUF-based application.

Ideally, we want a PUF to be function; on the same device it should always generate the exact same output for the same input. Ideally, we also want an average hamming distance of 50% between the PUF responses on two different devices. Unfortunately, the ideal PUF behavior is not observed in the real-world. Because of temperature variations, supply voltage changes, and device aging, PUF responses exhibit noise. Moreover, due to systematic bias, PUF responses do not have 100%
entropy. Therefore, several post-processing mechanisms have been proposed to deal with these issues. Handling bias issues are relatively easier, since it is a matter of getting extra response bits to satisfy the target entropy level. However, mitigating noise is more complex, especially for cryptographic protocols, because inputs of a cryptographic operation must be error free. If there is a single-bit error on one of the inputs, it would result in a completely uncorrelated and invalid output.

In this chapter, we are interested in PUF-based authentication protocols. These protocols can provide a physical assurance on the existence of a certain identity derived from PUFs. Since the scope of Part I is resource-constrained devices, we study lightweight PUF-protocols that enable authentication between a resource-constrained device that embodies a PUF, and a resource-rich server. In this scenario, the server registers PUF responses during a secure enrolment phase. Then, for verification, the server tests the knowledge of these responses by using a protocol.

Figure 3.2 demonstrates a simplified block diagram of the typical operations on a constrained device for PUF-based authentication protocols. First, the PUF is triggered with a challenge, either generated by the device or else received from the server. Then, a form of error correction is applied on a noisy response, followed by a randomness extractor to derive the left-over entropy. Finally, this entropy is

---

1Here, for simplicity, we omit the description of more exotic methods like the ones that registers a model of PUF then uses that model to predict CRPs
used by various cryptographic operations depending on the selected protocol. Several messages can be exchanged between the device and the server during this phase until the protocol terminates.

### 3.2.2 Distinguishing Lightweight PUF Protocols

Our analysis revealed and also the final results confirmed that, error-recovery mechanism is the key differentiator of lightweight PUF protocols. After investigating existing protocols, we opt our ASIP to support error-recovery with reverse fuzzy extractors. Although we do not discredit other proposals, we observed two properties of reverse fuzzy extractors that work for our favor: area-cost and security. Next, we describe these essential criteria in detail.

PUFs are noisy components and all practical PUF-based protocols require a method to handle the noise of the PUF core. In light of Delvaux et al. [75], we can categorize secure protocols into three primary groups based on their error-recovery mechanism: (1) fuzzy extractors, (2) reverse fuzzy extractors, (3) pattern matching. Fuzzy extractors are regarded as secure [80] but the error-decoding operations of the fuzzy extractor scheme [81] are very complex which makes it unsuitable for lightweight applications. A more efficient approach, described in Figure 3.3, is to utilize the reverse fuzzy extractors [73] which exchange the complex error-decoding operations of the PUF device with the simpler error-encoding operations of the authentication server. Indeed, Maes et al. show that PUF error-encoding operations require \(2 \times\) less computation area and \(98 \times\) less clock cycles than error-decoding [82], [74]. The protocols that use some form of pattern-matching are regarded as secure albeit they may require additional security countermeasures against helper data manipula-
Figure 3.3: The basic principle of (a) fuzzy extractors and (b) reverse fuzzy extractors. $c$ is the challenge, $r$ is the enrolled response, $r'$ is a noisy response, $h$ is a helper data, Gen is helper data generator (error coding operations), Rec is reconstruction (error decoding operations). At the end of both sessions in (a) and (b), the trustworthy parties should arrive at the same value ($r$ for (a), $r'$ for (b)), if the PUF noise is within the range of the error correction capacity of the system. Then, they can test each others knowledge on this value by several different methods, depending on the authentication protocol. The helper data $h$ leaks information about the PUF response, yet the constructions still assure a certain amount of left-over entropy.
Figure 3.4: Symbolic presentation for the RFE-2 protocol and the AGMSY’15 protocol

Aydin Aysu Chapter 3. Lightweight ASIP

These protocols replace symmetric-key cryptographic operations with supposedly lightweight computations. However, in Section 3.5, we quantify that they (slender PUF successor with security update [83]) occupy significantly more area. We also quantify that our complete engine with a reverse fuzzy extractor is smaller than area-optimized error-decoding hardware modules in PUFKY [82]. Thus, we can confirm the argument that the key derivative of secure and efficient PUF protocols is indeed applying the reverse fuzzy extraction.

Currently, the protocols that utilize the reverse fuzzy extractor construction are the reverse fuzzy extractors (RFE-1) [73], modified reverse fuzzy extractors (RFE-2) [74], and AGMSY’15 [5]. AGMSY’15 is a very recent protocol with a reverse fuzzy extractor that is not extensively evaluated yet, but it includes a formal proof of security that provides a high security confidence.

Figure 3.4 demonstrates the main building blocks of the RFE-2 and the AGMSY’15 authentication protocols. RFE-1 is a slight variation of RFE-2 which do not use a
TRNG. The notation follows Delvaux et al. [75]. Hash is hash operation, Encrypt is a symmetric-key encryption operation, PRF is a pseudo-random function, Gen is a helper data generator, TRNG is a true random number generator, NVM is a non-volatile memory and PUF is a Physical Unclonable Function. The detailed execution of RFE-2 and AGMSY’15 protocols may be consulted in Appendix A, for brevity, we omit the details of RFE-1. The figure shows that although these protocols share some common elements, they differ in the cryptographic primitives and arithmetic operations that they perform. This makes the ASIP design for PUF-protocols a non-trivial task and shows that it requires careful analysis and optimization of the protocol elements.

3.3 The Design of the ASIP

In this section, we will present our ASIP design in a top-down fashion. We will start with the design principles that we derive by analyzing protocols. Then, we will describe the details of the hardware architecture and the instruction set.

3.3.1 Design Principles

ASIP is a processor customized for a specific application. It is a proper choice to implement resource-constrained flexible designs because it offers a mechanism to combine serialization and integration with reuse. First, we can design specialized bit-serial architectures for the execution unit and then we can program the ASIP to reuse it for multiple protocols.

The main challenge in the design and implementation of any ASIP is to pick the
correct set of instructions. The advantage of an ASIP is that it can omit unnecessary generic operations and instead support application-specific irregular operations to improve area-cost. However, especially for compact ASIPs, the designer should carefully consider this selection process as each added instruction increases the area-cost. Therefore, a correct analysis and assessment of the application is essential prior to the design process.

The methodology we used was to first figure out the common operations of PUF protocols and then to pick and optimize them for the lightweight applications. Our ASIP is not generic in terms of executing any kind of protocol that can possibly be formulated using PUFs, and we argue that it requires specialization to achieve a compact architecture. Therefore, we provide a platform to transform and map lightweight protocols. The common elements of these protocols are entropy sources, cryptographic building blocks, error handling, and simple arithmetic and conditional operations [75]. Next, we discuss the composing elements of such protocols and how to provide all these functionalities in a compact manner.

**Entropy:**

A fundamental requirement of PUF-based protocols is entropy. In the context of these protocols, there are two types of entropy: static and dynamic entropy. Static entropy is used to encode the identity information of the parties involved in the protocol. Dynamic entropy is used to support the freshness (random nonces) of the protocol. Each protocol evaluates a PUF at least once for each authentication run, hence the system should have direct access to the PUF core to gather the static entropy. To minimize the area, we follow the previous methods [84], [85] that reuse
the PUF output noise also as the root of dynamic entropy. The ASIP design also includes a low-complexity, XOR-based PUF output post-processing capability which enables integrating PUF cores with different characteristics. For dynamic entropy, it would amplify the noise. For static entropy, it would reduce the systematic bias at the expense of reliability.

**Cryptographic Operations:**

The proposed ASIP supports hashing, encryption, and pseudo-random functions. To achieve a compact architecture, we have extrapolated the bit-serialized design of Chapter 2, which is known to be the smallest for the target technology [9], and which has lightweight extensions [86], [8]. The design uses the SIMON block cipher [36] (described earlier in Section 2) which so far does not reveal weakness against conventional cryptanalytic techniques [87], [88], [89]. Following the typical lightweight cryptography trend, we slightly reduced the security level of the reference design [9] to 96-bits (ECRYPT-II Level 5 or legacy standard-level [90]). Reducing it further to the next SIMON key-length of 72-bits is not desired as the de facto lightweight security is regarded around 80-bits. We then extend the block cipher capability by building modes-of-operations to support the required cryptographic operations. The details of these techniques described later in Section 3.3.2.

**Error Handling:**

Inevitably, PUF-based protocols have to deal with the noise caused by PUFs to provide robust authentication mechanisms. The lightweight protocols solve this problem through reverse fuzzy extractor schemes [73], [74], [91] in which the simpler error
encoding is performed on the constrained device and the complex error decoding operations are outsourced to the resource-rich server side. Typically, BCH codes [92] are used to generate the error code (helper data) due to its good error coding performance [93] and design flexibility. Our ASIP follows the reverseFE construction [73] with BCH codes and offset coding mechanism [81]. The BCH parameters are defined as design-time parameters. Note that the ASIP only supports the reverse fuzzy extractor construction due to its significant advantages over other schemes (see Section 3.2 for the details and the reasoning). Therefore, it omits the execution of other insecure or inefficient protocols such as the ones that uses other error handling schemes or the ones that performs error decoding on the constrained device.

Other Operations:

Most protocols require decision making, which can be supported with conditional jumps in the instruction set. In addition, the ASIP supports arithmetic and standard jump instructions. To hold the persistent state of the schemes, the memory that we have in our ASIP supports constant initialization vectors. Although some protocols use non-volatile memory to store data, we do not include it inside the processor design.

3.3.2 Hardware Architecture

Since we target lightweight applications, our hardware organization is as simple as possible and primarily targets area savings and flexible operations. To reduce the area-cost, the ASIP does not have any kind of data forwarding or caching infrastructure which would complicate the architecture. The ASIP does not utilize internal
Figure 3.5: Block diagram of the ASIP. The grey areas represent the PUF-specific building blocks. Integration of reverse fuzzy extractors and PUF core into the ASIP architecture are unique to PUF-based protocols.

registers to store data and all instructions with the associated data are directly read from and written into the main memory. This choice is motivated further because it does not result in a performance penalty; the design is ported to an FPGA technology in which the clock cycles required to load a data from a register and a BRAM are equal.

Figure 3.5 shows the top-level building blocks of our ASIP. The system uses a single BRAM that is partitioned into two sections (Instruction Memory and Data Memory) which separately store instruction and data elements. The size of the BRAM is 32 Kilobit (Kb) and it is organized as 1024x32 with 1024-address locations that can store up to 32-bits. Thus, our ASIP can support any instruction word up to 32-bits in length. In our current design, we used an instruction word of 26-bits and a data-width of 16-bits, which is discussed further in Section 3.3.3.
The *Instruction Fetch & Decode* unit loads the instructions and depending on the opcode it sets the control signals for all other modules. Then, the *Memory Load* communicates with the BRAM using the decoded addresses and reads the associated data. The execution units of the ASIP are the *Bit-serial Core* and the *PUF Controller*. The *Bit-serial Core* is the crypto-core of the ASIP and it is a bit-serial extension of SIMON with several modes-of-operation at 96-bit security. It also has integrated arithmetic and BCH encoding capabilities. The *PUF Controller* interacts with the PUF and provides static and dynamic entropy to the system. Note that we do not implement a PUF core, building and optimizing a PUF is out of scope of this work. Instead, the ASIP design includes a generic PUF controller that can be interfaced to strong or weak PUFs. Finally, the output of the execution units are transferred to the *Memory Store* module which writes it into the memory using the write address of the instruction.

### 3.3.3 Instruction Set

Figure 3.6 formulates the structure of the ASIP’s instructions. The machine code is composed of 26-bits: 10-bits for read address, 10-bits for write address, and 6-bits for instruction opcodes. The *opcode* is the operational code of the instruction. *Addr#1* and *Addr#2* are the addresses of the read and write operations respectively. For conditional operators and jump instructions, *Addr#2* holds the address offset of the jump destination. For all other executions, the instruction address is automatically incremented by 1 after execution. All instructions have direct memory access and there are no internal registers to store data after an operation. The address values specify the absolute physical location, the ASIP does not use any type of virtual addressing.
Table 3.1 summarizes the instruction set of the ASIP. The 6-bits of opcode space allows the system to define up to 64 distinct instructions. The execution of our assembly is simple and straightforward. For example, the mnemonic `SIMON.HASHF` loads the last block of the hash function, computes the hash, and writes the output back into the main memory. During this execution, `Addr#1` stores the read address of the input and `Addr#2` stores the write address of the output. Therefore, if one uses the symbolic representation `SIMON.HASHF(0,8)`, it will correspond to the hexadecimal machine code ‘0000203’, which is equal to value ‘0’ for `Addr#1`, value ‘8’ for `Addr#2` and to the `SIMON.HASHF` opcode value of ‘3’. Since the read operand size of this instruction is 48-bits, the system will read three address positions starting from ‘0’ (0,1 and 2) and hash it. After the operation is complete, the ASIP will write the resulting 192-bit output into the ‘12’ consecutive address locations of memory, starting from the address position ‘8’ (8,9,...,19).

**ENTRPTY**

The **ENTRPTY** instructions interact with a PUF core to provide dynamic and static entropy. The processor also supports two burst output modes: 96-bits and 384-bits. The PUF output may require successive XORing either to improve its min-entropy/uniqueness or else its randomness. This is enabled in our controller through
Table 3.1: Instruction set of the ASIP

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode (Binary)</th>
<th>Operand Size¹²</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>000000</td>
<td>-/-</td>
<td>Standby state waits for start pulse</td>
</tr>
<tr>
<td>ENTRPY.Stat.96</td>
<td>010001</td>
<td>-/96</td>
<td>PUF static entropy extraction</td>
</tr>
<tr>
<td>ENTRPY.Stat.96x4</td>
<td>010010</td>
<td>-/384</td>
<td>PUF static entropy extraction</td>
</tr>
<tr>
<td>ENTRPY.Dyn.96</td>
<td>010011</td>
<td>-/96</td>
<td>PUF dynamic entropy extraction</td>
</tr>
<tr>
<td>ENTRPY.Dyn.96x4</td>
<td>010100</td>
<td>-/384</td>
<td>PUF dynamic entropy extraction</td>
</tr>
<tr>
<td>BCH.Gen</td>
<td>001001</td>
<td>16/48³</td>
<td>BCH codeword generation</td>
</tr>
<tr>
<td>SIMON.ECB</td>
<td>000100</td>
<td>96/-</td>
<td>Plaintext load for SIMON-ECB execution</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>000110</td>
<td>96/-</td>
<td>Plaintext load for SIMON-CBC execution</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>000101</td>
<td>96/96</td>
<td>Key load for SIMON and start execution</td>
</tr>
<tr>
<td>SIMON.PRFF1</td>
<td>000111</td>
<td>96/-</td>
<td>Plaintext load for SIMON-PRF without XOR</td>
</tr>
<tr>
<td>SIMON.PRFF2</td>
<td>001000</td>
<td>96/-</td>
<td>Plaintext load for SIMON-PRF with XOR</td>
</tr>
<tr>
<td>SIMON.HASHS</td>
<td>000001</td>
<td>48/-</td>
<td>First input block load for hash</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>000010</td>
<td>48/-</td>
<td>Intermediate input block load for hash</td>
</tr>
<tr>
<td>SIMON.HASHF</td>
<td>000011</td>
<td>48/192</td>
<td>Last block load for hash and dump output</td>
</tr>
<tr>
<td>XOR.L1.96</td>
<td>001010</td>
<td>96/-</td>
<td>Operand #1 load for XOR operation</td>
</tr>
<tr>
<td>XOR.L2.96</td>
<td>001011</td>
<td>96/96</td>
<td>Operand #2 load and compute XOR operation</td>
</tr>
<tr>
<td>XOR.L1.64</td>
<td>001100</td>
<td>64/-</td>
<td>Operand #1 load for XOR operation</td>
</tr>
<tr>
<td>XOR.L2.64</td>
<td>001101</td>
<td>64/64</td>
<td>Operand #2 load and compute XOR operation</td>
</tr>
<tr>
<td>CHKJMP.L1</td>
<td>001110</td>
<td>96/-</td>
<td>Operand #1 load for comparison</td>
</tr>
<tr>
<td>CHKJMP.L2</td>
<td>001111</td>
<td>96/-</td>
<td>Operand #2 load and compute comparison</td>
</tr>
<tr>
<td>JMP</td>
<td>010000</td>
<td>-/-</td>
<td>Jump instruction</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>010101</td>
<td>16/16</td>
<td>Copy the memory content to another address</td>
</tr>
<tr>
<td>NOP</td>
<td>010110</td>
<td>-/-</td>
<td>Dummy instruction</td>
</tr>
<tr>
<td>SRAM.Send</td>
<td>010111</td>
<td>-/-</td>
<td>SRAM data transmission</td>
</tr>
<tr>
<td>SRAM.Rec</td>
<td>011000</td>
<td>-/-</td>
<td>SRAM data reception</td>
</tr>
</tbody>
</table>

¹ The symbol − denotes that the instruction do not use the associated field

² The operand size is defined in bits

³ The output of BCH is 47-bits but it is expanded to 48-bits by concatenating a single ‘0’ bit.
two design-time parameters. The designer can individually set these parameters and choose the number of XOR operations that the controller performs for static and dynamic cases.

**BCH**

We implement the encoding $BCH.\text{Gen}$ with parameters $(63,16,11)$. This setting allows to correct up to 11-bits of errors within a 63-bit PUF response. For the PUF error rates 2\%, 3.5\%, 7\%, the probability of failure (the probability of observing more than 11-bits of errors) within a 63-bit PUF response is approximately $10^{-9}$, $10^{-6}$, $10^{-3}$, respectively. These error rates are well within the limits of noise at normal operating conditions [94]. The lower-bound of residual entropy of a single codeword is 16-bits, omitting the systematic bias. The 47-bit parity is concatenated with a single ‘0’ bit and expanded into 48-bits.

**XOR**

There are two types of XOR instructions: $\text{XOR.L1.x}$ loads the first operand and $\text{XOR.L2.x}$ loads the second operand and starts XORing. The XOR operation is defined in 64-bits and 96-bits.

**SIMON**

We will describe the cryptographic operations with SIMON in further details because it represent a good example of integration with reuse strategy.
How can we combine the multi-purpose cryptographic operation requirements of protocols with an area resource-constraint? Figure 3.7 shows the advantage of our proposal compared to the traditional approaches. The straightforward approach is to include optimized single-purpose hardware blocks and to glue them with a finite-state machine wrapper. Clearly, a solution that uses this approach with disjoint kernels (like PRESENT [95] for encryption, PHOTON [96] for hashing, and TRIVIUM [97] for PRNG) yields a design that is larger than the sum of its composing kernels. It also ignores the opportunity to share the internal designs for each kernel. Another solution would be to use embedded software on a compact microcontroller. The program memory of the microcontroller can store an instruction-level description of...
Table 3.2: Integrating with modes-of-operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Kernel and Configuration</th>
<th>Modes-Of-Operation</th>
<th>Security-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption</td>
<td>SIMON 96/96</td>
<td>ECB, CBC</td>
<td>96-bits</td>
</tr>
<tr>
<td>Hash function</td>
<td>SIMON 96/144</td>
<td>Hirose [98]</td>
<td>96-bits(^1)</td>
</tr>
<tr>
<td>PRNG</td>
<td>SIMON 96/96</td>
<td>CTR</td>
<td>96-bits</td>
</tr>
<tr>
<td>PRF</td>
<td>SIMON 96/96</td>
<td>CBC-MAC</td>
<td>96-bits</td>
</tr>
</tbody>
</table>

\(^1\) SIMON 96/144 generates a digest of 192-bits which has 96-bits of strong collision resistance.

each operation, and can configure the small microarchitecture. But such a solution is not ideal either, because the instruction-set of the microcontroller is generic, and not optimized for the multi-purpose kernel which we have in mind. Therefore, we have investigated a third option: the design of a flexible yet specialized crypto-engine.

The hardware architecture implements cryptographic functionalities by reusing SIMON core with several modes-of-operation. The modes provide a security level of 96-bits. Table 3.2 shows the modes-of-operations used to realize the cryptographic operations, the SIMON configuration, and the associated security level. Note that we need to implement two SIMON configurations: SIMON 96/96 and SIMON 96/144. We applied integration with reuse to implement these two configurations. SIMON 96/96 and 96/144 have exactly the same round functions but the key generations are different. Figure 3.8 shows the key generation of SIMON 96/144. The gray colored region is the extra operations of SIMON 96/144, if SIMON 96/96 is selected, we simply forward the data from \(K_i\) to \(K_i+2\), skipping the \(K_{i+1}\). For brevity, we omit the details of hardware micro-architecture of these operations, interested readers can refer to [8] for details.
Figure 3.8: In the bit-serialized architecture \( K_{i+1} \) simply becomes a 48-bit shift register which is skipped for SIMON 96/96

The encryption uses the Electronic Codebook (ECB) and Cipher Block Chaining (CBC) modes-of-operation with the SIMON 96/96 configuration. `SIMON.CBC` and `SIMON.ECB` loads the plaintext of SIMON encryption in CBC and ECB modes of operations, respectively. To realize the hash function, the ASIP design employs the Hirose double-block-length construction [98] which enables building a secure and collision-resistant hash function from block ciphers. The input size of the hash is arbitrary but the SIMON configuration for the Hirose construction is set to 96/144. Therefore, the ASIP can process a single 48-bit input block at a time and ultimately generates 192-bits of hash output (96-bits of strong collision-resistance). A hash operation consists of three sub-steps: `SIMON.HASHS` starts hashing of the first 48-bit input block. `SIMON.HASHR` resumes it for the intermediate input blocks and `SIMON.HASHF` processes the final 48-bit input block of the hash function and produces the 192-bit hash output. If the input is smaller than 96-bits, it has to be expanded into 96-bits in the memory and the `SIMON.HASHR` has to be skipped.

A secure PRF can be constructed by CBC-Message Authentication Code (MAC) and its length extension with counter [20]. To implement this functionality, the ASIP
Figure 3.9: The hardware architecture of the cryptographic operations. The figure demonstrates (a) Hirose and (b) ECB. The Hirose uses both Round Functions and other arithmetic while the active parts of the ECB uses a smaller subset which is depicted in green. The figure is adapted from [99].

has two dedicated instructions: \texttt{SIMON.PRF1} and \texttt{SIMON.PRF2}. \texttt{SIMON.PRF1} performs the initial stage of CBC-MAC (without XOR). \texttt{SIMON.PRF2} iteratively performs the subsequent stages of CBC-MAC with input message. The last step of all SIMON based operations is the \texttt{SIMON.KeyAndGo} instruction that loads the key and starts
the associated process.

Figure 3.9 shows the high-level architecture of the hardware and illustrates the integration with reuse for two modes-of-operations. To implement Hirose, the architecture includes two copies of the block cipher. The SIMON Round Function Master is the full SIMON round function while SIMON Round Function Slave is the SIMON datapath receiving all control signals from the Master. These two blocks execute in parallel to perform the Hirose mode-of-operation and the key generation works for SIMON 96/144. For ECB mode, SIMON Round Function Master is reused while the Slave is inactive. The key generation now works for the SIMON 96/96 configuration.

MEMCPY

MEMCPY instruction copies the content of a memory element from one memory location into another. It is useful to reorganize the memory elements since many operations read from and write into multiple consecutive memory locations.

Static Random Access Memory (SRAM)

The SRAM instruction is used to organize the I/O communication. These instructions work with packets of 96-bits. SRAM.SEND and SRAM.RCV assert separate flags to indicate that the ASIP is ready for an I/O operation. Once these flags are deasserted by the outside module, the ASIP starts sending or receiving the associated data depending on the opcode.
CHECKJMP, JMP, NOP

CHECKJMP and JMP refer to the conditional and unconditional jump instructions, and NOP is a dummy instruction. To implement a conditional jump, one can load the first operand with CHCKJMP.L1 and then load and compare the second operand with CHCKJMP.L2. The ASIP jumps to instruction at Addr#2 of CHCKJMP.L2 if there is a mismatch between these two operands.

IDLE

IDLE instruction is the initial state of the ASIP. During this step, the system waits for a start signal pulse to go to the next instruction and begin processing a protocol run. After a protocol run is complete, the ASIP returns to this step and waits for the next start pulse.

3.3.4 Key Features

The summary of our design’s key features are as follows.

Bit-serialized processing: To minimize the area of the ASIP, our architecture applies the bit-serialization strategy for control and datapath.

Lightweight core: We employ a lightweight block cipher, SIMON [36], and rather than having disjoint kernels for different cryptographic operations, we build everything around SIMON by using several modes-of-operations. These constructions ensure a security level of 96-bits.

Shift register reuse: The shift registers of our bit-serialized SIMON hard-
ware architecture also stores the variables for BCH encoding and arithmetic operations.

**Complex dedicated instructions:** Typically, general-purpose embedded processors execute simple instructions like arithmetic and conditional instruction. In addition to these ones, our ASIP is also able to execute more complex dedicated instructions such as hashing and encoding.

**Variable operand size:** The ASIP is capable of working with various operand sizes. For example, with a single instruction, it can encrypt 96-bit plaintext into 96-bit ciphertext, or implement a 64-bitwise XOR operation, or gather 384-bits of entropy.

**PUF interface:** The PUF interface is primarily for the weak-PUFs, PUFs that have small challenge-response space. Maes shows that weak SRAM PUFs yield the smallest area for lightweight authentication scenarios [74]. However, since the design can generate pseudo-random numbers, it can also generate challenges for a strong-PUF. In addition, the ASIP supports an internal XOR operation which may be used for enhancing min-entropy of PUF responses at the expense of reliability.

**Shared memory model:** Our design can easily be ported into a more complex system-on-chip infrastructure through a shared memory model. The ASIP provides instructions for I/O communications.

### 3.4 Protocol Instantiations on ASIP

To demonstrate the capability of our ASIP, we mapped three available protocols that follow the reverse fuzzy extractors. These are the modified RFE-1 [73], RFE-2 [74],
and the AGMSY’15 [5] protocols. Section 3.2 explains the reasons why we map these three protocols.

The first step of the transformation is to convert the security level of the original schemes into 96-bits since our SIMON core works with a fixed 96-bit security. This process also equalizes the security levels and allows a more meaningful comparison. Originally, the AGMSY’15 protocol is defined with 64 and 128-bits security levels. The security level of RFE-1 protocol is not described but it uses the lightweight SPONGENT hash function [100] so one can reasonably expect it to be 80-bits. However, the instantiations on ASIP require a redefinition of the utilized cryptographic modules. The transformed versions of these protocols replace the original cryptographic blocks (e.g. SPONGENT) with SIMON modes-of-operations.

A significant challenge of protocol mapping is to transform the error correction operations. We favour offset coding due to its operational simplicity and reasonable entropy performance [81]. The security analysis of this construction can be found in [101] (JW-DRS fuzzy extractors). Figure 3.10 shows the specification, building block interpretation, and the ASIP pseudocode of the offset coding. The offset coding encodes an input random string into a codeword, then XORs the codeword with the PUF response to generate the helper data. This helper data reconstructs the random input if the noise of the PUF output is within the limits of the error correcting capability of the BCH block code. In offset coding, the entropy of the public helper data is equal to the entropy of the random input. The building blocks of this construction are TRNG, BCH code, PUF and XOR. In our ASIP, a single iteration of the offset coding generates 64-bit helper data which has 16-bits of entropy. To achieve a security level of 96-bits, this operation should be replicated 6 times. The figure shows the pseudocode of the first iteration. The symbolic representation is
formulated as \texttt{Mnemonic} (Read Address, Write Address) and \texttt{-} represents the Don’t Care condition.

The RFE-1 and RFE-2 protocols follow a syndrome construction which can be systematically converted into the offset coding by following Figure 3.10 and the operations of RFE-2 are demonstrated by the detailed mapping in Section A.1. The AGMSY’15 protocol does not need such a transformation because it is already defined with offset coding.

The main operations of the RFE-2 protocol are extracting 384-bits of static and
96-bits of dynamic entropy, cryptographic hashing and BCH coding. The AGMSY’15 protocol uses 768-bits of static and 384-bits of dynamic entropy, PRF evaluation, encryption, and the BCH coding. Interested readers can refer to Appendix A.1 and A.2 for the details and the listings of these mappings.

3.5 Implementation Results and Comparisons

We have implemented the proposed hardware architecture of the ASIP in Verilog HDL. We synthesized the Verilog RTL codes to the lowest-cost Xilinx Spartan-6 FPGA (XC6SLX4-2TQG144) with a speed grade -2 using XST. The resulting netlists are placed and routed to the same FPGA using PlanAhead in ISE 14.3. The post place-and-route static timing analyzer reveals that the maximum achievable operating frequency of this design is 81 MHz. The critical path of the design is generating and propagating the control signals between the Instruction Fetch & Decode and Bit-serial Core.

Table 3.3 highlights the resource breakdown of ASIP submodules on a low-cost Spartan-6 FPGA. The design is compiled to minimize the cross-hierarchy optimizations. We can observe that the control overhead of bit-serial design strategy and bit-serialized core are the dominant cost factors. The total area cost of the ASIP is 244 Registers, 262 LUTs and 1 BRAM.

Table 3.4 shows a detailed comparison of secure PUF-based constructions. It includes the implementations of the three secure protocols on our ASIP, on a general-purpose processor, and using a dedicated hardware. We also highlight two PUF-based key generation hardware. The results show the advantages of our ASIP. It
is a flexible low-power engine that can efficiently execute PUF-based protocols and it is smaller than all previous implementations. We also quantify the performance improvement of the ASIP compared to a baseline general-purpose processor. The table quantifies the costs of the protocol operations (BCH, encryption, etc.), therefore the costs related to the PUF core is excluded from these results. Optimizing the PUF core is an orthogonal research direction, which we do not address in this work.

The typical clock frequency of lightweight RFID domains is 100 Kilo Hertz (KHz) resulting in an execution time around 1 second for the ASIP design. With the maximum achievable frequency of 81 MHz, the execution time reduces approximately to 1 ms. On the ASIP, compared to the RFE-2 scheme, AGMSY’15 protocol requires 15% more clock cycles, 36% more communication and 127% more memory. On the other hand, it offers privacy and a formally proven min-entropy level of 96-bits. We
calculate the efficiency metric as $1/(\text{area}_\text{compute} \times \text{cycle}_\text{count} \times \text{area}_\text{memory})$

where $\text{area}_\text{compute}$ is the logic cost of the hardware defined in slices, $\text{cycle}_\text{count}$ is the clock cycles (cc) required to complete the operations, and $\text{area}_\text{memory}$ is the size of the instruction memory which reflects the cost of flexibility. The operating frequency is omitted in the efficiency metric to allow a platform-independent comparison.
<table>
<thead>
<tr>
<th>Application</th>
<th>AGMSY'15</th>
<th>AGMSY'15</th>
<th>AGMSY'15</th>
<th>RFE-2</th>
<th>RFE-2</th>
<th>RFE-1</th>
<th>RFE-1</th>
<th>RMKWD'14</th>
<th>PUFKY</th>
<th>FP'14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>This work</td>
<td>This work</td>
<td>[5]</td>
<td>This work</td>
<td>This work</td>
<td>This work</td>
<td>[74]</td>
<td>[83]</td>
<td>[82]</td>
<td>[102]</td>
</tr>
<tr>
<td>Design Method</td>
<td>ASIP</td>
<td>GPP1</td>
<td>Coprocessor</td>
<td>ASIP</td>
<td>GPP1</td>
<td>ASIP</td>
<td>Dedicated Hardware</td>
<td>Dedicated Hardware</td>
<td>Dedicated Hardware</td>
<td>Dedicated Hardware</td>
</tr>
<tr>
<td>Area (Slices)</td>
<td>66</td>
<td>521</td>
<td>306</td>
<td>66</td>
<td>521</td>
<td>66</td>
<td>165</td>
<td>349</td>
<td>281</td>
<td>384</td>
</tr>
<tr>
<td>Memory Footprint</td>
<td>5648/3666</td>
<td>6624/7056</td>
<td>8320/1040</td>
<td>2096/2002</td>
<td>21504/60832</td>
<td>2000/1898</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Estimated Slices</td>
<td>102</td>
<td>823</td>
<td>343</td>
<td>82</td>
<td>843</td>
<td>81</td>
<td>165</td>
<td>349</td>
<td>281</td>
<td>384</td>
</tr>
<tr>
<td>Execution Time</td>
<td>112199</td>
<td>1440549</td>
<td>18597</td>
<td>97631</td>
<td>3010016</td>
<td>92071</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Power (mW)</td>
<td>22</td>
<td>77</td>
<td>NA²</td>
<td>22</td>
<td>77</td>
<td>22</td>
<td>NA²</td>
<td>NA²</td>
<td>NA²</td>
<td>NA²</td>
</tr>
<tr>
<td>Energy (J)</td>
<td>49</td>
<td>2218</td>
<td>NA²</td>
<td>43</td>
<td>4635</td>
<td>44.5</td>
<td>NA²</td>
<td>NA²</td>
<td>NA²</td>
<td>NA²</td>
</tr>
<tr>
<td>Efficiency</td>
<td>3.68×10⁻¹¹</td>
<td>2.39×10⁻¹⁴</td>
<td>1.69×10⁻¹⁰</td>
<td>7.75×10⁻¹¹</td>
<td>1.05×10⁻¹⁴</td>
<td>8.67×10⁻¹¹</td>
<td>NA²</td>
<td>NA²</td>
<td>NA²</td>
<td>NA²</td>
</tr>
<tr>
<td>Communication Cost</td>
<td>192/1248</td>
<td>192/1248</td>
<td>256/2168</td>
<td>768/288</td>
<td>768/288</td>
<td>288/672</td>
<td>NA²</td>
<td>128/1668</td>
<td>0/128</td>
<td>0/256</td>
</tr>
<tr>
<td>Security Level</td>
<td>96</td>
<td>96</td>
<td>128</td>
<td>96</td>
<td>96</td>
<td>96</td>
<td>80</td>
<td>128</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>Reconfiguration</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
<td>Very High</td>
<td>Very High</td>
<td>Very High</td>
<td>Very High</td>
</tr>
<tr>
<td>Complexity</td>
<td>Update Microcode</td>
<td>Update Software</td>
<td>Modify</td>
<td>Update</td>
<td>Update</td>
<td>Update</td>
<td>Update</td>
<td>Redesign</td>
<td>Redesign</td>
<td>Redesign</td>
</tr>
<tr>
<td>Reconfiguration Method</td>
<td>Update</td>
<td>Microcode</td>
<td>Software</td>
<td>Hardware</td>
<td>Software</td>
<td>Microcode</td>
<td>Update</td>
<td>Redesign</td>
<td>Dedicated Control</td>
<td>Dedicated Control</td>
</tr>
<tr>
<td>Target FPGA</td>
<td>XC6SLX4</td>
<td>XC5VLX30</td>
<td>XC5VLX30</td>
<td>XC6SLX4</td>
<td>XC5VLX30</td>
<td>XC6SLX4</td>
<td>XC5VLX50</td>
<td>XC5VLX110</td>
<td>XC6SLX45</td>
<td>XC5VFX70</td>
</tr>
<tr>
<td>Summary</td>
<td>Smallest, Low-power</td>
<td>Baseline Design with C</td>
<td>Fastest, Most-efficient</td>
<td>Smallest, Low-energy</td>
<td>Baseline Design with C</td>
<td>Smallest, Low-power</td>
<td>Dedicated Hardware</td>
<td>Dedicated Hardware</td>
<td>Dedicated Hardware</td>
<td>Dedicated Hardware</td>
</tr>
</tbody>
</table>

1. GPP: General-purpose Processor
2. NA: Not Available
3. Area is estimated from LUTs and Registers
4. Using the method in [35], memory footprint is converted into equivalent slices (1 slice = 4 LUT6 = 256-bits of memory)
5. To compensate for different target FPGAs, dynamic power consumption is reported
6. The power and energy results are estimated at 50 MHz
7. The RMKWD’14 protocol also requires 64 Kilo Byte (KB) ROM and 4 KB RAM for TRNG, PUF area is not included
The general-purpose processor implementations utilize a 16-bit MSP430 soft-core processor \cite{103} and C programming language. The reference C software is the same for the ASIP and for the MSP430 realization which means that they use exactly the same building blocks for operations (hash, BCH, etc.). The difference is that the ASIP code is generated manually (see Appendix A.1 and A.2 for opcodes) while the \texttt{mspgcc} assembler \cite{104} is used to generate the MSP430 assembly codes. The MSP430-based design cannot fit into the target low-cost Spartan-6 FPGA and it occupies 2084 LUTs, 684 registers and 8 BRAMs on a medium-cost Xilinx Virtex-5 FPGA (XC5VLX30-1FFG324). This time, unlike their ASIP performance, RFE-1 and RFE-2 has a worse execution time compared to the AGMSY’15 protocol. The primary reason for this difference is hidden in the implementation of the hash function. The RFE-2 uses hash functions while the AGMSY’15 utilizes encryption and PRFs. Since the lightweight hash implementation uses the Hirose construction, it requires two encryption calls to perform hashing. In our ASIP design, these two encryption blocks work in parallel. In contrast, MSP430 executes everything sequentially. Therefore, on our ASIP design, hash function executes faster in comparison to the other cryptographic operations, accelerating the RFE-2 relatively more than the AGMSY’15.

Previous designs of reverse fuzzy extractors and AGMSY’15 protocol are dedicated hardware architectures or coprocessors and are mapped on medium-cost Virtex-5 FPGAs (which has the same slice elements but a faster interconnect compared to Spartan-6). Compared to our ASIP which optimizes the area-cost for lightweight applications, they target high-performance applications and thus require a significantly larger area.

Figure 3.11 visualizes the area-cost and the execution time of several implemen-
Figure 3.11: Implementation results and comparison of PUF-based protocol implementations. The figure clearly demonstrate the area advantage of serialization and integration with reuse compared to a general-purpose processors and even to a dedicated solution.

tations. The figure verifies that compared to a baseline implementation on general-purpose microprocessors, the ASIP has a better performance and smaller memory footprint. Therefore, the ASIP design offers an improvement of up to three orders of magnitude. Of course, our baseline implementation on MSP430 can be improved by using assembly codes. A recent work on similar ciphers (SPECK-96) shows that assembly coding can improve the execution time on MSP430 by $4 \times [105]$ which is, as expected, not enough to compensate for a gap of three orders of magnitude. Moving to a dedicated solution [5] that is not serialized further improves the execution time at the expense of area-cost. Arguably, the non-serialized solution in [5] is more efficient than the serialized proposal but our context in Part I is resource-constrained systems not resource-efficient ones. Note that this figure does not include the computational
performance of the dedicated solution for RFE-1 protocol as its cycle count is not documented in the previous work [73], [74].

**Related Work:**

In our best knowledge, this work is the first one to design a flexible architecture for PUF-based protocols and we also propose optimizations for lightweight applications. However, there is a large body of work on flexible engines for conventional cryptographic systems, including customized GPPs, (reconfigurable) hardware crypto co-processors, crypto processors, and crypto arrays. Interested readers can refer to [106] for an extensive overview of these proposals. Most recently, Hutter et al. proposed a lightweight ASIP design for the NaCl’s crypto library [106]. They have the same optimization targets (area and power) and they also use alternative ciphers (e.g., Salsa20), but they work on conventional protocols (including public-key primitives) and they target ASIC technology.

### 3.6 Conclusion and Future Work

This chapter demonstrates the application of serialization and integration with reuse on a complex ASIP design targeted for PUF-based authentication protocols. We quantified that our approach can address both flexibility and resource-constrained requirements; the ASIP is smaller than general purpose microprocessors and even than a dedicated non-serialized solution. We confirm these claims by mapping three secure PUF-based protocols with respective design methods. We validate that by bit-serialization, reusing a lightweight core, simplifying the architecture and with
clever selection of an instruction set, a designer can achieve a compact and flexible solution.

One of the major outcome of this work is to reveal that the classic understanding of ‘lightweight’ do not fully reflects itself into the PUF technology. Traditionally, lightweight referred to protocols that do not use cryptographic operations like hashing, encryption, etc. However, in recent years, new cryptographic building blocks are proposed which can, if implemented properly as shown, occupy a small space while supporting many operations. This makes the noise handling mechanism to become the bottleneck of the PUF system. This fact is not obvious purely from a theoretical point-of-view if one does not have lightweight implementation expertise or if one investigates single blocks without considering the full integrated system. The Slender PUFs successor [83] is designed and implemented based on the traditional ‘lightweight’ insights. Instead of using a combination of noise handling and symmetric-key cryptographic operations, they rely on pattern matching. However, the operations required to implement pattern matching end up costing more than a carefully design crypto core with reverse fuzzy extractors. As a result, their design costs $5\times$ more than our ASIP. We do not discredit protocols based on other principles, under different error capability assumptions, they may be more advantageous. On the contrary, we welcome exploring such scenarios and building engines with different principles that are smaller than this work.

To date, there are only three secure protocols with reverse fuzzy extractors and we have implemented all on our ASIP. However, we expect to see more protocols in the future. For example, one could convert some protocols using forward fuzzy extractors to a reverse fuzzy extractor based protocol. Therefore, our future work is to fully automate our framework. To achieve this goal, we are planning to develop
an assembler that can transform a high-level specification (in the form of C or a
GUI) into the ASIP machine codes. Currently, this is manually performed and it
takes around 24 engineer-hours to map and verify a single protocol. With proper
automation tools, we expect to reduce it to several minutes. We envisage that such
a framework would be beneficial to easily and rapidly prototype and benchmark
protocols under development.
Chapter 4

Lightweight PUF Integration

This chapter describes a lightweight strategy to combine multiple PUF components within an electronic device into a single, unified, board identity. We investigate different methods and we reveal that an integration with reuse strategy, relying on a single error correction code, can simplify PUF-based authentication protocols. We analyze the security vulnerabilities of the proposed techniques and formulate the conditions to provide a certain level of assurance. Based on these formulations and the characterization of multiple PUF components on a population of 22 devices, we quantify error coding requirements of several approaches.

4.1 Introduction

The constrained IoT devices include several commercial off-the-shelf components such as sensors, reconfigurable units, and memory elements. These devices work in a remote and unsecured medium. Hence, IoT node has to provide an assurance that
the board installed in the field is physically the same as the one that was originally deployed. This can reveal hardware tampering, or tamper within the supply chain from manufacturing/deployment to remote installation. This Chapter presents a technique for hardware attestation of such an IoT node that includes a collection of a diverse set of electronic components.

A naive solution to the problem is to add a board-level identifier, such as for example implemented through a hard-coded identifier in non-volatile memory. Of course, an unprotected non-volatile memory is not tamper-proof. This makes the proof of hardware attestation meaningless. But even a tamper-protected non-volatile memory is not an adequate solution. A protocol based on authenticating this remote hard-coded identifier would only demonstrate the presence of the non-volatile memory. It would still require a tamper-resistant integration of the physical IoT with the non-volatile memory.

Therefore, we need a mechanism that authenticates all major computational and memory elements of the IoT itself as part of attesting the IoT node. Such elements can include FPGAs, SRAM memory, processors, sensors, as well as non-volatile memory. By extracting a hardware fingerprint from these components using Physical Unclonable Functions (PUFs), we can construct the fingerprint of an entire IoT by considering the ensemble of chips on the IoT device. It is clear that this makes the hardware attestation more trustworthy: tampering with any single chip (such as a non-volatile memory) will not allow forgery of the entire IoT node.
4.1.1 Hardware Attestation with Fusion PUF

Figure 4.1 sets up our authentication scenario. An IoT node is installed at a remote site, and it needs to be attested by an infrastructure operator. Each IoT device has a unique identity, and the verifier (either a server or another IoT node) maintains a list of installed, valid nodes that is used to verify if a tested IoT device is valid. The verifier runs a remote hardware attestation on the IoT node, and wants to correctly identify every previously installed and enrolled IoT devices. In addition, the operator also wants to systematically detect unknown, previously non-enrolled IoT nodes.

The IoT node runs a remote attestation protocol that demonstrates its presence using a device-level PUF, constructed from individual component-level PUFs. We will use the term Fusion PUF to describe this device-level PUF. Depending on the statistical characterization of the quality of each PUF component, PUF Fusion
process will generate a method and its parameters to integrate multiple PUF components. Thus, the higher-level cryptographic application virtually interacts only with a single device-level PUF.

The rationale for using a PUF is to ensure that a proof of identity can be established on the presence of components on the IoT device, rather than on a stored secret. This is driven by similar concerns as a two-factor authentication protocol - we transform the authentication problem from *something that the IoT stores* to *something that the IoT has*. We aim to establish this assurance with a security level of 80-bit and a probability of error of one part in one million.

### 4.1.2 Contributions and Outline of the Chapter

This chapter brings two contributions. First, it describes a methodology for dimensioning and parameter design of PUFs. Second, it proposes an efficient integration of multiple PUFs into a Fusion PUF, which validates our approach on a collection of 22 boards.

In the Section 4.2, we give a brief background on PUF protocol parameters and introduce a methodology for PUF protocol dimensioning. In Section 4.3, we describe the Fusion PUF and formulate the conditions to satisfy protocol requirements. Section 4.4 analyzes the complexity of the proposed techniques. Section 4.5 concludes the chapter.
Table 4.1: Table of glossary for PUF-based authentication protocols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Type</th>
<th>Measured in</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
<td>Error Rate</td>
<td>Protocol Input</td>
<td>%</td>
</tr>
<tr>
<td>$d$</td>
<td>Distance Rate</td>
<td>Protocol Input</td>
<td>%</td>
</tr>
<tr>
<td>$H_{\text{min}}$</td>
<td>Min-entropy</td>
<td>Protocol Input</td>
<td>%</td>
</tr>
<tr>
<td>$\text{FAR}$</td>
<td>False Accept Rate</td>
<td>Protocol Constraint</td>
<td>%</td>
</tr>
<tr>
<td>$\text{FRR}$</td>
<td>False Reject Rate</td>
<td>Protocol Constraint</td>
<td>%</td>
</tr>
<tr>
<td>$h$</td>
<td>Desired Entropy</td>
<td>Protocol Constraint</td>
<td>bits</td>
</tr>
<tr>
<td>$n$</td>
<td>PUF data size, Codeword Length</td>
<td>Protocol Parameter</td>
<td>bits</td>
</tr>
<tr>
<td>$k$</td>
<td>Message Length</td>
<td>Protocol Parameter</td>
<td>bits</td>
</tr>
<tr>
<td>$t$</td>
<td>Correction Capability</td>
<td>Protocol Parameter</td>
<td>bits</td>
</tr>
</tbody>
</table>

### 4.2 Parameter Definition in PUF Protocols

Over the last decade, many quality metrics and protocol parameters have been proposed for PUF based authentication systems. Table 4.1 enumerates a glossary of symbols that we used. We argue that the entries in this list are sufficient to describe the parameters of PUF based authentication protocols for our scenario. In this Section, we will first introduce the Part 2 of background on PUF technology which relates to this Table. Then, we discuss a generic methodology that uses the Protocol Inputs and Protocol Constraints to generate Protocol Parameters.
4.2.1 Background on Silicon PUF Technology — Part 2

Figure 4.2 plots a distribution of $n$-bit PUF responses. The Intra hamming distance (HD) represents noise; it is the HD of the same PUF response measured a number of times on the same device. Error rate $e$ is the probability of a bit error. Hence, given $n$-bits, we expect the Intra HD to be $n.e$ which is the mean ($\mu_{HD,intra}$) of Intra HD distribution. Inter HD represents the uniqueness of PUF responses; it is the HD of the same response measured on different devices. Distance rate $d$ is the probability of bit-flip between two instances of the same PUF. Hence, given $n$-bits, we expect the Inter HD to be $n.d$ which is the mean ($\mu_{HD,inter}$) of Inter HD distribution.
Unfortunately, it is not practical to know the true value of \( d \) and \( e \), doing so would require to test every single device in the population. However, using statistics, we can test a sample set and estimate our results to cover a large percentage of the population (see [107] for instructions).

There are two metrics that relate to the authentication quality: FAR and FRR. FAR refers to an invalid PUF that happens to generate a valid ID and thus falsely authenticated. FRR, on the other hand, is a genuine PUF mislabelled as a fake because of the PUF noise. Definitely, an authentication system aims to minimize both of these ratios. The authentication threshold value \( t \) defines the HD to determine valid/invalid PUFs. FAR is the probability of observing less than \( t \)-bit distance on an \( n \)-bit response and FRR is the probability of observing more than \( t \)-bit error on an \( n \)-bit response.

Recall that, we need an error correction on PUF responses in order to use them in cryptographic protocols. Again, as in Chapter 3, we use BCH coding and offset-coding mechanism (see Figure 3.10 for details). In offset-coding scheme, \( n \)-bit PUF output is XORed with the \( n \)-bit codeword of the BCH error coder. This error coder should at least be able to correct up to \( t \)-bit errors (where \( t \) is the authentication threshold) on an \( n \)-bit noisy codeword \(^1\). The resulting BCH parameters \((n,k,t)\) provide at least \( n.H_{min} - (n - k)\)-bits of residual entropy, where \( H_{min} \) is the min-entropy of the original PUF content, which can be estimated by using NIST’s recommendations [108].

\(^1\)If multiple \( n \)-bit blocks are decoded, \( t \) is set so that all of these blocks can be corrected with a rate less than FRR.
4.2.2 Protocol Parameter Design Flow

We consider the coding requirements for the use of a PUF-generated key as the basis for a classic MAC-based authentication protocol. Such a protocol works by enrolling the PUF key once, and by verifying it later through a MAC-based challenge-response. Compared to a traditional implementation, the secret key is noisy and thus a protocol that uses it requires additional analysis. This analysis reveals how many PUF response bits are required to generate a stable key with a given target entropy. The analysis also reveals how much error coding is required to obtain a key with a given target error rate.

Figure 4.3 shows a sequence of steps to derive parameters of a PUF-based au-
Lightweight PUF Integration

authentication protocol. We assume two PUF components \( PUF_1 \) and \( PUF_2 \) with bit error probabilities \( e_1 \) and \( e_2 \), and inter-device bit-flip probabilities of \( d_1 \) and \( d_2 \). These probabilities can be estimated for the population by statistical means [107]. Additional constraints include the expected misidentification rates including the FAR and FRR, and desired security level \( (h) \).

- **PUF Fusion** defines how to merge PUF components with different noise \((e_1, e_2)\) and inter-distance \((d_1, d_2)\) characteristics, and how to derive the required PUF bits \( n_1, n_2 \), with the identification threshold \( t \) that satisfies the FAR and FRR. We note that the choice of a pair \( n_1, n_2 \) that satisfies a given FAR and FRR is not unique. In that case, multiple pairs should be considered as possible candidates for the following steps. The most suitable pair would be the one that meets an optimality criterion for the entire system.

- Next, **Error Coding** searches a suitable coding scheme that can correct up to \( t \)-bit errors on an \( n = n_1 + n_2 \) bit codeword. Using this coding scheme with code-offset mechanism [81], it is possible to generate a helper data to remove the noise of genuine PUF responses. However, this helper data contains parity information which reduces the entropy content of the PUF response.

- **Entropy Analysis** calculates the original entropy content of the PUFs and computes a lower-bound \( H_{\text{min}} \) on the residual entropy after the error coding. To accumulate the target entropy \( h \), the protocol can process \( \lceil h/H_{\text{min}} \rceil \) blocks of \( n \)-bit PUF responses. To satisfy the FAR and FRR requirements, the protocol has to apply an error correction of up to \( t \)-bits on each of these \( n \)-bit PUF responses.
In this section, we discuss four alternatives to merge PUF components with different error ($e$) and inter-distance ($d$) characteristics: protocol-level fusion, component-level fusion with concatenation (CAT), component-level fusion with XOR operation, and circuit-level fusion. Figure 4.4 visualizes these four methods.

### 4.3.1 Protocol-level Fusion

In protocol-level fusion (Figure 4.4 (a)), the device extracts PUF responses and separately generates helper data of individual components. Likewise, PUF responses

Figure 4.4: Representation of the four fusion methods
are reconstructed by separate error decoding functions with distinct parameters. Then, the verifier could detect if any of the PUF components are fake. Therefore, on an authentic IoT node, the FRR and FAR requirements has to individually hold for every component. The authentication protocol sets an identification threshold \( t \) and assumes that an enrolled \( n \)-bit PUF response could have at most \( t \)-bit noise. Hence, given the probability of bit error \( e \) we can calculate the FRR by using Equation 4.1. The FRR corresponds to the likelihood of observing more than \( t \)-bit errors on an \( n \)-bit response from the same PUF instance. Likewise, given the probability \( d \) of a bit-flip between two instances of the same component, we can calculate the probability of FAR by using Equation 4.2. The FAR corresponds to the likelihood of observing less than a \( t \)-bit distance between \( n \)-bit responses from different PUF instances. These equations estimate error rates by using the cumulative binomial distribution function \( F_c(t, n, p) \). This function computes the probability of up to \( t \) successes in \( n \) independent trials each having a success rate of \( p \). Since the error coding capability is proportional to its entropy leakage, it is desirable to set \( t \) as the minimum integer value that satisfies FAR and FRR.

\[
\text{FRR}(t, n) = 1 - F_c(t, n, e) \tag{4.1}
\]

\[
\text{FAR}(t, n) = F_c(t, n, d) \tag{4.2}
\]

The disadvantage of protocol level fusion is that two separate block codes are needed on the device and on the server. This may limit its usability in an embedded context, where the program memory of the target microcontroller or the logic area of the target FPGA is limited. If the system is designed for a single block code,
the error-coding parameters must satisfy the error rates of the lowest quality PUF. This would cause performing excessive error-corrections on high-quality PUFs, and getting less entropy from them.

4.3.2 Hardware-level Fusion

In component level-fusion, the device extracts the PUF response on individual components, combines them, and generates a single helper data for the entire collection. Now, the high-level kernel perceives PUF block as a unified single instance which we call the Fusion PUF. The system applies a single block code on the Fusion PUF and generates a single helper data. Using this helper data, the verifier should be able to authenticate the IoT node with the target error rates. Depending on how the Fusion PUF combines the PUF components, the FRR and FAR are calculated differently. We investigate two variations, a concatenation-based and an XOR-based component-level fusion.

Concatenation-based

In concatenation-based fusion (Figure 4.4 (b)), the Fusion PUF concatenates the response $PUF_1$ and $PUF_2$ respectively having $n_1$-bits and $n_2$-bits. Hence, the response size of Fusion PUF is $n = n_1 + n_2$-bits. On average, the Fusion PUF with $n$-bit response would have $n_1e_1 + n_2e_2$ bit noise. If this noise is more than $t$ for a genuine PUF, then this would lead to a false reject. Equation 4.3 shows how to calculate this probability for parameters $n_1$, $n_2$, and $t$. $F_p(t, n, p)$ is the binomial probability density function which computes the probability of exactly $t$ successes in $n$ independent trials each having a success rate of $p$. For a successful authentication,
if one component of a genuine Fusion PUF has exactly $t - i$-bit noise, the other must at most have $i$-bit noise, and vice versa. The sum of all such error combinations from 0 to $t$ would give the total probability of observing errors with threshold $t$. Although it achieves the same FRR, this scheme would allow correcting some error occurrences that are not possible with the protocol-level fusion. For example, if $PUF_1$ has $t_1 + 1$-bit error, fusing at protocol-level would always fail, but the component-level fusion would succeed if $PUF_2$ has less than $t - t_1$-bit error.

\[
FRR_1(n_1, n_2, t) = \sum_{i=0}^{t} (F_p(i, n_2, e_2)(1-F_c(t-i, n_1, e_1)))
\]

\[
FRR_2(n_1, n_2, t) = \sum_{i=0}^{t} (F_p(i, n_1, e_1)(1-F_c(t-i, n_2, e_2)))
\]

\[
FRR = \max\{FRR_1, FRR_2\}
\]  

An adversary can try to trick the protocol by only changing one Fusion PUF component. If such a device is validated, it would be a false accept. Therefore, FAR should reflect the necessity to detect even if a single PUF component is changed within the IoT node. In this scenario, the intra-Hamming distance of the genuine PUF plus the inter-Hamming distance of the fake PUF should be greater than the threshold $t$. For example, if $PUF_1$ is fake, and if $PUF_2$ has $i$-bit error, the Fusion PUF will be misidentified as valid (FAR) if $PUF_1$ have at most $t - i$ bit inter-distance. The sum of all such combinations of $i$ from 0 to $t$ would give the total FAR. This would also guarantee an invalid authentication if both PUF components are fake. Equation 4.4 formulates these conditions for the two PUF setting and for parameters $n_1$, $n_2$, and $t$. 
\[ \text{FAR}_1(n_1, n_2, t) = \sum_{i=0}^{t} (F_p(i, n_2, e_2)F_c(t-i, n_1, d_1)) \]

\[ \text{FAR}_2(n_1, n_2, t) = \sum_{i=0}^{t} (F_p(i, n_1, e_1)F_c(t-i, n_2, d_2)) \]

\[ \text{FAR} = \max\{\text{FAR}_1, \text{FAR}_2\} \]

**XOR-based**

XOR-based fusion (Figure 4.4 (c)) uses an XOR operator to mix the PUF responses of equal size \(n\). The parameters for threshold \(t\) and the size \(n\) can be calculated exactly like a single PUF instance by using equations 4.1 and 4.2. However, the binomial estimators \(e\) and \(d\) have to be modified slightly as the XOR operation will affect the characteristics of the Fusion PUF. Now, a Fusion PUF bit is noisy if and only if one of the two of its input is noisy. Therefore, the bit error \(e_V\) of the Fusion PUF is defined as \(e_1 + e_2 - (e_1e_2)\). When only one PUF component is fake, a Fusion PUF bit flips if one of its input is flipped but not both. Assuming that either PUF can be fake, the \(d_V\) is defined as \(\min\{d_1(1 - e_2) + (1 - d_1)e_2, d_2(1 - e_1) + (1 - d_2)e_1\}\). This would also guarantee detecting fake Fusion PUFs when both PUF components are fake.

XOR-based fusion may have interesting applications due to its privacy-friendly authentication feature. The verifier can detect if the device is not authentic but it can not detect the source of authenticity. XOR operations amplify entropy but they also increase the noise. Therefore, XOR-based fusion is also useful if the PUF components are reliable but have a low-entropy.
4.3.3 Circuit-level Fusion

Circuit-level fusion, depicted in Figure 4.4 (d), is to design a single PUF component with an architecture that is interwoven from multiple PUFs. This PUF design can combine the advantages of different types of PUFs. For example, Ganta et al. designed a novel PUF which is a mixture of Ring Oscillator PUF and Arbiter PUF [109]. The circuit combines the CRP expansion advantage of Arbiter PUFs with the reliability and design simplicity of the RO PUFs. Unfortunately, circuit-level fusion is not suitable for our scenario because the systems we target include several off-the-shelf components, some of which are not reconfigurable. Therefore, we have to use an integration technique at higher abstraction levels.

4.4 Implementation Results

We characterized our methods on 22 Altera DE2-115 boards with 2 PUF components: FPGA PUF and SRAM PUF. Table 4.2 summarizes our experimental setup and the error/distance rate of the PUF instances. The board has a Cyclone EP4CE115 FPGA and we used the popular Ring-Oscillator PUF with 255 Ring Oscillators to extract PUF behavior. We follow the guidelines of Feiten et al. [110], which describes several optimizations on Altera FPGAs, including the optimal logic design for ROs and manual back-end tool manipulations. Then, we analyze the raw RO frequencies on all FPGAs and minimize systematic bias further by comparing the ROs that have closest frequency, similar to sequential pairing [111]. This method would generate $n-1$ bit responses from $n$ ROs. The board also incorporates an ISSI IS61WV102416BLL SRAM, which is a 1024K × 16 complimentary metal-oxide-semiconductor (CMOS)
Table 4.2: Experimental setup

<table>
<thead>
<tr>
<th>PUF component</th>
<th>Cyclone EP4CE115 FPGA</th>
<th>ISSI IS61WV102416BLL SRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUF type</td>
<td>Ring-Oscillator</td>
<td>Power-up state</td>
</tr>
<tr>
<td>PUF characterization size</td>
<td>255 RO</td>
<td>2 MB</td>
</tr>
<tr>
<td>Error rate (e)</td>
<td>0.0128</td>
<td>0.0522</td>
</tr>
<tr>
<td>Distance rate (d)</td>
<td>0.4867</td>
<td>0.4838</td>
</tr>
</tbody>
</table>

static RAM chip. The board incorporates a Cyclone EP4CE115 FPGA and a ISSI IS61WV102416BLL SRAM. We have used the entire SRAM contents to derive the parameters. The reference responses for characterization are generated by measuring the PUF response three times and by using a majority voting. Then, for error rate measurements, we measure each PUF response five times on every board and compare it with the reference value. All measurements are performed under normal operating conditions.

4.4.1 Parameter Extraction for Fusion PUF

We derive the binomial estimators from experiments on 22 boards. We implement a simple brute-force search to find all possible combinations of \(n_1\) and \(n_2\) up to 511-bits. Table 4.3 shows different configurations for Fusion PUF and associated parameters. \textit{Min. bit} shows the combination of \(n_1, n_2\) that uses the minimum number of PUF response-bits \(n\) to achieve the FAR and FRR requirement. Due to the construction of BCH codes, the optimal error-coding efficiency occurs when the PUF size is equal
Table 4.3: The results of parameter extraction for $d_1 = 0.4838$, $d_2 = 0.4867$, $e_1 = 0.0522$, $e_2 = 0.0128$

<table>
<thead>
<tr>
<th>Fusion mode</th>
<th>Protocol-lvl</th>
<th>XOR</th>
<th>CAT-Balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SRAM</td>
<td>FPGA</td>
<td></td>
</tr>
<tr>
<td>$(n_1,t_1)$</td>
<td>$(n_2,t_2)$</td>
<td>$(n_1,n_2,t)$</td>
<td>$(n_1,n_2,t)$</td>
</tr>
<tr>
<td>$(n,k,t)$</td>
<td>$(n,k,t)$</td>
<td>$(n,k,t)$</td>
<td>$(n,k,t)$</td>
</tr>
<tr>
<td>Min. bit</td>
<td>(73,15)</td>
<td>(48,7)</td>
<td>(84,84,19)</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>(48,9,15)</td>
<td>(80,74,18)</td>
</tr>
<tr>
<td>$n_1+n_2=255$</td>
<td>(127,21)</td>
<td>(127,10)</td>
<td>(128,127,25)</td>
</tr>
<tr>
<td></td>
<td>(127,29,21)</td>
<td>(127,64,10)</td>
<td>(125,15,27)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(255,91,25)</td>
</tr>
</tbody>
</table>

to the order of the generator polynomial (ie. $n = 2^w - 1$). $n_1+n_2=255$ corresponds to this setting where the configurations use a total of 255-bit PUF response.

SRAM PUF noise limits the applicability of XOR-based fusion on our solution. However, it is useful if the system has robust but low-entropy PUF constructions. Concatenation (CAT), on the other hand, requires a single and simple error correction mechanism, and is on par with the protocol-level fusion method in terms of residual entropy. Figure 4.5 shows the details of the hardware-level fusion with concatenation. There are 76 possible combinations of $n_1$, $n_2$, and $t$ that satisfy Equation 4.3, 4.4, and $n_1+n_2 = 255$. The Figure highlights three possible combinations, Min-RO, Min-SRAM, and Balanced. We opted for a balanced concatenation combination that merges 128-bits of SRAM with 127-bits of RO PUF response. This selection is also preferred if one wants to evenly distribute the entropy among Fusion PUF input sources with similar min-entropy rates. The residual entropy ($H_{min}$) is approximately 40-bits for
Figure 4.5: Different combinations of $n_1$, $n_2$, $t$ for concatenation based fusion and the associated residual entropies

our observed min-entropy rate of 80% while it is 91-bits for an ideal entropy rate of 100%. To achieve the target entropy of 80-bits, the process should iterate ($\lceil h/H_{\min} \rceil$) twice, and use a total of 256-bit SRAM response and 254-bit RO PUF response.

The parameters of our authentication protocol are a concatenation-based fusion with $(n_1=128, n_2=127)$ Balanced concatenation, $\lceil h/H_{\min} \rceil = 2$, with BCH error coding parameters $n_{BCH} = 255$, $k_{BCH} = 91$, and $t_{BCH} = 25$.

The reliability of the Fusion PUF can be improved with hierarchical codes, by first applying a repetition code and then the BCH code. There is a debate on the residual entropy after revealing the helper data with repetition coding ( [112]
Table 4.4: Comparison of our work with [71]

<table>
<thead>
<tr>
<th></th>
<th>This Work</th>
<th>System-of-PUFs [71]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration level</td>
<td>Component</td>
<td>Component</td>
</tr>
<tr>
<td>Security</td>
<td>Relies on fuzzy extractors</td>
<td>Modelling weakness [75]</td>
</tr>
<tr>
<td>Crypto Operations</td>
<td>Both sides</td>
<td>Verifier side</td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Coding Overhead</td>
<td>Both sides</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUF requirement</td>
<td>Somewhat reliable</td>
<td>Highly reliable, modelling-resistant, CRP expansion</td>
</tr>
<tr>
<td>Off-the-shelf suitability</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

vs. [113]). In this work, for simplicity, we do not apply the repetition codes. We also calculate the min-entropy based on the pessimistic lower-bound assumptions. However, a recent work argues tighter unified bounds on fuzzy sketches that could reduce the number of iterations to accumulate the target entropy [114].

### 4.4.2 Comparison with Previous Work

Table 4.4 summarizes a comparison of this work with the previous work. Königsmark et al. proposed a different method to integrate multiple component-level PUFs [71]. Their method, System-of-PUFs, relies on the combination of three types of PUF constructions: Hidden PUF, Guard PUF, and Secure PUF. The scheme is more advantageous compared to ours because it does not have error coding overhead and
only have a very little crypto overhead on the verifier side. Unfortunately, removing error coding and cryptographic operations make their proposal vulnerable to PUF noise and attacks, and their proposal is observed to have a modelling weakness [115]. Moreover, since their proposal need special properties for the three component PUFs, it not suitable to apply on off-the-shelf components.

In contrast, the security of our proposal relies on fuzzy extractors which has been rigorously tested over a decade. The Fusion PUF, especially the XOR-based technique, can have limited suitability for very noisy PUF components, but overall, it is much more feasible on off-the-shelf components.

4.5 Conclusions and Future Extensions

This chapter introduces new methods to combine multiple PUF components into a single board identifier. Rather than using a separate error coding mechanism on each individual PUF component, our technique relies on integration with reuse that can apply a single error coding for a collection of different type of PUFs with various error rates. This can simplify the interaction of the PUF components with the higher-level application that uses those PUF responses because now it is perceived as a single virtual PUF instance. Depending on the noise level, it can also simplify the error coding requirements.

We discussed how to enable Fusion PUF for a system with two PUF components. Therefore, a possible future work is to extend this concept to a comprehensive set of PUF technologies.

An orthogonal advantage of our proposal, which is omitted for the purpose of
this dissertation, is enabling privacy-friendly hardware attestation. The Fusion PUF merges the PUF responses, hence the verifier can detect, among a collection of PUF components, that at least one of them is not trustworthy. However, especially for XOR-based fusion, it can not detect which one is actually fake. This can be useful in applications where the PUF component does not want to reveal its full identity yet has to contribute to the physical assurance.
Part II

Resource-efficient Cryptosystem Design
The Application Context of Part II

As we have discussed in Chapter 1, clarifying the application context allows analyzing algorithms from other perspectives. This is different than the feasibility driven designs, which are often generic, proof-of-concept solutions without a specific application target. In Part II of this dissertation, we have explored a new application context for post-quantum cryptosystems which uses embedded devices for real-time systems. Below, we motivate our application scenarios and then we discuss why precomputation, the primary strategy for the next chapters, is particularly suitable for the exact scenarios we target.

Motivating Application Scenarios

Part II focuses on optimizations for real-time signature generation, rather than for signature verification. We argue that this has many practical applications when the generation needs to be supported in a constrained, embedded system. We illustrate two concrete application scenarios involving user identification and message authentication.

The first example, demonstrated in the Figure 4.6 (a), is the application context of Chapter 5. The figure visualizes an SHM system integrated in road infrastructure such as a bridge or a road surface. These monitors can be energy-harvesting sensor nodes that keep track of significant physical events such as stress, pressure, maximum/minimum temperature and so forth. They are typically integrated into the road infrastructure and their useful service life extends over many years. When a mobile service unit drives along an SHM, the SHM will deliver the signed recorded
Figure 4.6: Examples of low-latency signature generation: (a) structural-health monitoring system, (b) cyber infrastructure system for roadside electronics

data to the unit. We make a reasonable assumption that the data needs to be signed because of legal reasons (in particular when the failure of SHM may result in significant loss). In this scenario, the energy-harvested SHM should provide a relatively fast signature with its limited energy content, to allow the mobile service unit to advance at reasonable speed.

The second example, depicted in the Figure 4.6 (b), is the application context of Chapter 6. The figure demonstrates an upcoming cyber infrastructure system for roadside electronics. The objectives of this system is to perform a basic safety message mechanism for mobile units (cars, trucks, etc) in real-time as they drive past
In contemporary safety check mechanisms, the roadside unit can detect the speed of the mobile unit and in some advanced systems, it can measure the weight of a moving vehicle. Even these advanced weight station systems require a significant modification of the road infrastructure, such as an adjustment of the number of lanes in the road, the guidance of traffic along the weight stations, and slowing down of vehicle for weight measurement.

However, the future cyber infrastructures may consist nothing more than a roadside unit that communicates wireless with the on-vehicle embedded sensors. Through those sensors, this system can measure several key ingredients for safety, such as insufficient tire pressure, inadequate headlights, unfastened seat belts, etc. in addition to speed and weight conditions, and can warn the driver and/or law-enforcement authorities. In some safety-critical vehicles they can even perform additional tests like identity (fingerprint) of the driver, heartbeat of the passengers, geo-location of the vehicle, and so on. The mobile units in this context again require low-latency solutions, although energy may not be the primary issue since the sensors can be powered from the main battery of the mobile unit. The current car communication systems use simple track-and-trace schemes [116] which have serious security and privacy implications [117], [118]. Public-key based solutions (such as digital signatures) may achieve more secure and scalable constructions. For example, an improved security can be realized with each sensor acknowledging a valid authentication request while passing the roadside unit by signing a random challenge. This system topology needs the mobile unit to respond instantaneously on the request; this latency becomes smaller when the mobile unit drives faster, or when the location of authentication requires higher physical precision.
Table 4.5: The asymmetry of the two computing devices

<table>
<thead>
<tr>
<th>Computing Device</th>
<th>Center of the cloud (Servers)</th>
<th>Edge of the cloud (Low-cost embedded systems)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>High-end CPUs</td>
<td>Microcontrollers, FPGAs</td>
</tr>
<tr>
<td>Optimization Targets</td>
<td>Throughput, Power</td>
<td>Rate, Energy, Latency, Area</td>
</tr>
<tr>
<td>Operation</td>
<td>Signature verification</td>
<td>Signature generation</td>
</tr>
<tr>
<td>Delivery Rate</td>
<td>1K per minute</td>
<td>1 per hour</td>
</tr>
</tbody>
</table>

The Case for Efficiency with Precomputation

In Part II, we target optimizations for large-scale real-time security systems where a large-number of low-cost embedded devices generate on-demand signatures and a centralized server performs verification. Table 4.5 summarizes the differences of the computing devices for the target scenario. Both of these examples, the SHM system and the road-tolling system, illustrate cases where the implementation constraints for signature generation and verification are asymmetric. The signature is generated inside of a low-cost embedded system, and it is later verified within a server-class computing infrastructure. The cloud server consists of a sea of high-end Central Processing Units (CPUs) that can batch-verify thousands of signatures per minute. Operations within the cloud are streamlined and the executions are optimized for throughput and low-power, to maximize the number of operations per unit time while keeping a low thermal dissipation. In contrast, the edge of the cloud is a low-cost embedded node. The objective of this work is to optimize the embedded nodes,
not the servers. These nodes become rarely active (e.g., once per hour) but they are real-time, hence they need different optimization strategies than servers.

We apply *precomputation* to address the challenges for realizing these applications. We argue that precomputation can bring *resource-efficiency*, by improving the performance and reducing the resource waste. We illustrate that precomputation allows to do more with given resources. The chapters are organized in tandem.

- In Chapter 5, the target platform is an energy-harvesting embedded node running a low-cost microcontroller with limited energy. The resource is thus energy and we show that precomputation can improve *energy-efficiency*. We apply precomputation to increase the quantity and latency of output for a given energy profile.

- In Chapter 6, the target platform is a relatively more capable embedded system that can incorporate an FPGA. The resource in this case is the area of the FPGA and we show that precomputation can improve *area-efficiency*. We apply precomputation to improve the latency of output for a given area-cost of the FPGA.

Precomputation is very useful for our context because the systems that we target have long idle periods interrupted by an immediate request that requires a quick response. Therefore, their nature allows a precomputed execution by partitioning computations into offline and online phases. The offline phase refers to the precomputable workload that can be handled at idle periods before a request comes. In our context, it corresponds to all computations that can be completed without full knowledge of the message to sign; it includes operations like accumulating an entropy pool, generating keys, selecting random numbers, and computing message indepen-
dent variables. Online computations include all operations that have an immediate dependency on the message. These operations appear with the real-time signing request. The system then transitions to the online phase and quickly generates the output with precomputed coupons.
Chapter 5

Energy-efficient Post-quantum Signatures

Energy-harvesting techniques can be combined with wireless embedded sensors to obtain battery-free platforms with an extended lifetime. Although energy-harvesting offers a continuous supply of energy, the delivery rate is typically limited less than a Joule per day. This is a severe constraint to the achievable computing output on the embedded sensor node, and to the achievable latency obtained from applications running on those nodes. This chapter addresses these constraints with precomputation and shows that precomputation improves the energy-efficiency for such platforms: for a given energy profile, it increases the number of computed signatures an the latency of a signature generation.

Our idea is to reduce the amount of computations required in response to application inputs, by partitioning the algorithm in an offline part, computed before the inputs are available, and an online part, computed in response to the actual input.
We show that this technique works well on hash-based post-quantum cryptographic signatures. By preprocessing the key-related material and other input-independent variables, and by storing them as run-time coupons in non-volatile memory, there is a drastic reduction of the run-time energy needs for a signature, and a drastic reduction of the run-time latency to generate it. For a Winternitz hash-based scheme at 128-bit pre-quantum and 84-bit post-quantum security level on an MSP430 microcontroller, we measured a run-time energy reduction of $11.9 \times$ and a run-time latency reduction of $23.5 \times$.

The rest of the chapter is organized as follows. Section 5.1 motivates the target public-key primitives. Section 5.2 introduces our precomputation methodology and Section 5.3 shows its application on the target primitives. Section 5.4 describes our energy-harvesting platform. Section 5.5 reports the implementation results and highlights related work. Section 5.6 concludes the chapter.

### 5.1 The Need for Alternative Public-key Cryptosystems

Digital signatures are arguably the most important cryptographic primitive. We rely heavily on these signatures to authenticate critical electronic data such as identity information on e-passports, quantity, source, and destination of financial transactions, consumption amount and time of smart-meters, and enterprise names on software distribution. Even tough these applications currently use well-established standard tools like Elliptic Curve Digital Signature Algorithm (ECDSA) and Rivest-Shamir-Adleman (RSA), recent advances in cryptanalysis and quantum-computers motivate
Table 5.1: Security reductions of pre- and post-quantum era

<table>
<thead>
<tr>
<th>Operation</th>
<th>Hash Function</th>
<th>Symmetric Key Encryption</th>
<th>Public Key Encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>SHA-2</td>
<td>AES</td>
<td>ECC</td>
</tr>
<tr>
<td>Key Size</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>Target Security</td>
<td>$2^n$</td>
<td>$2^n$</td>
<td>$2^n$</td>
</tr>
<tr>
<td>Pre-quantum Security</td>
<td>$2^{n/2}$</td>
<td>$2^n$</td>
<td>$2^{n/2}$</td>
</tr>
<tr>
<td>Post-quantum Security</td>
<td>$2^{n/3}$</td>
<td>$2^{n/2}$</td>
<td>$N^3$</td>
</tr>
</tbody>
</table>

new pillars for the future of our digital security.

Traditional cryptography, and classic public-key cryptography in particular, faces an increasing risk at a catastrophic event because of improvements in quantum computing architectures, and also because of continuous progress in the cryptanalysis of traditional public-key algorithms. Table 5.1 highlights the impact of quantum cryptanalysis on the security of fundamental cryptographic constructions. Grover’s algorithm enables a faster brute-force search and hence reduces the security of an $n$-bit key to $n/2$-bits [119]. Likewise, quantum birthday attacks reduce the strong collision resistance of hash functions from $n/2$-bits to $n/3$-bits [120]. While these security reductions indicate that quantum computers affect symmetric key and hash-based constructions, we can still assure pre-quantum security levels by simply doubling and tripling the key size and the hash output, respectively. The case for public-key
encryption is much worse. Shor’s algorithm \[28\] can solve the factorization and the (elliptic curve) discrete logarithm problem in polynomial time \[29\]. Hence, to preserve the pre-quantum security, we have to increase the key size exponentially, which is infeasible in practice. This is the main motivation of post-quantum public-key constructions. We need practical public-key building blocks for the post-quantum era.

5.2 Precomputation for Post-quantum Embedded Systems

The Ant and the Grasshopper is the story of reducing the dependence on future supplies by performing extra work on the currently available resources. This chapter demonstrates the same principle with post-quantum cryptographic signatures on energy-harvesting platforms. We apply precomputation techniques to hash-based signature schemes. We show that by utilizing the current excess resources, the system can achieve improvements of over an order-of-magnitude in the run-time energy and latency requirements of future signatures.

Figure 5.1 explains why optimization of energy-usage on a conventional battery operated system is fundamentally different from optimization on an energy-harvested platform. Figure 5.1(a) shows a battery-operated system. This system is the Grasshopper of the story. It operates towards a state-of-depletion in which the processing unit works until there is no more energy left in the power supply. In this setup, the processor performs monolithic executions; it starts the evaluation when input appears (which happens at an arbitrary moment) and generates output
Figure 5.1: Energy profiles of (a) battery-operated and (b) energy-harvesting systems
with a continuous, uninterrupted execution. In energy-harvesting platforms (Figure 5.1(b)), the system gathers energy from surrounding energy sources such as light or vibration. These systems are fundamentally different because they operate towards a state-of-equilibrium in which the energy input (which varies over time) is equal to the energy consumption. Precomputation is a technique that uses opportunities of the new energy profile. Now, the Ant is able to precompute a section of the algorithm when there is an abundance of energy and prepare for future by reducing its run-time requirements.

5.2.1 Precomputation as a Re-emerging Topic in Cryptographic Computing

In the post-quantum era, not only the security primitives will change but also the paradigm of computing for the embedded systems. In this chapter, we argue that precomputation, an old practice that is typically overlooked in the embedded domain, will be a recurring method for the emerging computing systems. Precomputation in cryptography was previously proposed to accelerate exponentiation or elliptic curve multiplication [121], [122], [123], [124]. A major disadvantage of precomputation is that it usually requires more computation (energy) and storage, two resources that are constrained in traditional embedded systems. However, these assumptions are changing. The technology of flash memory, the predominant storage unit of embedded domains, introduced 15 new generations of products over the last 20 years, accumulating to a cost improvement of 25,000× [125]. This trend will make the integration of more capable storage units increasingly cheaper. On the other hand, although the Moore’s law does not apply to the battery technologies, energy-harvesting
platforms make energy no longer a limited and monotonically decreasing concept.

Evidently, this is not the first research on energy optimization for harvesting nodes. Previous works like Dewdrop [126] and DEOS [127] propose to relabel iterative operations as atomic tasks and then to (dynamically) schedule them for maximum computations with the available energy. In contrast, we leverage our application specific expertise to transform algorithms into a set of divisible tasks. This transformation enables precomputation optimizations with energy-aware partitions; the system precomputes and stores input-independent values at energy-friendly intervals, and minimizes the run-time energy and latency. Mementos also divides the atomic operations but it aims to allocate checkpoints within a task to quickly restart in case of power failure or energy depletion [128]. The techniques we apply in this chapter are orthogonal, they can be implemented on top of the previous work.

Ateniese et al. makes similar claims on precomputation for wireless sensor nodes implementing pre-quantum primitives like ECDSA [129]. They show that a mote can precompute intermediate values when there is an (excess) energy available and then use it to minimize the latency. We extend these strategies for the post-quantum era. We show that a post-quantum signature scheme, hash-based digital signatures, can be accelerated by precomputation techniques. Then, we quantify the efficiency of these techniques in terms of latency, energy, and system yield.

5.2.2 Moving Towards the Precomputed Execution

Table 5.2 presents the glossary of symbols for the precomputation strategy. An energy-harvesting system designer ideally wants to ensure that the system always has sufficient energy ($Q > \epsilon$) so that it can act as battery-operated and compute at any
Table 5.2: Symbols of the precomputation methodology

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>Energy consumption of the monolithic execution</td>
</tr>
<tr>
<td>$t$</td>
<td>Execution time of the monolithic execution</td>
</tr>
<tr>
<td>$\epsilon_p$</td>
<td>Energy consumption of the offline phase</td>
</tr>
<tr>
<td>$t_p$</td>
<td>Execution time of the offline phase</td>
</tr>
<tr>
<td>$\epsilon_o$</td>
<td>Energy consumption of the online phase</td>
</tr>
<tr>
<td>$t_o$</td>
<td>Execution time of the online phase</td>
</tr>
<tr>
<td>$Q_{max}$</td>
<td>Maximum energy on the supercapacitor</td>
</tr>
<tr>
<td>$Q_{excess}$</td>
<td>Excess energy</td>
</tr>
<tr>
<td>$Q$</td>
<td>Current energy on the supercapacitor</td>
</tr>
<tr>
<td>$I$</td>
<td>Current energy influx of the system</td>
</tr>
</tbody>
</table>

given moment. However, this may not be the case due to variations in the harvested energy resources. Therefore, a practical approach to build harvesting systems is to match the estimated total energy influx with the expected average number of output requests within a certain time frame [130]. Then, it becomes the objective of the application designer to maximize availability: the number of iterations the system can perform given a certain amount of energy. Next, we formalize how precomputation can optimize the availability.

**Principles of the Precomputed Execution**

The precomputed execution separates computations into offline and online phases (Figure 5.1(b)). During the offline phase, the system computes coupons, the set
of input-independent variables of the algorithm. Each coupon is unique and every iteration of the algorithm requires a new coupon. With the real-time input, the system transitions to the online phase and generates the output using the precomputed coupon and the input. The main advantage of precomputation is its ability to turn extra energy into coupons each saving $\epsilon_p$ joules and $t_p$ seconds of future iterations. Using a coupon, the precomputed execution reduces the run-time energy from $\epsilon$ to $\epsilon_o$, and the latency from $t$ to $t_o$. This transformation allows performing useful work in smaller steps at energy-friendly intervals, improves the energy-efficiency of the system, and brings the system closer to its state-of-equilibrium.

The example in Figure 5.2 demonstrates the advantage of precomputed execu-
tion on a harvested energy profile. For the example, we assume that $\epsilon_p + \epsilon_o \approx \epsilon$ and $t_p + t_o \approx t$. The excess energy occurs if the energy store is fully charged ($Q=Q_{max}$), no output request exists, and if there is still an energy influx ($I>0$), because the system harvests more energy than it can store or consume. This excess energy is wasted with the monolithic execution but the precomputed execution can generate coupons.

When there are $Q$ joules on supercapacitor, the availability of the monolithic version of the algorithm will be $\lfloor Q/\epsilon \rfloor$. Using precomputed execution and a set of coupons, the availability will increase to $\lfloor Q/\epsilon_o \rfloor$. For instance, in Figure 5.2, the availability of the monolithic execution is 0 when $\epsilon_o < Q < \epsilon$, while the precomputed execution can generate 1 output if there is a coupon.

We define the run-time energy improvement as the ratio of the availability of precomputed execution to the monolithic execution, which corresponds to $\lfloor Q/\epsilon_o \rfloor / \lfloor Q/\epsilon \rfloor \approx (\epsilon/\epsilon_o) \approx (1+\epsilon_p/\epsilon_o)$. This value estimates an expected improvement factor on average, but in practice, the net increase in the availability depends on the exact value of $Q$ and the number of existing coupons. For the example in Figure 5.2, while the run-time energy improvement is $2 \times$, the precomputed execution can triple the availability of the monolithic version for a short period by utilizing 2 coupons. The precomputed execution also reduces latency, the execution time of a single output generation, by a factor of $t/t_o \approx (1+t_p/t_o)$. On the other hand, these techniques are disadvantageous for battery-operated systems since there is no concept of excess energy and since the total energy required for an algorithm iteration is usually more for the precomputed execution ($\epsilon_p + \epsilon_o$) than the monolithic execution ($\epsilon$).
Quantifying the Impact of Precomputation

The key for effectively applying precomputation is to identify the offline and online phases of the algorithm and to maximize \(1 + \epsilon_p/\epsilon_o\) and \(1 + t_p/t_o\), the ratio of offline to online computations. As the driving application, we use hash-based signatures and show how to utilize precomputed execution with energy-harvested setting. We reveal that on these signatures, precomputation has a significant potential to increase run-time energy and latency. We quantify the savings on our energy harvesting platform and show that precomputation can achieve a run-time energy improvement of \(11.9 \times\) and a latency improvement of \(82.7 \times\).

5.2.3 Contributions and Organization

The major contributions of this chapter are as follows.

- We study precomputation, an orthogonal optimization strategy on energy-harvesting platforms and we show that they are applicable on current systems.

- We propose novel precomputation techniques on hash-based signatures

- We investigate the impact of these methods on run-time energy using our energy-harvesting platform. To our best knowledge, this is the first work to quantify and optimize energy of hash-based signatures.

- We also provide relative latency improvements of signature generation on constrained microcontrollers.
5.3 Precomputation of Hash-based Signature Schemes

In this section, we first review how traditional hash-based signatures are implemented. Then, we apply optimizations and transformations to enable precomputation.

5.3.1 Hash-based Signatures

In this work, we implement the Winternitz one-time signature (W-OTS) scheme defined as in [132]. W-OTS offers a trade-off between signature size (communication energy) and execution time (computation energy). We have used the standard SHA-256 as hash kernel because it is fast, and the generic post-quantum security of a $n=256$-bit hash function against pre-image attacks is $n/2=128$-bits [119], while its strong-collision resistance is $n/3\approx84$-bits [120], [133]. Any secure hash function that has at least 256-bit outputs (eg. Keccak) can also replace the SHA-256 in our implementation. H"ulsing et al. shows that replacing hash function with a pseudo-random function and using different security assumptions suffices for the security in the classic case [134]. However, the effect of such transformations on the quantum security is not discussed. Therefore, we used the more conservative approach of Buchmann et al. [133].

In the W-OTS scheme, one generates hash chains $Y = h(h(...h(X))))$ with the secret key $X$ and the public-key $Y$. A signature of message $\mu$ is created by selecting intermediate values on the hash chains using portions of $\mu$. The verifier can, using $\mu$, check if those intermediate values indeed reveal the public-key $Y$ at the end of the chain. Since disclosing hash chain leaks information about the structure
of the signature, a new chain that originates from a new secret key is used for every new message. In what follows, we derive a formal definition of these operations.

The key idea behind the W-OTS scheme is to sign \( \omega \)-bit portions of the message simultaneously. Hence, the Winternitz parameter \( \omega \geq 2 \) determines the trade-off between the execution time and the signature size. The number of required hash computations grow exponentially while the signature and key size reduce linearly. The parameters \( t_1, t_2 \), and \( t \) of the W-OTS are defined in equation 5.1 with \( m = 256 \) (message digest length) for our instantiations.

\[
t_1 = \left\lceil \frac{m}{\omega} \right\rceil , \quad t_2 = \left\lceil \frac{\log_2 t_1 + 1 + \omega}{\omega} \right\rceil , \quad t = t_1 + t_2 \tag{5.1}
\]

The secret key \( X \) is \( t \)-blocks of 256-bit hash output \( (x_0, x_1, ..., x_{t-1}) \) generated from a single random seed and the public-key \( Y \) is \( t \) blocks of 256-bit hash output \( (y_0, y_1, ..., y_{t-1}) \) generated using hash chains of \( 2^\omega - 1 \) length where \( y_i = h^{2^\omega - 1}(x_i), 0 \leq i \leq t - 1 \). The scheme divides the message (or its hash) into \( t_1 \) blocks of \( \omega \)-bits \( (b_{t-1}, ..., b_0) \) and also computes the checksum of \( t_2 \) blocks of \( \omega \)-bits \( (b_0, ..., b_{t-1}) \). Then, depending on the values of these blocks (from 0 to \( 2^\omega - 1 \)), it calculates the signature \( \sigma (s_0, s_1, ..., s_{t-1}) \) with hash chains ranging from 0 to \( 2^\omega - 1 \) as

\[
\sigma = (s_0, s_1, ..., s_{t-1}) = (h^{b_0}(x_0), h^{b_1}(x_1), ..., h^{b_{t-1}}(x_{t-1})) \tag{5.2}
\]

To verify the signature \( \sigma \), one has to first use the message (or its hash) to compute the values of \( t_1 \) and \( t_2 \) blocks of \( \omega \)-bits \( (b_0, ..., b_{t-1}) \). Then, if the signature is valid, applying the remaining hash chains of \( 2^\omega - 1 - b_i \) \( (0 \leq i \leq t - 1) \) to the signature \( \sigma \) should reveal the public-key \( Y (y_0, y_1, ..., y_{t-1}) \).
Generating the secret and the public-key requires $t$ and $t(2^w - 1)$ hash computations respectively. The number of hash operations required to compute the signature depends on the message (or its hash) to be signed and is bounded with $[0, t(2^w - 1)]$. The length of the secret key, public-key and the signature is $t \times n$ bits.

Using Merkle trees [135] or Chaining, the W-OTS can be extended into schemes that can sign multiple messages. Chaining is simply including the next public-key to the message to be signed. Therefore, it requires generating key pairs one iteration in advance. Computing one signature with Chaining consists of four tasks:

$T_1$: Generating the secret key $sk_i$ of the current iteration.

$T_2$: Generating the public-key $pk_{i+1}$ of the next iteration.

$T_3$: Computing the signature on tuple $<\text{message, } pk_{i+1}>$.

$T_4$: Transmitting the signature and the next public-key to the verifier.

Although it enforces the verifier to validate signatures in an order, we opt to implement Chaining as its design does not limit the number of signature generations for a given device.

5.3.2 Precomputation for Hash-based Signatures

We have applied two types of precomputation techniques on hash-based signatures with Chaining. The first one is a fine-grain intra-task optimization on a single task of Chaining while the second is a coarse-grain inter-task scheduling optimization that affects multiple tasks.
Fine-grain Optimizations

This optimization targets task $T_3$ of Chaining. The main idea behind it is to store some of the intermediate steps of the hash chain during the offline phase and to start the chain from the closest possible node during the online phase. There is an obvious trade-off between the memory and the execution time of this operation. If there is sufficient memory space, we could precompute and store all intermediate steps of the hash chain. Then, the online phase simply becomes selecting and loading the appropriate values from the memory.

Figure 5.3 illustrates an example scenario of how precomputation can accelerate the signature generation. If we assume that the entire hash-chain is generated online, then the computation of $s_0$, $s_1$, and $s_{t-1}$ requires $1$, $2^{\omega-1} + 1$, and $2^\omega - 2$ evaluations of the hash function respectively. If we precompute the intermediate hash-chain step
of $h^{2\omega-1}$, the number of evaluation steps for $s_1$ and $s_{t-1}$ shortens to 1 and $2^{\omega-1} - 2$ respectively. Therefore, if we store additional intermediate steps, the number of evaluation steps decreases in proportion.

Coarse-grain Optimizations

Figure 5.4 demonstrates the schedule of the operations of two consecutive iterations of Chaining with four methods. The four methods are (a) Mono, (b) Mono-opt, (c) Precomp, and (d) Precomp-opt. In the schedules, the Load and Store are shown as interleaved with Generate because they require marginal operations compared to the Generate operations. Mono (Figure 5.4(a)) does not store any information between two consecutive iterations. Hence, it recomputes the secret keys for every iteration of the algorithm. This can be optimized with Mono-opt (Figure 5.4(b)) in which the processor stores the secret key of the next iteration which has to be generated while computing its public-key. Now, before generating a signature, the processor can load the precomputed secret key and perform signing. Precomputed execution Precomp (Figure 5.4(c)) can further reduce the latency. The processor can precompute a number of one-time keys (say $j$ keys) during the offline phase and later use those coupons for future signature generations. The processor can stay in the online phase as long as there is a precomputed key. Once the system uses all precomputed keys, it has to return back to the offline phase. This operation effectively removes the generation of the keys from the run-time operations. Finally, Precomp-opt can apply the precomputation of intermediate steps (Figure 5.4(d)) to minimize amount of run-time operations. The flow of Precomp-opt is same with Precomp but it generates and stores the precomputed steps ($ps$) during the offline phase to compute the signature with less operations during the online phase.
Figure 5.4: Operation flow of (a) Mono, (b) Mono-opt, (c) Precomp, and (d) Precomp-opt execution. Rows represent a breakdown of the tasks of Chaining, columns represent the value of index $i$ or $i + j$ of the iterations and arrows represent the data-dependency between iterations.
The performance benefit of precomputation comes with an increased storage cost. For \texttt{Mono}, it is sufficient to just store the 256-bit root of the secret key where as \texttt{Mono-opt} requires $t \times 256$-bits of storage. As it precomputes $j$ secret and public-keys, \texttt{Precomp} needs $t \times j \times 256 \times 2$-bits of memory and if system utilizes $ps$ precomputed steps, then memory cost of \texttt{Precomp-opt} becomes $t \times j \times 256 \times (2 + ps)$-bits.

### 5.4 Target Platform

We used our research prototype with energy harvesting and measurement capabilities. This section gives a brief overview but we refer the interested readers to the work of Pabbuleti \textit{et al.} [136], which describes the complete setup developed by our
Figure 5.6: The block diagram of the research platform

group in further details. Figure 5.5 is a picture of our setup and Figure 5.6 shows the block diagram of the building blocks of our system. A Photovoltaic cell converts the energy from photons into electricity and transmits it to the Anagear ANG 1010 energy-harvesting board. Attached to the Anagear, a low-leakage supercapacitor can accumulate an energy-level up to 710 mJ. This energy can activate the main processing unit of the system, a low-power 16-bit MSP microprocessor with 16KBs of SRAM and 128 KBs of Flash memory that executes the digital signature schemes. The RF Frontend of the system (CC2500 transceiver of Texas Instruments) also uses the harvested energy and it can send the signatures with 64-byte packets. The MSP microprocessor is also responsible to send trigger signals to the measurement unit, an OpenADC board with a Spartan-3 FPGA that can measure the energy consumption of both computation and communication with a high-precision.
5.5 Implementation Results

We used the C programming language to implement the digital signature schemes. We compiled the software codes with the gcc 4.6.3 cross compiler with an optimization level of 02 that minimizes the code size. The processor runs at 10 MHz and does not utilize a hardware multiplier. We used the RELIC 0.3.3 library [137] to realize the hash primitive and select the SHA-256 hash function as it provides an acceptable level of collision resistance (84-bits) against quantum-computer attacks. Even though the software-efficiency can potentially be improved with assembly level programming, we aim to show the relative savings of precomputation.

Here we summarize our conclusions from the implementation result. We derive these observations using our platform and the calibration factors of the results depend on the experimental setup.

- The proposed methods increase the availability of the system and can significantly boost the number of computed signatures especially for critical energy levels.

- Precomputation can improve the run-time performance by an order of magnitude

- The parameter $\omega$ can be tuned to optimize either run-time energy or latency

- The energy overhead of transmitting signatures has a significant impact and should be accounted for optimizing algorithm parameters

- Hash-based signature schemes have comparable time/energy demands with traditional signatures
5.5.1 Implementation Parameters

The experiments evaluate the offline computations, online computations, and the communication of signature components. We assess both the time and energy requirements of these operations. The memory space is fully utilized for all implementations to store precomputed coupons. For hash-based signatures, we experiment with two Winternitz parameters: $\omega=4$ and $\omega=8$, and with three execution methods: Mono-opt, Precomp, and Precomp-opt. We further highlight the memory-performance trade-off that exists within the Precomp-opt by utilizing different number of precomputed steps. We omit Mono and implement Mono-opt as the improvement requires marginal memory space.
Table 5.3: Energy and execution time results of hash-based signatures

<table>
<thead>
<tr>
<th>Method</th>
<th>Offline Energy (mJ)</th>
<th>Offline Time (s)</th>
<th>Online Energy (mJ)</th>
<th>Online Time (s)</th>
<th>Communication Energy (mJ)</th>
<th>Communication Time (s)</th>
<th>Run-time Total Energy (mJ)</th>
<th>Run-time Total Time (s)</th>
<th>Improvement</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono-opt</td>
<td>68.59</td>
<td>36.76</td>
<td>27.53</td>
<td>15.39</td>
<td>4.66</td>
<td>0.13</td>
<td>100.78</td>
<td>–</td>
<td>52.28</td>
<td>–</td>
</tr>
<tr>
<td>Precomp</td>
<td>68.59</td>
<td>36.76</td>
<td>27.53</td>
<td>15.39</td>
<td>4.66</td>
<td>0.13</td>
<td>32.19</td>
<td>3.13×</td>
<td>15.52</td>
<td>3.37×</td>
</tr>
<tr>
<td>Precomp-opt, pre = 1</td>
<td>68.87</td>
<td>36.76</td>
<td>14.01</td>
<td>7.59</td>
<td>4.66</td>
<td>0.13</td>
<td>18.67</td>
<td>5.40×</td>
<td>7.72</td>
<td>6.77×</td>
</tr>
<tr>
<td>Precomp-opt, pre = 3</td>
<td>68.53</td>
<td>36.76</td>
<td>8.10</td>
<td>4.37</td>
<td>4.66</td>
<td>0.13</td>
<td>12.76</td>
<td>7.90×</td>
<td>4.50</td>
<td>11.62×</td>
</tr>
<tr>
<td>Precomp-opt, pre = 7</td>
<td>68.73</td>
<td>36.76</td>
<td>3.84</td>
<td>2.09</td>
<td>4.66</td>
<td>0.13</td>
<td>8.50</td>
<td>11.86×</td>
<td>2.22</td>
<td>23.55×</td>
</tr>
<tr>
<td>Mono-opt</td>
<td>9.81</td>
<td>4.83</td>
<td>3.28</td>
<td>1.88</td>
<td>8.77</td>
<td>0.27</td>
<td>21.86</td>
<td>–</td>
<td>6.98</td>
<td>–</td>
</tr>
<tr>
<td>Precomp</td>
<td>9.81</td>
<td>4.83</td>
<td>3.28</td>
<td>1.88</td>
<td>8.77</td>
<td>0.27</td>
<td>12.05</td>
<td>1.81×</td>
<td>2.15</td>
<td>3.25×</td>
</tr>
<tr>
<td>Precomp-opt, pre = 1</td>
<td>9.77</td>
<td>4.83</td>
<td>1.68</td>
<td>0.92</td>
<td>8.77</td>
<td>0.27</td>
<td>10.45</td>
<td>2.09×</td>
<td>1.19</td>
<td>5.87×</td>
</tr>
<tr>
<td>Precomp-opt, pre = 3</td>
<td>9.93</td>
<td>4.83</td>
<td>0.77</td>
<td>0.42</td>
<td>8.77</td>
<td>0.27</td>
<td>9.54</td>
<td>2.29×</td>
<td>0.69</td>
<td>10.12×</td>
</tr>
</tbody>
</table>
5.5.2 Impact of the Precomputation on Run-time Performance

Table 5.3 reports the breakdown of the energy consumption and the execution time of the hash-based signature schemes. The results separately show the costs of offline operations, online operations, and the signature transmission. The run-time energy and the latency of the Mono-opt is equal to the sum of all these operations whereas the precomputed executions Precomp and Precomp-opt do not include the offline computations. To quantify the memory-performance trade-off, we also provide the results of several precomputation levels. pre = 7, pre = 3, and pre = 1 correspond to hash chains with 7, 3, and 1 precomputed steps respectively (omitting the first step, secret key). Ideally, Precomp-opt would store the complete hash chain but the size of the SRAM in MSP430F5438 is 16KB and it can not contain more than 7 precomputed steps for ω = 8 and 3 precomputed steps for ω = 4. The results show that precomputation can save more than 90% of run-time energy and latency of hash-based signatures. The run-time energy improvement is 11.86× and the latency is reduced by a factor of 23.55×. On our experimental setup, precomputation brings W-OTS run-time requirements to a minimum of 8.5 mJ and 0.69 s, depending on the value of ω.

Table 5.4 shows the number of run-time hash operations for different executions. The computation time of the signature scheme is not fixed and depends on the input message which determines the lengths of the hash chains. Therefore, the table gives worst-case, best-case and average-case savings of hash operations and its expected improvement ratio. These values match closely with the computation results provided in Table 5.3. On our setup, it takes 46758 clock cycles to evaluate
Table 5.4: Run-time hash operation comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Gen. $s_k$</th>
<th>Gen. $p_k$</th>
<th>Sign WCS</th>
<th>Sign BCS</th>
<th>Sign Avg.</th>
<th>Impr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono-opt</td>
<td>34</td>
<td>255×34</td>
<td>255×34</td>
<td>0</td>
<td>127.5×34</td>
<td>-</td>
</tr>
<tr>
<td>Precomp</td>
<td>-</td>
<td>-</td>
<td>255×34</td>
<td>0</td>
<td>127.5×34</td>
<td>3.01×</td>
</tr>
<tr>
<td>Precomp-opt, pre_1</td>
<td>-</td>
<td>-</td>
<td>127×34</td>
<td>0</td>
<td>63.5×34</td>
<td>6.09×</td>
</tr>
<tr>
<td>Precomp-opt, pre_3</td>
<td>-</td>
<td>-</td>
<td>63×34</td>
<td>0</td>
<td>31.5×34</td>
<td>12.17×</td>
</tr>
<tr>
<td>Precomp-opt, pre_7</td>
<td>-</td>
<td>-</td>
<td>31×34</td>
<td>0</td>
<td>15.5×34</td>
<td>24.74×</td>
</tr>
<tr>
<td>Mono-opt</td>
<td>67</td>
<td>15×67</td>
<td>15×67</td>
<td>0</td>
<td>7.5×67</td>
<td>-</td>
</tr>
<tr>
<td>Precomp</td>
<td>-</td>
<td>-</td>
<td>15×67</td>
<td>0</td>
<td>7.5×67</td>
<td>3.13×</td>
</tr>
<tr>
<td>Precomp-opt, pre_1</td>
<td>-</td>
<td>-</td>
<td>7×67</td>
<td>0</td>
<td>3.5×67</td>
<td>6.71×</td>
</tr>
<tr>
<td>Precomp-opt, pre_3</td>
<td>-</td>
<td>-</td>
<td>3×67</td>
<td>0</td>
<td>1.5×67</td>
<td>15.66×</td>
</tr>
</tbody>
</table>

WCS: Worst-Case Scenario, BCS: Best-Case Scenario, Avg.: Average, Gen.: Generate, Impr.: Improvement

As expected, the communication energy reduces linearly while the computation energy grows exponentially with $\omega$. Likewise, the execution time and the energy of the online phase scales down linearly with the increase in the precomputed steps. With the available memory and the transmission infrastructure, hash-based signatures with $\omega = 8$ requires less run-time energy while $\omega = 4$ has a lower latency. Since a monolithic ECDSA signature operation takes 91 mJ and 12.5 s to compute on the same platform [136], we can argue that the hash-based signatures are quite
competitive, if not better suited for real-time applications with energy constraints.

The experiments also reveal that the extra operations of Precomp-opt, which is storing intermediate steps of the hash chains, require minimal calculations (approximately <1% variation) during the offline phase. Hence, we conclude that the cost of extra preparatory operations of precomputation is marginal especially compared to its saving and given more memory space, the precomputation can further reduce the execution time and the energy of the online phase.

We emphasize that we apply orthogonal optimization strategies. Indeed, optimization at other abstraction levels, such as algorithmic-level optimizations like using AES with a suitable mode-of-operation instead of SHA-3, or architecture-level improvements like moving towards more sophisticated microcontrollers with crypto co-processors or with Dynamic Voltage Frequency Scaling (DVFS) capability can further enhance our results. We do not aim to show the fastest implementation to date but to demonstrate the relative savings of the precomputation strategies.

5.5.3 Impact of the Precomputation on Availability

In this section, we quantify the opportunities discussed in Section 5.2.2 which is increasing the availability of the system. In practice, this improvement results in providing more service (more signature) for the same energy influx, because pre-computation avoids wasting harvested energy that cannot be immediately used or stored.

The energy accumulated by the system depends on many factors such as the quality and the size of the photovoltaic cell and the supercapacitor, the geo-location of the board and its relative position to the light sources, the timing of the sun-
rise/sunset, and the weather condition. Therefore, rather than reporting our ad-hoc results, we refer to the typical energy levels detailed in [131]. On our platform, the maximum energy level ($Q_{\text{max}}$) is 710 mJ and the excess energy level ($Q_{\text{excess}}$) for this energy profile is 827 mJ.

Figure 5.7 quantifies the increase in the availability with precomputation for the hash-based signature scheme. Although the expected increase in the availability is equal to the run-time energy improvement ($11.86 \times$), in practice it depends on two factors: The amount of excess energy ($Q_{\text{excess}}$) which determines the number of coupons, and the current energy level on the supercapacitor ($Q$) which determines how many of those coupons can be used. The figure plots reflect the potentials of the
system after the excess energy interval has passed and is converted into coupons. The $x$-axis represents the available energy on the supercapacitor from 0 to the maximum energy level of 710 mJ and the $y$-axis shows the maximum number of signatures that system can generate for the corresponding energy level. To see the full impact of precomputed execution on the complete energy spectrum, the system has to generate $\left\lfloor \left\lfloor \frac{\epsilon_{\text{max}}}{\epsilon} \right\rfloor \right\rfloor \frac{\epsilon}{\epsilon_0} \approx 5706$ mJ of energy. For an excess energy of 827 mJ, the processor can generate coupons for 12 hash-based signatures. This model assumes that the system is not bounded with memory, in practice this is also feasible since 128 KBs of available flash memory allows storing 12 coupons with Precomp-opt, pre.7. In this setting, to compute the first 12 signatures, Precomp and Precomp-opt require significantly less energy. Then, Mono-opt and Precomp-opt become equivalent but Precomp-opt still requires less energy to perform a signature due to its precomputation of the intermediate steps. $\Delta s$ corresponds to the difference of signature generation capability between different methods. From these results, we can clearly observe the effect of our optimization methods and moreover deduce that its impact is much more important for critical energy levels. For example, if a burst signing request comes when the energy level is 100 mJ, the Precomp-opt, Precomp, and Mono-opt can generate 11, 3, and 0 signatures respectively. This also shows that at certain times (critical energy intervals) the system can only generate a signature with an optimized execution.

5.5.4 Comparison with Related Work

There have been several implementations of hash-based post-quantum digital signatures. However, most of them are optimized for execution time, do not quantify the impact on energy, and none have the notion of precomputation with the of-
fline/online phases. Rohde et al. presented the first implementation of hash-based signatures on constrained devices [138]. Then, Hülsing et al. proposed several optimizations for W-OTS and Merkle-tree constructions, which can reduce the execution time [134], [139]. The fastest hash-based signature takes 41 ms on an 8-bit AVR microcontroller [140] but it uses a co-processor for acceleration. Since our methods are orthogonal transformations, it can also improve these results.

5.6 Conclusions

In this chapter, we investigated precomputation as a potential optimization strategy for post-quantum digital signatures on energy-harvested microcontrollers. We showed that it is possible to partition the operations of these signature schemes into an offline phase, precomputed before the input, and to an online phase, computed in response to the application input. We provided a generic implementation of such signatures utilizing precomputed executions. We demonstrate that today’s embedded low-cost platforms can effectively employ precomputation strategies for complex signature schemes. We first showed that precomputation can save significant runtime energy (up to $11.9 \times$) and can bring the execution time below one second mark. Then, we highlight its impact on the overall system and demonstrate that it can improve energy-efficiency by increasing the rate of output generation for a given energy profile.
Chapter 6

Area-efficient Post-quantum Signatures

This chapter presents methods based on precomputation and integration with reuse for area-efficient post-quantum cryptosystems. We apply proposed techniques on a lattice-based digital signature scheme and show that they can bring area-efficiency; for a given area, they can decrease the latency of output generation. The key idea is to partition the signature generation scheme into offline and online phases as in Chapter 5 and then to use an efficient hardware/software allocation for these phases. This mapping assigns complex precomputation operations to software on a compact microprocessor, and uses remaining hardware resources to accelerate timing-critical online operations. To find the optimum hardware architecture for the target platform, we define and explore the design space of online operations, and implement two design configurations. We realize our solutions on the Altera Cyclone-IV CGX150 FPGA. The implementation consists of a NIOS softcore processor and a
low-latency hash and polynomial multiplication engine. On average, the proposed low-latency architecture can generate a signature with a latency of 96 clock cycles at 40 MHz, resulting in a response time of 2.4 µs for a signing request. On equivalent platforms, compared to previous hardware and software implementations, this respectively corresponds to a performance improvement of 33× and 105×, and an area-efficiency improvement of 86× and 48×.

6.1 Introduction

Low latency encryption is required to secure embedded systems in domains such as automotive/aviation, optical links, and Internet of Things. The typical scenario is that the security feature of the device will activate ‘every once in a while’, but once triggered, it has to generate the response as quickly as possible. This is different from the traditional throughput goal that optimizes the maximum computations per second in a streamlined fashion. Knežević et al. recently investigated the latency of lightweight symmetric-key encryption engines [31]; Borghoff et al. proposed PRINCE to minimize the encryption execution time of block ciphers [141]. In this chapter, we investigate a public-key based digital signature generation scheme and propose techniques to minimize its latency for a given area-cost.

We first focus on the signature scheme proposed by Güneysu et al. [142], which was previously optimized for high-throughput. Then, we optimize the latency for the application scenario of real-time embedded systems where the latency of a response is more important than optimizing the hardware for operations per second. To achieve our goal, we apply precomputation and integration with reuse via a hardware/software codesign methodology. We first partition the signature generation
scheme into offline and online phases. The system produces and stores coupons during the offline phase that can be quickly spent to generate a valid signature at the online phase. The offline phase does not involve the message to be signed and its operations can be handled by a compact soft processor on FPGA. The online phase starts when a signature generation is requested on an input message. Generating a signature is accelerated by utilizing the parallelism of hardware. We explore the hardware architecture design space and identify several relevant solutions to minimize latency. This methodology enables scalable solutions and provides a significant area-efficiency improvement compared to previous work on similar platforms.

### 6.1.1 Novelty

The main contributions of this chapter are:

- To apply a latency-optimized computational model on a lattice-based signature scheme. Even et al. proposes a novel algorithm that can be used in an offline/online setting [143]. In contrast, we propose algorithmic transformations to retrofit our model into an existing post-quantum cryptographic scheme.

- To design a hardware/software co-designed system that minimizes the challenges of the signature scheme. The major drawback of lattice-based schemes is the probabilistic nature of the rejection sampling. There is always a possibility of not generating a valid signature for a signing request. However, the design of our system ensures that using the available precomputed coupons, this probability can be made arbitrarily small.

- To define the design space of sparse polynomial multiplication and to explore
it with a latency-optimization target for the given FPGA platform. The exploration yields several tradeoffs and a systematic approach for optimizing the available FPGA resources.

The rest of the chapter is organized as follows. Section 6.2 gives a high-level overview of the proposed techniques and its impact on area-efficiency. Section 6.3.3 introduces the fundamentals of lattice-based cryptography and presents previous work on lattice-based implementations. Section 6.4 describes the partitioning and the allocation of the operations into hardware and software components, and proposes the hardware architectures that optimize the design space for minimum-latency. Section 6.5 presents the implementation results and its comparison to previous work. Section 6.6 highlights some possible future extension of this work and concludes the chapter.

### 6.2 The Impact of Proposed Methods on Efficiency

Before going into the details of the proposed techniques and their applications to lattice-based signature generation, we give a high-level overview on why they work by using a basic example. Figure 6.1 demonstrates the Monolithic (a), Precomputed (b), Monolithic-parallel (c), and Precomputed-parallel solutions. For Monolithic and Precomputed cases, there is single computing unit $P_1$ so total area is equal to $P$. For Monolithic-parallel and Precomputed-parallel cases, there are 4 computing units so total area is equal to $4P$. In all examples, there are four distinct operations ($O_1$, $O_2$, $O_3$, and $O_4$) with equal workloads and with a dependency highlighted as in the figure. We assume that if an operation $O_i$ ($i \in 1, 2, 3, 4$) uses one computing unit $P$, it takes $1t$ to complete it. We also assume that computing units can accelerate computations with inverse proportion; if an operation $O_i$ uses $n$ computing units, it
Figure 6.1: Resource allocation for (a) Monolithic and (b) Precomputed (c) Monolithic-parallel (d) Precomputed-parallel. The operations colored in red ($O_1, O_2, O_3$) highlight offline operations and the green colored operation ($O_4$) is the online operation takes $(1/n) \times t$ to complete it.

The Monolithic solution in Figure 6.1 (a) uses the single computing unit without precomputation. Since it takes $1t$ to complete one operation, the total latency of this system is $4t$. The throughput of the system is also $4t$, as it can generate a new output at every $4t$ intervals.

The Precomputed solution in Figure 6.1 (b) applies precomputation. In this example, three operations can be precomputed during offline phase and online phase only has one operation. Therefore, the latency becomes $t$, while the throughput is
still $4t$. Indeed, this example was our precomputation method in Chapter 5, in which we define latency improvement as $1 + t_p/t_o$ where $t_p$ and $t_o$ is the execution time of precomputed and online operations, respectively.

The \textit{parallel} extensions represents the cases where the same operations can be ported to multicore systems or to an FPGA in which the designer has the flexibility to customize the hardware for multiple computing units. The \textbf{Monolithic-parallel} solution in Figure 6.1 (a) is the extension of monolithic execution with throughput optimizations into a system that can use 4 computing units. All operations are mapped to a separate computing unit with pipelined outputs and therefore the latency is $4t$ while the throughput is reduced to $t$.

The \textbf{Precomputed-parallel} solution in Figure 6.1 (a) is the extension of precomputed execution with latency optimizations into a system that can use 4 computing units. This system is optimized for latency both with precomputation and integration with reuse. After partitioning for precomputed executions, the system applies a smart allocation of the operations. The precomputed operations, which do not have an impact on the latency, reuses a single computing unit, while the three remaining computing units accelerate the online operation. This acceleration can triple the speed of the online computation and thus latency becomes $1/3t$. The throughput on the other hand, is determined by the bottleneck at $P_1$ which makes it $3t$. Note that, $P_1$ can also accelerate the online operation but it is omitted for simplicity.

This example illustrates the impact of clarifying the context, which is resource-efficiency for area and latency optimizations. Area-efficiency, defined as $1/(\text{Latency} \times \text{Area})$, can quantify the impact of the proposed methods, which increases it moving from \textbf{Monolithic} to \textbf{Precomputed}, and from \textbf{Monolithic-parallel} to \textbf{Precomputed-parallel}. 
Therefore, the proposed methods are indeed area-efficient because they reduce waste and allow to do more with the given resources. In the example, for the same area-cost, the Precomputed-parallel can increase the latency by a factor of 12 compared to Monolithic-parallel. Recall that, the precomputed execution offers $1 + \frac{t_p}{t_o}$ latency improvement, which is equal to 4 in this example. The additional factor of 3 comes from optimizing latency with smart allocations that apply integration with reuse for precomputed executions and that accelerate online operations with the rest of the resources.

We will show how to realize this example with hardware/software codesign on a complex lattice-based signature generation. A compact microprocessor which occupies $\approx 2.5\%$ of the logic resources will be responsible to implement all offline operations by reusing this area and the remaining resources will accelerate online operations.

Enabling these techniques have two main challenges: (1) Partitioning operations into offline ($O_1, O_2, O_3$) and online ($O_4$) phases and (2) finding the best strategy to accelerate $O_4$ with the remaining area. Section 6.4.1 and 6.4.2 addresses the first challenge and Section 6.4.3 tackles the second one.

### 6.3 Lattice-based Signatures

The family of lattice-based signatures is a promising group of candidates in the post-quantum era. Currently, lattice-based signatures that utilize Fiat-Shamir paradigm [144] yields the most efficient constructions. These constructions first introduce an identification scheme and then transform it to signatures.
The main challenge of lattice-based signatures is to minimize the signature size as well as public and secret keys. Figure 6.2 illustrates recent works on practical lattice-based digital signature schemes using the Fiat-Shamir transformation. Lyubashevski et al. proposed the corpus of work in this field and hence we will refer these schemes as Lyubashevski-like constructions. The figure shows that over a couple of years, using several optimization techniques such as, matrix to polynomial reduction [145], changing the basis of lattice problems [146], optimizing the parameter set [142], compression [142], [147], [148], and more efficient sampling [149] reduced the signature size down to 5 Kb. This size will probably even become smaller in the future as there is currently no theoretical limit on how small they eventually be.

We will focus on the Güneysu-Lyubashevsky-Pöppelmann (GLP) signature scheme presented in [142] due to its conceptual simplicity and reasonable signature size. However, the optimization methods that we propose apply to all Lyubashevski-like constructions and we can extrapolate our results to other lattice-based signature
schemes as well.

6.3.1 Background on Lattice Based Cryptography

Efficient lattice-based cryptographic primitives are defined over $\mathbb{Z}_p[x]/\langle x^n + 1 \rangle$ where the typical value of $n$ is 256, 512 or 1024 and the size of $p$ is 12-bits to 24-bits. Elements in $\mathbb{Z}_p[x]/\langle x^n + 1 \rangle$ can be represented as polynomials of degree $n - 1$ with coefficients modulo $p$. The underlying operations of $\mathbb{Z}_p[x]/\langle x^n + 1 \rangle$ are polynomial addition and polynomial multiplication. Polynomial addition is simply adding the polynomial coefficients of the same degree. However, polynomial multiplication uses the reduction function $x^n + 1$ to map the resulting polynomial of degree $2n - 2$ to a polynomial of degree $n - 1$. The Schoolbook method of polynomial multiplication is a computationally intensive operation that has a complexity of $O(n^2)$. However, NTT reduces the cost of polynomial multiplication to $O(n \log n)$. NTT is essentially a Discrete Fourier Transform defined over finite fields without the use of complex arithmetic [150]. The NTT exists if and only if $\exists t \ s.t. \ p = tn + 1, \omega^n = 1 \mod p$ and $\forall i < n, \omega^i \neq 1$. The values of $p$ and $n$ are chosen to satisfy these conditions. Moreover, the reduction function for practical lattice-based cryptography is $f(x) = x^n + 1$ and is chosen to make the reduction very simple.

6.3.2 GLP Signature Scheme

Algorithm 1 shows the signature generation procedure, proposed by Güneysu et al. [142]. The scheme combines the previous work of Lyubashevksi [151], [152]. This scheme requires sampling from a uniform distribution and the parameters are tuned for implementation efficiency. Generating a signature consists of two steps:
Key Generation and Signing. Note that all the elements except the message \( \mu \) are in \( \mathbb{Z}_p[x]/(x^n+1) \). \( R_p^{\alpha} \) represents polynomials of degree \( n-1 \) with the coefficients modulo \( p \) and \( R_p^{\alpha}_t \) denotes polynomials of degree \( n-1 \) with coefficients in the range \((-t,t)\). The underlying operations of the signature scheme are picking uniformly random polynomials (\( \text{rand()} \)), polynomial multiplication, polynomial addition, and hashing (\( H() \)).

**Listing 1** The basic signature scheme of [142]

```plaintext
1: procedure Key Generation\((a, s_1, s_2, t)\)
2: \[ s_1, s_2 \leftarrow \text{rand}(R_1^{p^n}) \]
3: \[ a \leftarrow \text{rand}(R^{p^n}) \]
4: \[ t \leftarrow as_1 + s_2 \]
5: end procedure

6: procedure Signing\((s_1, s_2, \mu, z_1, z_2, c)\)
7: \[ y_1, y_2 \leftarrow \text{rand}(R_k^{p^n}) \]
8: \[ c \leftarrow H(ay_1 + y_2, \mu) \]
9: \[ z_1 \leftarrow s_1c + y_1, z_2 \leftarrow s_2c + y_2 \]
10: if \( z_1 \) or \( z_2 \notin R_{k-32}^{p^n} \) go to step 7
11: end procedure

12: procedure Verification\((z_1, z_2, c, \mu, t, )\)
13: Validate iff
14: \[ z_1, z_2 \in R_{k-32}^{p^n} \]
15: \[ c = H(az_1 + z_2 + tc, \mu) \]
16: end procedure
```

Key Generation creates the secret signing key \( s_1, s_2 \), and the public verification key \( t \). Key Generation is an initialization phase and it is not performed every time
When a signing is requested, this process starts with selecting secret key polynomials $s_1, s_2$, from a subset of $\mathbb{F}_p[x] \langle x^n + 1 \rangle$ in which the coefficients are sampled uniformly random from $-1/0/1$. Then, the global parameter polynomial $a$ is initialized, with all its coefficients chosen uniformly at random in the range $[-(p - 1)/2, (p - 1)/2]$. To compute the verification key $t$, we have to perform a polynomial multiplication and a polynomial addition using the secret key. Given a number of verification keys $t$ and the global parameter $a$, there is no known algorithm to find the secret keys $s_1, s_2$ in sub-exponential time using either a traditional computer or a quantum computer, due to the Decisional Compact Knapsack problem [153].

Signing a message $\mu$ starts with randomly sampling two masking polynomials $y_1, y_2$ each having coefficients in the range $[-k, k]$. Then, the signer uses these polynomials to generate the nonce $ay_1 + y_2$ and hashes with the input message. The 160-bit digest of this hash is transformed into a sparse polynomial, by mapping each 5-bit segment of it into a polynomial that has 16 coefficients with only one being -1/1 and the rest being 0. These polynomials are then concatenated into the sparse polynomial $c$ that has 512 coefficients with only 32 of them being -1/1 and the rest being 0. Afterwards, the resulting sparse polynomial is multiplied with the secret key $s_1, s_2$, and added to the masking polynomials $y_1, y_2$ to generate the corresponding signature $z_1, z_2$. If all of the coefficients of the signature polynomials are within the bound $[-(k - 32), k - 32]$ it is regarded valid, else the complete signing process is repeated until this condition is satisfied. Verifying the signature starts by checking if step 7 in Listing 1 holds. Then, to validate a signature, the verifier can check if the equation $c = H(az_1 + z_2 + tc, \mu)$ holds.

We have used the parameters of Set I [142] where $n, p$, and $k$ are set to 512, 8383489, and $2^{14}$ respectively. Table 6.1 gives the summary of the parameters. Note
that the secret keys are represented with a simple notation. This work implements the basic version proposed in [142]. There is also an extension of this algorithm that is primarily used to reduce the signature size by compression. We have opted to use the basic version as our proof-of-concept implementation because compressing the signature size comes at the expense of further signature rejection and reduced security. Ducas et al. shows that the intended 100-bit security of the algorithm diminishes to 80-bits [149]. The main advantage of GLP signatures was the elimination of Gaussian sampling, however recent work showed that Gaussian sampling can be performed very efficiently [154], [155]. Therefore, latest lattice based signatures like Bimodal Lattice Signature Scheme (BLISS) propose very efficient constructions that can further reduce the signature size, the repetition rate [149], [156] and hence the execution time [154].

### Table 6.1: Signature scheme parameters

<table>
<thead>
<tr>
<th>Reference</th>
<th>Basic Scheme [142]</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>512</td>
</tr>
<tr>
<td>p</td>
<td>8383489</td>
</tr>
<tr>
<td>k</td>
<td>16384</td>
</tr>
<tr>
<td>Signature bit size</td>
<td>15520</td>
</tr>
<tr>
<td>Secret key bit size</td>
<td>2048</td>
</tr>
<tr>
<td>Public key bit size</td>
<td>11776</td>
</tr>
<tr>
<td>Expected number of repetitions</td>
<td>7</td>
</tr>
<tr>
<td>Bit Security</td>
<td>$\approx80$</td>
</tr>
</tbody>
</table>
Table 6.2: The implementations of the lattice-based cryptography. HE-µprocessor and LE-µprocessor stands for high-end and low-end microprocessor, respectively.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Operation</th>
<th>Optimization Target</th>
<th>Platform</th>
<th>Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>[157]</td>
<td>Arithmetic</td>
<td>Area</td>
<td>FPGA</td>
<td>Hardware</td>
</tr>
<tr>
<td>[13]</td>
<td>Arithmetic</td>
<td>Area-Efficiency</td>
<td>FPGA</td>
<td>Hardware</td>
</tr>
<tr>
<td>[34]</td>
<td>Arithmetic</td>
<td>Area</td>
<td>FPGA</td>
<td>Hardware</td>
</tr>
<tr>
<td>[158]</td>
<td>Arithmetic</td>
<td>Throughput</td>
<td>FPGA</td>
<td>Hardware</td>
</tr>
<tr>
<td>[159]</td>
<td>Arithmetic</td>
<td>Throughput</td>
<td>GPU</td>
<td>Software</td>
</tr>
<tr>
<td>[160]</td>
<td>Hash Function</td>
<td>Throughput</td>
<td>FPGA</td>
<td>Hardware</td>
</tr>
<tr>
<td>[161]</td>
<td>Public-key Encryption</td>
<td>Throughput</td>
<td>FPGA</td>
<td>Hardware</td>
</tr>
<tr>
<td>[162]</td>
<td>Public-key Encryption</td>
<td>Area-Efficiency</td>
<td>FPGA</td>
<td>Hardware</td>
</tr>
<tr>
<td>[163]</td>
<td>Public-key Encryption</td>
<td>Area</td>
<td>FPGA</td>
<td>Hardware</td>
</tr>
<tr>
<td>[142]</td>
<td>Digital Signature</td>
<td>Throughput</td>
<td>FPGA</td>
<td>Hardware</td>
</tr>
<tr>
<td>[154]</td>
<td>Digital Signature</td>
<td>Throughput</td>
<td>FPGA</td>
<td>Hardware</td>
</tr>
<tr>
<td>[164]</td>
<td>Digital Signature</td>
<td>Throughput</td>
<td>HE-µprocessor</td>
<td>Software</td>
</tr>
<tr>
<td>[165]</td>
<td>Digital Signature</td>
<td>Memory Footprint</td>
<td>LE-µprocessor</td>
<td>Software</td>
</tr>
<tr>
<td>[166]</td>
<td>Digital Signature</td>
<td>Throughput</td>
<td>LE-Microprocessor</td>
<td>Software</td>
</tr>
<tr>
<td><strong>This work</strong></td>
<td>Digital Signature</td>
<td>Latency</td>
<td>FPGA</td>
<td>Codesign</td>
</tr>
</tbody>
</table>

### 6.3.3 Related Work

Several implementations of lattice-based cryptographic primitives have been published in the recent years and Table 6.2 compares their main properties. Pöppelmann et al. introduced the first FPGA implementation of polynomial multiplication that uses NTT with a reduction function of \( f(x) = x^n + 1 \) [157]. This architecture is further optimized for area [13], [34]. Györfi et al. targets high-throughput hard-
ware architectures and their implementation on FPGAs [158]. Emeliyanenko showed that the graphics hardware can accelerate the NTT to 77 Giga multiplications per second [159].

Apart from the primitives, complete lattice-based cryptographic constructions also exist in the literature. Györfi et al. implements a hash function that uses lattice-based cryptography [160]. Göttert et al. presents a Ring Learning-with-Errors (LWE) based public-key encryption and its software and hardware implementations [161]. The public-key encryption is later optimized for high-throughput [162] and low-cost applications [163] both by Poppelmann et al. On the other hand, Güneysu et al. proposes a signature scheme that was built on recent publications of Lyubashevski [151], [152]. The proposed signature scheme was optimized for embedded systems and implemented on FPGAs using a schoolbook multiplier for polynomial multiplication [142]. Then, the same signature scheme was optimized for software realizations [164]. Most recently, Boorghany et al. shows a comparison of signature schemes on low-cost microcontrollers [165] and Oder et al. targets commodity microprocessors [166]. Finally, Poppelmann et al proposes an FPGA implementation of BLISS with improved Gaussian sampling [154].

6.4 Area-efficient Hardware/Software Codesign

This section presents our design strategy to achieve area-efficient and low-latency signature generation. First, we partition the operations for a precomputed execution based on their data dependencies. Then, by considering the application scenario, we allocate them into resource-efficient components which can utilize integration with reuse. Afterwards, we define and explore the design space, and we discuss the effect
of the design decisions on the throughput and latency. These design decisions reveal several trade-offs for different optimization targets. Finally, we will show how to design hardware components for the proposed optimization targets.

### 6.4.1 Partitioning the Signature Scheme

As in Chapter 5, precomputation exploits the fact that some operations of the signature scheme do not depend on the message to be signed and thus can be handled beforehand\(^1\). Figure 6.3 shows (a) the principle of operation for the monolithic vs. (b) the percomputed execution. Monolithic execution starts with picking a masking polynomial pair \(y_1, y_2\), applying all the steps of the algorithm and finally checking if the generated signature is valid. If it is invalid, it picks another polynomial pair.

---

\(^1\)The suitability of this observation for lattice-based signatures was first made by Lyubashevsky [152]
$y_1, y_2$ and repeats the same process until a valid signature is generated. In the pre-computed execution, during the first phase, we pick $i$ pairs of $y_1^i, y_2^i$ and apply the steps of the algorithm to generate corresponding $init^i$ values. When we are given a signing request for an input message, we transition to the second phase. During this phase, precomputed $init^i$ coupons generate the signature. If all $init^i$ coupons fail, then we have to extend the online phase with additional offline operations which will stall the message, recompute new $init^i$ values and attempt once more to generate a new signature. This process continues until a valid signature is generated. If we use a single polynomial pair $y_1, y_2$, the probability of generating a valid signature is approximately 0.135. Using multiple polynomial pairs, the probability of a valid signature rapidly increases. For example, with 256 precomputed pairs, the probability of success is $1 - (7 \times 10^{-17})$. If an even better probability is needed, additional pairs can be precomputed. Section 6.5.1 elaborates further on the yield.

### 6.4.2 Mapping of the Operations

The mapping of the offline/online partition into silicon resources exploits the fact that only the online phase is timing-critical. Therefore, we map the computationally intensive, message-independent, precomputation operations to a low-cost processor that can perform all these operations in a small area, and spend the silicon resources to accelerate easier online operations. Indeed, the offline phase operations that generate the keys, that sample the masking polynomials $y_1, y_2$, and that compute the NTT-based polynomial multiplication $ay_1$ are costly. However, the operations after receiving the message are relatively easier. These include hashing$^2$, multiplication of

$^2$Hashing is split between online and offline phases, a portion of $init^1$ is pre-hashed because of its size
Figure 6.4: Partition of signature generation to hardware and software operations.

Figure 6.4 shows the proposed partitioning. The software handles the offline phase computation for 256 polynomials and stores them in a buffer memory. Once a signature is requested for an input message, the hardware is responsible of computing the signature candidates (sequentially or in parallel) and check if a valid signature is generated. Note that once a masking polynomial pair $y_i$, $y_j$ is used for a valid signature, it should be discarded and a new pair should be generated. The proposed partitioning enables low-latency signature generation for real-time systems. In the results section, we elaborate further on the precomputation time and the latency of generating a signature after a signing request.
6.4.3 Multiplication of Polynomials with Small Coefficients

The polynomial multiplication of a small polynomial with a sparse polynomial is the most critical online operation that is handled by the hardware. Figure 6.5 shows the multiplication of \( t = sc \) where \( c \) is a sparse polynomial. The figure highlights an example case where \( c_0, c_{26}, ..., c_{495} \) is \(-1/1\). Since \( c \) is a sparse polynomial with only 32 non-zero coefficients (among 512) being \(-1/1\), the multiplication can be implemented with 32 shift and 31 add operations. The shift operation is negacyclic and variable-length as the reduction function has the form of \( x^n + 1 \). The shift amount depends on the position of the non-zero value within a 16-coefficient group (eg. \( c_{15...c0} \)). The add operation is simply summing the polynomials coefficient-wise and it does not even require a modulo operation since the bounds of the resulting coefficients are \([-32, 32]\).
Figure 6.6: (a) Design dimensions and (b) required operations for polynomial multiplication. The proposed architectures are shown in (a) are A, B and C with coordinates (512,16,1), (512,1,31) and (64,2,31) respectively in (coefficients, polynomials, additions).

6.4.4 Design Dimensions
In the design space of polynomial multiplication for the signature generation, several parallelism dimensions exist and a designer can opt for different parallelism choices within each dimension. Fig 6.6(a) illustrates a three-dimensional representation of these dimensions where the $x$, $y$, and $z$ axis respectively stands for the dimensions of parallelism for coefficients, additions and polynomials. We also show several hardware architectures that we have evaluated. In the first dimension, shown in $z$ axis, while computing a signature candidate, we can perform just 1 addition at a time up to doing all 31 additions at the same time. In the second dimension, shown in $x$ axis, we can parallelize the number of coefficients. This number can range from one coefficient to $n$-coefficients where $n$ is the size of the polynomial. In the third dimension, shown in $y$ axis, we can have multiple copies of the multiplication engine. Depending on the target speed, we can use one copy to $e$-copies of the multiplication engine, where $e$ is the maximum number of copies we can fit into our FPGA.

Depending on the parallelism choices at each dimension, the processing capability of the hardware implementation can range from $e$-parallel polynomial multiplications per clock cycle to one coefficient of one addition of one polynomial per clock cycle. The latency of one polynomial multiplication can also range from one clock cycle to $31 \times n = 15872$ clock cycles. We will give brief descriptions of these choices, but for the scope of this work, we only will investigate the hardware architecture for low-latency implementations.

**Parallelism of Polynomials (Y-axis)**

Parallelizing the multiplication engine is useful because generating a signature requires at least two polynomial multiplications, and it takes several trials (on average
7) before achieving a valid signature. The latency of the signature generation also reduces because parallel executions has a better chance of finding a valid signature in a shorter amount of time. Figure 6.6(b) shows a generic engine that can do multiple polynomial multiplications at the same time. The computed polynomial multiplications scales linearly with the number of engines used. The design entities A, B and C of Figure 6.6(a) use 16, 1, 2 engines, hence calculates 8, 1/2, 1 signature candidates at the same time, respectively.

**Parallelism of Additions (Z-axis)**

Within a polynomial multiplication engine, the parallelism choice ranges from computing 1 addition per clock cycle, up to 31 additions per clock cycle. In order to do all 31 additions at the same time, the hardware has to first apply the shift operations. Since this is a variable length negacyclic shifting (varying from 0 to 511), the parallelism at this dimension is costly. On the other hand, the parallelism at this dimension reduces latency, because instead of computing 31 additions sequentially in 31 steps, the hardware can do it just in 5 steps using an adder tree.

**Parallelism of Coefficients (X-axis)**

While computing a polynomial addition, we can process only 1 coefficient of the polynomial at one clock cycle up to \( n \) coefficients. Parallelism at this level linearly increases both the throughput and latency. If we can process \( cf \)-coefficients in one clock cycle, the computation of one polynomial addition takes \( \lceil \frac{n}{cf} \rceil \) clock cycles.

To generate one signature candidate we have to compute \( s_1 c + y_1 \) and \( s_2 c + y_2 \).
This is equal to $512 \times 31 \times 2$ (to generate $s_1 c, s_2 c$) + $512 \times 2$ (to add $y_1, y_2$) additions and $512 \times 2$ comparisons to check if the signature is valid. Since on average it takes 7 trials to generate a valid signature, the total number of operations is $512 \times 31 \times 14 + 512 \times 14$ additions and $512 \times 14$ comparisons. Ideally, we would like to do all these operations at the same time, but the resulting hardware does not fit into the target FPGA. Therefore the efficiency challenge is to maximize the parallelism that can achieve minimum latency. The target platform (Altera EP4CGX150) is a medium-cost platform and there exist high-end platforms that cost $10 \times$-$20 \times$ more and that provide $10 \times$-$20 \times$ more area which might do all operations in parallel.

### 6.4.5 Top-level Block Diagram

Figure 6.7 shows the block diagram of the proposed architecture. This diagram shows the generic architecture both for the design configurations A and C. During the initialization phase, NIOS processor first generates the keys $s_1, s_2$ and loads it to the registers $Reg s_1$ and $Reg s_2$. Next, it picks a $y_1^i, y_2^i$ masking polynomial pair and loads it to the BRAMs in $Mem y$. Later, NIOS computes the computationally intensive $init = ay_1^i + y_2^i$ value and stores it to the BRAMs in $Offline Hash Memory$. After an $init$ value is generated, it is hashed by the hash module and the output is stored back in the $Offline Hash Memory$ to be concatenated by the input message and hashed for the online phase. This process repeats for 256 polynomials. NIOS controls the memories with $Memory control$ module through an integrated custom instruction interface. Online phase starts after initialization is done for 256 polynomials. Once the online phase starts, the processor can also simultaneously execute to generate more fresh polynomials for the hardware.
Since the signature algorithm uses a 160-bit hash output to generate a sparse polynomial, the natural selection of the hash function is SHA-1. However, the proposed partition and optimization methodology can be used with other hash functions as well. We further optimized the SHA-1 by using the methods proposed by Lee et al. [167] and unroll it 1,20,5 times for design configurations A,B,C respectively to match the throughput of the polynomial multiplication. The output of the hash is simultaneously returned to a sparse polynomial and loaded into the polynomial multiplication engine. Finally, the output of the engine is compared with the Threshold $k$ and the validity of the generated signature is checked. We can set the value of $k$ from the NIOS processor which allows a flexible tradeoff between execution time vs.
Figure 6.8: Hardware architecture for the polynomial multiplier of design configuration A

forgeability of the signature.

6.4.6 Polynomial Multiplication

Design Configuration A (512 coefficients, 16 polynomials, 1 addition)

The design configuration A in Figure 6.6 uses a high-throughput polynomial multiplication to minimize the latency. Since this design does compute one polynomial addition at a time, we can simplify the shift operation. Figure 6.8 shows the resulting architecture. The shift operation is implemented using a parallel-in parallel-out fixed-length shift register. This reduces the cost of shift operation significantly compared to variable-length shift approach. First, the value of $s_i$ is loaded into the shift register. Then, at every clock cycle, the shift register shifts by 1, and if the input $c_i = '1'$ then the 512 coefficients are added at the same time. The output of the final addition ($31^{th}$) is transferred to a parallel-in serial-out shift register. Since the next
result will be ready after 512 clock cycles, the architecture can use only one adder and one comparator to add $y_i$ and to check if the generated signature is valid. The latency of one polynomial multiplication is 512 clock cycles. The design configuration A uses 16 of these blocks to compute 8 signature candidates at the same time.

**Design Configuration B (512 coefficients, 1 polynomial, 31 additions)**

The adder tree for 31 additions of one coefficient requires a total of 31 parallel adders arranged as 16, 8, 4, 2, and 1 of size 3-bit, 4-bit, 5-bit, 6-bit and 7-bit, respectively. Moreover, to generate one signature candidate, for each coefficient we also have to perform a 16-bit addition (to add $y_i$) and a comparison (to check if $z_i$ is valid). This number of operations times 512, to parallelize it for 512 coefficients, and the cost of shifting makes the design configuration too large to fit into our target FPGA. However, the design configuration B may still be implemented on larger high-cost FPGAs. The resulting architecture would generate a signature candidate in 2 clock cycles and a valid signature at approximately every 14 clock cycles after hashing.

**Design Configuration C (64 coefficients, 2 polynomials, 31 additions)**

Since the design configuration B does not fit into our target FPGA, we scale down in $x$-axis and implement the design point of (64,2,31). Because the shift operation is applied both for $s_1$ and $s_2$ using the same $c$ we also scale up one level in $y$-axis to improve efficiency. Figure 6.9 shows the hardware architecture of the design configuration C. It computes 64 coefficients in parallel, and thus, completes a signature candidate in 8 clock cycles. In the first cycle, the value of the polynomial $s$ is loaded into a shift register. At each clock cycle, the hardware applies the following steps to
produce 64 coefficients. First, it uses a 16-to-1 multiplexer and a Sign Computation block to obtain the $j$-th element of $i$-th partial product ($P[i][j]$). The multiplexer based selection results in a smaller area compared to the barrel shifter [168]. The multiplexer performs the selection by using the least significant four bits of the corresponding hash result ($c[i \times 5 + 3 : i \times 5]$), and the Sign Computation block computes the sign by using the most significant bit ($c[i \times 5 + 4]$). Then, an adder tree of 31 additions with a final 16-bit adder computes each coefficient candidate $j$. Finally, the configuration $C$ checks if the computed coefficients are valid by concurrently comparing them with $\pm (2^{14} - 32)$ via 64 comparator blocks. After each clock cycle, the shift register is rotated left by 64-bits. The signs of the rotated bits are inverted to apply the polynomial reduction function.
6.5 Implementation Results

We have implemented the proposed architectures using the Altera tool chain. The Design Space Explorer is used to get an optimized place and route result. For the design entity A and C, the maximum frequency is 112 MHz and 40 MHz respectively. Although an operating frequency of 40 MHz may look unsatisfactory at a first glance, it is primarily due to the parallelism in the z-dimension. Recall that our aim is to minimize latency, not throughput.

6.5.1 Yield

The lattice-based digital signature generation scheme that we have implemented does not have a deterministic execution time. Figure 6.10 shows how many trials it
takes to generate a valid signature. The probability of finding a valid signature is \( (1 - \frac{64}{2k+1})^{2n} \). On average, it takes 7 trials to generate a valid signature. Since we use 256 precomputed pairs of \( y_1^i, y_2^i \), the probability of not generating a valid signature is approximately \( 7.10^{-17} \). To put the likelihood into perspective, the probability of not generating a valid signature is 11 orders-of-magnitude less likely than the probability of an air-plane that performs the signature to crash [169]. Given more storage space, this probability can be made even smaller. If the system has to reply to burst signature requests, it can generate 36 signatures on average until it depletes all precomputed coupons. Then, it has to wait until the coupons are regenerated.

### 6.5.2 Cost and Latency Comparison

Due to different architectures, EDA tools, and optimization goals and efforts, a comparison is very hard to make on the basis of only the technology. Therefore, we find the best known implementations of GLP signature scheme, and we present an economically meaningful comparison for the area-cost efficiency of several designs. A similar method was used by Zimmermann et al. [170]. Table 6.3 shows our implementation results and its comparison to previous hardware and software realizations on platforms with similar costs. The selected FPGA, EPC4GLX150, costs approximately \( 1/3 \times \) and \( 2 \times \) of the hardware cost required for the previous-fastest hardware and software implementation, respectively. The results show that, compared to throughput optimized implementations, offline/online computational model drastically improves the latency by a factor of \( 33 \times \) and \( 105 \times \). The area-efficiency, defined as \( 1/(\text{Latency(s)} \times \text{area-cost(USD)}) \), is improved by a factor of \( 85.6 \times \) and \( 48.5 \times \).

As the execution time is not deterministic and as there is a separation of of-
Table 6.3: FPGA implementation results and comparisons with previous work

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Implementations</th>
<th>[164]</th>
<th>[142]</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td></td>
<td>Ivy Bridge</td>
<td>Virtex-6</td>
<td>Cyclone-IV</td>
<td>Cyclone-IV</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LX 130</td>
<td>CGX 150</td>
<td>CGX 150</td>
</tr>
<tr>
<td>Design Method</td>
<td></td>
<td>Software</td>
<td>Hardware</td>
<td>Codesign</td>
<td>Codesign</td>
</tr>
<tr>
<td>Price-2014 (USD)</td>
<td></td>
<td>225</td>
<td>1,267.5</td>
<td>488.6</td>
<td>488.6</td>
</tr>
<tr>
<td>Slice/Logic Element (S/LE)</td>
<td></td>
<td>-</td>
<td>19896 S</td>
<td>145224 LE</td>
<td>99672 LE</td>
</tr>
<tr>
<td>DSP</td>
<td></td>
<td>-</td>
<td>216</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>BRAM (18Kb)</td>
<td></td>
<td>-</td>
<td>234</td>
<td>274.5</td>
<td>270</td>
</tr>
<tr>
<td>Clock Frequency (MHz)</td>
<td></td>
<td>2500</td>
<td>204-416</td>
<td>112</td>
<td>40</td>
</tr>
<tr>
<td><strong>Latency (µs)</strong></td>
<td></td>
<td><strong>254</strong></td>
<td><strong>79.2</strong></td>
<td><strong>13.4</strong></td>
<td><strong>2.4</strong></td>
</tr>
<tr>
<td><strong>Efficiency (1/(s×USD))</strong></td>
<td></td>
<td><strong>17.49</strong></td>
<td><strong>9.96</strong></td>
<td><strong>152.73</strong></td>
<td><strong>852.77</strong></td>
</tr>
</tbody>
</table>

For fline and online computations, the classic throughput computation method is not suitable for our solution. However, we can still compute the latency of a single execution. To make a fair comparison, we used the same execution time calculation method of [142] and [164], which is \(\text{average number of trials required} \times \text{the execution time of one trial}\). The latency of the previous hardware implementation is estimated optimistically from the results of [142] and probably is worse than this estimation due to heavy pipelining of the design. For the design configuration A, the latency of generating a signature is 1505 (480 hash + 512 polynomial multiplication and + 512 addition) cycles with a clock frequency of 112 MHz, and hence, computing a signature takes 13.4 µs. The latency of the design entity C is much better. It takes 96 clock cycles (40 hash + 56 polynomial multiplication) on average to gener-
A signature, and at a clock frequency of 40 MHz a valid signature is generated in 2.4 $\mu$s. The proposed solutions of A and C cost approximately 145K and 100K logic elements (LE), and 274.5 and 270 BRAMs, respectively. Both architectures use 4 Digital Signal Processor (DSP) blocks to implement the NIOS processor. The initialization phase takes approximately 0.3s ($\approx30$M clock cycles @ 100MHz) for every polynomial (coupon) using the low-cost NIOS soft-core processor. The software performs sampling from a uniform distribution, NTT based polynomial multiplication and hashing. If required, the offline phase can also be accelerated further by multi-processors each computing independent coupons.

Table 6.4 gives the resource usage breakdown. The design configuration A barely fits into the device due to its heavy polynomial-level parallelism. Since the design configuration A computes one coefficient of a polynomial at a time, the addition of $y_1^i$, $y_2^i$ polynomials and checking the validity of $z_1^i$, $z_2^i$ costs marginal and is integrated into the sparse polynomial engine. In contrast, these operations cost around 25% of the total area of design configuration C. Registers to store $s_1$ and $s_2$ are also mapped into the sparse multiplication block. Both design configurations utilize a NIO2-II-f softcore processor to execute software.

To understand the baseline performance, we also map the online operations using C language to the NIOS-II soft processor that also performs the offline computations. The computation of a single signature candidate takes 897034 clock cycles. Clocked at 100 MHz, it would on average take 9 milliseconds (ms) $\times 7 = 63$ ms to generate a signature. This value can be improved using a more efficient or assembly level software, or through processor architecture optimization. If the software performance does not meet the design goal, then a designer can use our design space and the detailed implementation results to estimate the cost of the required hardware.
Table 6.4: Resource usage breakdown

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>LC Combinational</th>
<th>LC Registers</th>
<th>BRAMs (M9K)</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparse multiplication</td>
<td>137526</td>
<td>117019</td>
<td>521</td>
<td>0</td>
</tr>
<tr>
<td>y_addition and z_checking</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Hashing</td>
<td>2575</td>
<td>2596</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>NIOS processor</td>
<td>3995</td>
<td>2903</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>144096</strong></td>
<td><strong>122518</strong></td>
<td><strong>547</strong></td>
<td><strong>4</strong></td>
</tr>
<tr>
<td>Sparse multiplication</td>
<td>56063</td>
<td>4219</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>y_addition and z_checking</td>
<td>21381</td>
<td>18447</td>
<td>514</td>
<td>0</td>
</tr>
<tr>
<td>Hashing</td>
<td>5409</td>
<td>3584</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>NIOS processor</td>
<td>3995</td>
<td>2903</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>86848</strong></td>
<td><strong>29153</strong></td>
<td><strong>540</strong></td>
<td><strong>4</strong></td>
</tr>
</tbody>
</table>

acceleration.

The results also show that the lattice-based signatures are not only quantum-secure but they can also be quite competitive compared to the traditional schemes. Gaj et al. proposes a hardware architecture on the XC2000e FPGA that can do 163-bit elliptic curve multiplication in 48 $\mu$s [171]. Then, it was optimized further by Rebeiro et al. to 8.6 $\mu$s on an XC5VLX85t FPGA [172]. Kong et al. presents a 2048-bit modular exponentiation implementation on an XC6VLX760 that has a latency of 0.35 $\mu$s, however the proposed FPGA costs 30 more than our target FPGA [173].


6.6 Conclusion and Future Work

In this chapter, we applied precomputation and integration with reuse to enable latency-optimized and area-efficient modern cryptosystems. We used the precomputation methods discussed first in Chapter 5 and then we proposed a hardware/software co-design with area-efficient partitions. This partitioning is based on integration with reuse; a compact softcore processor implements resource-consuming but latency-uncritical operations in a small area, allowing more resources to accelerate the latency-critical operations. We defined the design space and explored several points for the hardware acceleration. The results show that when a signature is requested, on average the proposed implementation can generate it with a latency of 2.4 $\mu$s. Our system is scalable and it can be employed on embedded platforms for real-time applications.

The latency of the hashing contributes significantly to the latency of the signature scheme, therefore a possible extension is to implement this system with a low-latency hash function. With a proper mode of operation, the PRINCE block cipher [141] or a modified version of it (since it was optimized for ASIC not FPGAs) can be transformed into a hash and be used to this end. The polynomial multiplication is executed in constant time and does not leak information for timing-based attacks. However, there could still be vulnerabilities against power-based side-channel attacks and one possible future work could be testing the strength of the implementation against these attacks. Finally, we observe that the signature verification does not allow partitioning of operations into precomputation and online phases. The verifier has to compute the computationally intensive polynomial multiplication with the large coefficients. Therefore, low-latency verification of quantum-resistant signatures
still remains an open problem for real-time applications.
Chapter 7

Conclusions and Future Work

This dissertation studies resource-constrained and resource-efficient modern cryptosystem design. Our results provide a better understanding of the cryptographic engineering challenges and opportunities of modern cryptosystems. We have identified three recurring strategies, (i) serialization, (ii) integration with reuse, and (iii) precomputation. We demonstrate how to fine-tune these strategies on a number of modern cryptographic primitives including key generation mechanisms, symmetric-key primitives, authentication protocols, and digital signatures. We showed that designing for resource-constraints is different than designing for resource-efficiency. We validate that by clarifying the application context and by using the recurring strategies, it is possible to achieve drastic improvements of up to two orders of magnitudes over previous approaches.

In light of our research, we make three key observations for the implementations of real-world modern cryptographic systems. First, such cryptographic systems are composed of multiple components, yet this aspect is often disregarded in many pro-
posals. Chapter 3 and Chapter 4 demonstrates that thinking beyond the boundaries of a single building block reveals new system-level and cross-component opportunities. Second, the proposed solutions are subject to change because the algorithms of interests are very new and not yet standardized. This is the main reason for defining the design space (as in Chapter 2 and Chapter 6) and highlighting the recurring strategies, so that it can extend an arguably-ephemeral point-solution into a long-lasting contribution. This approach allows applying the same set of principles and techniques on the future iterations of the target algorithms. Third, the current solutions can be drastically optimized. Especially for the complex public-key post-quantum cryptosystems, significant savings in key performance indicators are imperative to employ them on low-cost platforms. Chapter 5 and Chapter 6 quantifies how much improvement was possible but previously unknown. Next, we iterate on the conclusions of each chapter.

Table 7.1 summarizes the context, applied strategies, and the improvement factors of the chapters. The context of Part I, which is composed of Chapter 2, 3, and 4, is resource-constraints for area. Chapter 2 confirms that by applying serialization on a modern block cipher design, it is possible to achieve the smallest block cipher ever proposed in the literature of cryptographic engineering at 128-bit level security. It explores the bit-serial design methods for modern block ciphers and presents a novel hardware architecture with bit-serialization. Chapter 3 demonstrates that modern cryptosystems for authentication protocols need to perform several types of operations, and the challenges to build such a compact cryptosystem can not be met alone by solely focusing on individual components. It proposes a possible solution to this problem through an ASIP design which has serialized execution units based on the block cipher in Chapter 2. This execution unit is reused for multiple opera-
The context of Part II, which is composed of Chapter 5 and 6, is resource-efficiency. Chapter 5 tackles the problem of realizing complex security solutions and the ASIP offers a multi-purpose design that is able to compute multiple PUF protocols each having different types of building blocks. Chapter 4 investigates systems that have multiple on-board components and that require authenticating all of these elements using the PUF technology. It proposes a novel method for such resource-constrained systems: a compact PUF Fusion technique that can reuse a simplified error coding mechanism to integrate multiple PUF components without degrading authentication requirements.
for low-cost energy-harvesting platforms. It quantifies that precomputation is a very effective and applicable strategy on such embedded platforms, and that pre-computation improves the energy-efficiency of a complex post-quantum signature scheme. Chapter 6 addresses the challenges of realizing low-latency security solutions for hardware-designed platforms. It applies the precomputation of Chapter 5 and couples it with an integration with reuse strategy to improve the area-efficiency of a complex post-quantum signature scheme. The proposed solution is a hardware/software codesign with area-efficient partitions; a compact microprocessor implements resource-consuming percomputed operations in a small area, while the remaining area-resources accelerate the run-time operations.

7.1 Future Research Directions

In the conclusion section of each chapter, we comment on a few possible incremental steps, if any, to broaden the spectrum of that particular effort. Here, we describe more general future research directions.

- **Implementation Attacks on Modern Cryptosystems:** Implementation attacks such as side-channels and fault attacks have been thoroughly investigated and still an ongoing research direction for well-established cryptographic primitives like as AES, RSA, and ECC. For modern cryptographic primitives we target like PUFs, SIMON, and post-quantum signatures, there is limited, if any, research on implementation attack and countermeasure aspects. The systems we described in this dissertation also do not consider implementation attacks. Nevertheless, it is a crucial element if we want to use these primitives.
in real-world embedded systems because they are always susceptible to such attacks when they are deployed in an open environment.

- **Design Space Exploration for Performance/Area/Risk Optimality:** Previously, unrolling, a design method to improve the performance, was explored for its impact on side-channel analysis [174]. Likewise, Ghalaty et al. recently studied fault-based side channel variation of block ciphers in two configurations: round-serial and nibble-serial. However, as we discussed in Section 2.4 block ciphers have a large design space (ranging from 1-bit 1-round 1-encryption per clock cycles to single-cycle multi-encryption architectures). Likewise, as we discussed in Section 6.4.4, there is a large design-space for polynomial multiplication. It is not clear if there is a direct or linear correlation between the performance and side-channel risks. For example, previous work showed that such a linear correlation does not exists for key size of Keccak and its power-based side-channel leakage [175]. If side-channel risks change with the three proposed strategies, then, searching the design space for optimal ‘performance/area/risk’ configurations would be a valuable contribution.

- **System-level Security Evaluations and Countermeasures:** Another important future direction is to evaluate the system-level security aspects of modern cryptosystems. Recent work started to focus on design and analysis of protocols for Transport Layer Security [176], [177] and on anonymous Onion-Routing networks of Tor [178]. Our proposed methods can introduce additional security vulnerabilities to such systems, hence it is also important to analyze them. For example, precomputation in energy-harvesting systems uses coupons and enforces the system to store coupons in a non-volatile memory. These coupons correlate with the secret key value. Thus, the system has to in-
corporate a secure memory element to store coupons or else it has to obfuscate the memory access. A similar concern also applies for PUF based protocols we used, which assumes that PUFs have privileged access that is only used during the enrolment phase of the authentication protocols. Providing this solution for a low-cost board with minimal trusted computing base and with multiple PUF components is far from trivial.

- **Technology Mapping:** We have mapped our solutions on FPGA boards and on 16-bit MSP430 microcontrollers. An important aspect, which is not covered in this dissertation, is to port them on other platforms like 8-bit AVR microcontrollers, 32-bit embedded ARM microcontrollers, or ASIC. This could show how well the impact of the proposed methods change from one platform to the other.
Appendices
Appendix A

Detailed Protocol Mappings

In this section, we demonstrate the mapping details of the AGMSY’15 [5] and the modified reverseFE [74] protocols to our ASIP design.

A.1 Mapping of ReverseFE Protocol

Figure A.1 shows the steps of the modified reverseFE protocol [74] and Table A.1 gives the details of its mapping to the proposed ASIP. X represents the Don’t Care condition, for simplicity, they are reset to ‘0’. The data memory for the reverseFE protocol is organized as below. Note that an address in instruction and data memory holds 26- and 16-bits, respectively. [a-b] refers to the memory location between addresses a and b. $\delta(x:y)$ refers to the subset of bits of $\delta$ from $x$ to $y$.

[0-23]: $y'_i$; [24-29]: $\delta$ Random input of code-offset; [30-32]: BCH Output 1; [33]: $\delta(15:0)$; [34-36]: BCH Output 2; [37]: $\delta(31:16)$; [38-40]: BCH Output 3; [41]:
Setup Phase

Server (DB)  \[ ID_i, y_i \]
\[ puf_i \rightarrow y_i \]
Device (ID_i,puf)

Authentication Phase

Server (DB)

\[ ID_i, \omega_i, r_1 \]
\[ y_i'' := \text{Recover}(y_i, \omega_i) \]
\[ r_2 \leftarrow \{0, 1\}^l \]
\[ u_1 := \text{Hash}(ID_i, w_i, y_i'', r_1, r_2) \]
\[ u_2 := \text{Hash}(ID_i, y_i', r_2) \]
\[ \text{Hash}(ID_i, w_i, y_i', r_1, r_2) \overset{?}{=} u_1 \]
If no match, Abort

\[ u_2 := \text{Hash}(ID_i, y_i', r_2) \]
\[ \text{Hash}(ID_i, w_i, y_i', r_1, r_2) \overset{?}{=} u_1 \]
If no match, Abort

Figure A.1: The modified reverseFE protocol [74]

δ(47:32); [42-44]: BCH Output 4; [45]: δ(63:48); [46-48]: BCH Output 5; [49]: δ(79:64); [50-52]: BCH Output 6; [43]: δ(95:80); [54-77]: \( \omega_i \) Helper data of Code offset output; [78-83]: \( r_1 \); [84-95]: \( u_1 \) Received from verifier; [96-101]: \( r_2 \); [102-107]: \( ID_i \); [108-119]: \( u_1 \) Generated on the device; [120-131]: \( u_2 \);

<table>
<thead>
<tr>
<th>Protocol Step</th>
<th>Assembly Code (Mnemonic)</th>
<th>Read Offset (Dec.)</th>
<th>Write Offset (Dec.)</th>
<th>Opcode (Dec.)</th>
<th>Machine Code (Hex.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>IDLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>puf(_i) \rightarrow y'_i</td>
<td>ENTRPY.Stat.96x4</td>
<td>X</td>
<td>0</td>
<td>18</td>
<td>00000012</td>
</tr>
<tr>
<td>( \omega_i := y'_i \cdot H^T )</td>
<td>ENTRPY.Dyn.96</td>
<td>X</td>
<td>24</td>
<td>19</td>
<td>0000613</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>24</td>
<td>33</td>
<td>21</td>
<td>0180855</td>
<td></td>
</tr>
<tr>
<td>BCH.Gen</td>
<td>33</td>
<td>30</td>
<td>9</td>
<td>0210789</td>
<td></td>
</tr>
<tr>
<td>MEMCPY</td>
<td>25</td>
<td>37</td>
<td>21</td>
<td>0190955</td>
<td></td>
</tr>
</tbody>
</table>
\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{BCH.Gen} & 37 & 34 & 9 & 0250889 \\
\text{MEMCPY} & 26 & 41 & 21 & 01A0A55 \\
\text{BCH.Gen} & 41 & 38 & 9 & 0290989 \\
\text{MEMCPY} & 27 & 45 & 21 & 01B0B55 \\
\text{BCH.Gen} & 45 & 42 & 9 & 02D0A89 \\
\text{MEMCPY} & 28 & 49 & 21 & 01C0C55 \\
\text{BCH.Gen} & 49 & 46 & 9 & 0310B89 \\
\text{MEMCPY} & 29 & 53 & 21 & 01D0D55 \\
\text{BCH.Gen} & 53 & 50 & 9 & 0350C89 \\
\text{XOR.L1.64} & 0 & X & 12 & 000000C \\
\text{XOR.L2.64} & 30 & 54 & 13 & 01E0D8D \\
\text{XOR.L1.64} & 4 & X & 12 & 004000C \\
\text{XOR.L2.64} & 34 & 58 & 13 & 0220E8D \\
\text{XOR.L1.64} & 8 & X & 12 & 008000C \\
\text{XOR.L2.64} & 38 & 62 & 13 & 0260F8D \\
\text{XOR.L1.64} & 12 & X & 12 & 00C000C \\
\text{XOR.L2.64} & 42 & 66 & 13 & 02A108D \\
\text{XOR.L1.64} & 16 & X & 12 & 010000C \\
\text{XOR.L2.64} & 46 & 70 & 13 & 02E118D \\
\text{XOR.L1.64} & 20 & X & 12 & 014000C \\
\text{XOR.L2.64} & 50 & 74 & 13 & 032128D \\
\multicolumn{5}{|l|}{r_1 \leftarrow \{0, 1\}^l} \\
\multicolumn{5}{|l|}{\text{ENTRPY.Dyn.96} \ X \ 78 \ 19 \ 0001393} \\
\multicolumn{5}{|l|}{\text{SRAM.SEND} \ 102 \ X \ 0000017} \\
\multicolumn{5}{|l|}{\text{SRAM.SEND} \ 102 \ 54 \ 0360017} \\
\hline
\end{array}
\]
<table>
<thead>
<tr>
<th></th>
<th>SRAM.SEND</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>102</td>
<td>60</td>
<td></td>
<td>03C0017</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>66</td>
<td></td>
<td>0420017</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>72</td>
<td></td>
<td>0480017</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>78</td>
<td></td>
<td>04E0017</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>u1,r2</td>
</tr>
<tr>
<td></td>
<td>SRAM.RCV</td>
<td>84</td>
<td>X</td>
<td>0001518</td>
</tr>
<tr>
<td></td>
<td>SRAM.RCV</td>
<td>90</td>
<td>X</td>
<td>0001698</td>
</tr>
<tr>
<td></td>
<td>SRAM.RCV</td>
<td>96</td>
<td>X</td>
<td>0001818</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHS</td>
<td>102</td>
<td>X</td>
<td>0660001</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>105</td>
<td>X</td>
<td>0690002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>54</td>
<td>X</td>
<td>0360002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>57</td>
<td>X</td>
<td>0390002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>60</td>
<td>X</td>
<td>03C0002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>63</td>
<td>X</td>
<td>03F0002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>66</td>
<td>X</td>
<td>0420002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>69</td>
<td>X</td>
<td>0450002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>72</td>
<td>X</td>
<td>0480002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>75</td>
<td>X</td>
<td>04B0002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>0</td>
<td>X</td>
<td>0000002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>3</td>
<td>X</td>
<td>0030002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>6</td>
<td>X</td>
<td>0060002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>9</td>
<td>X</td>
<td>0090002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>12</td>
<td>X</td>
<td>00C0002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>15</td>
<td>X</td>
<td>00F0002</td>
</tr>
<tr>
<td></td>
<td>SIMON.HASHR</td>
<td>18</td>
<td>X</td>
<td>0120002</td>
</tr>
</tbody>
</table>
\[ u_2 := \text{Hash}(ID_i, y_i', r_2) \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMON.HASHR</td>
<td>21</td>
<td>X</td>
<td>2</td>
<td>0150002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>78</td>
<td>X</td>
<td>2</td>
<td>04E0002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>81</td>
<td>X</td>
<td>2</td>
<td>0510002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>96</td>
<td>X</td>
<td>2</td>
<td>0600002</td>
</tr>
<tr>
<td>SIMON.HASHF</td>
<td>99</td>
<td>108</td>
<td>3</td>
<td>0631B03</td>
</tr>
<tr>
<td>CHECKJMP.L1</td>
<td>84</td>
<td>X</td>
<td>14</td>
<td>054000E</td>
</tr>
<tr>
<td>CHECKJMP.L2</td>
<td>108</td>
<td>0</td>
<td>15</td>
<td>06C000F</td>
</tr>
<tr>
<td>CHECKJMP.L1</td>
<td>90</td>
<td>X</td>
<td>14</td>
<td>05A000E</td>
</tr>
<tr>
<td>CHECKJMP.L2</td>
<td>114</td>
<td>0</td>
<td>15</td>
<td>072000F</td>
</tr>
<tr>
<td>SIMON.HASHS</td>
<td>102</td>
<td>X</td>
<td>1</td>
<td>0660001</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>105</td>
<td>X</td>
<td>2</td>
<td>0690002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>0</td>
<td>X</td>
<td>2</td>
<td>0000002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>3</td>
<td>X</td>
<td>2</td>
<td>0030002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>6</td>
<td>X</td>
<td>2</td>
<td>0060002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>9</td>
<td>X</td>
<td>2</td>
<td>0090002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>12</td>
<td>X</td>
<td>2</td>
<td>00C0002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>15</td>
<td>X</td>
<td>2</td>
<td>00F0002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>18</td>
<td>X</td>
<td>2</td>
<td>0120002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>21</td>
<td>X</td>
<td>2</td>
<td>0150002</td>
</tr>
<tr>
<td>SIMON.HASHR</td>
<td>96</td>
<td>X</td>
<td>2</td>
<td>0600002</td>
</tr>
<tr>
<td>SIMON.HASHF</td>
<td>99</td>
<td>X</td>
<td>2</td>
<td>0631E03</td>
</tr>
<tr>
<td>SRAM.Send</td>
<td>120</td>
<td>X</td>
<td>23</td>
<td>0780017</td>
</tr>
<tr>
<td>SRAM.Send</td>
<td>126</td>
<td>X</td>
<td>23</td>
<td>07E0017</td>
</tr>
<tr>
<td>JMP</td>
<td>X</td>
<td>0</td>
<td>16</td>
<td>0000010</td>
</tr>
</tbody>
</table>
A.2 Mapping of AGMSY’15 Protocol

Figure A.2 shows the steps of the modified reverseFE protocol [5] and Table A.2 gives the details of its mapping to the proposed ASIP. X represents the Don’t Care condition, for simplicity, they are reset to ‘0’. The data memory for the ReverseFE protocol is organized as below. Note that, an address in instruction and data memory holds 26- and 16-bits, respectively. [a-b] refers to the memory location between addresses a and b. δ(x:y) refers to the subset of bits of δ from x to y.

<table>
<thead>
<tr>
<th>Protocol Step</th>
<th>Assembly Code (Mnemonic)</th>
<th>Read Offset (Dec.)</th>
<th>Write Offset (Dec.)</th>
<th>Opcode Offset (Dec.)</th>
<th>Machine Code (Hex.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>IDLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000000</td>
</tr>
</tbody>
</table>
\( z'_1 \leftarrow_R f(x_i, y_1) \) & ENTROPY.Stat.96x4 & X & 12 & 18 & 0000312 \\
\( \delta, \text{rnd}, y'_2 \leftarrow_R 0, 1 \) & ENTROPY.Dyn.96x4 & X & 36 & 20 & 0000914 \\
r_1 := G(sk, z'_1 \text{rnd}) & SIMON.PRF1 & 12 & X & 7 & 00C0007 \\
SIMON.KeyAndGo & 0 & X & 5 & 0000005 \\
SIMON.PRF2 & 18 & X & 8 & 0120008 \\
SIMON.KeyAndGo & 0 & X & 5 & 0000005 \\
SIMON.PRF2 & 24 & X & 8 & 0180008 \\
SIMON.KeyAndGo & 0 & X & 5 & 0000005 \\
SIMON.PRF2 & 30 & X & 8 & 01E0008 \\
SIMON.KeyAndGo & 0 & X & 5 & 0000005 \\
SIMON.PRF2 & 42 & X & 8 & 02A0008 \\
SIMON.KeyAndGo & 0 & X & 5 & 0000005 \\
SIMON.PRF2 & 48 & X & 8 & 0300008 \\
SIMON.KeyAndGo & 0 & X & 5 & 0000005 \\
SIMON.PRF2 & 60 & X & 8 & 03C0008 \\
SIMON.KeyAndGo & 0 & X & 5 & 0000005 \\
SIMON.CBC & 66 & X & 6 & 0420006 \\
SIMON.KeyAndGo & 0 & 72 & 5 & 0001205 \\
h d_1 := \text{FE.Gen}(z'_1, \delta) & \text{MEMCOPY} & 36 & 81 & 21 & 0241455 \\
& \text{BCH.Gen} & 81 & 78 & 9 & 0511389 \\
& \text{MEMCOPY} & 37 & 85 & 21 & 0251555 \\
& \text{BCH.Gen} & 85 & 82 & 9 & 0551489 \\
& \text{MEMCOPY} & 38 & 89 & 21 & 0261655 \\
& \text{BCH.Gen} & 89 & 86 & 9 & 0591589
<table>
<thead>
<tr>
<th>Function</th>
<th>1.</th>
<th>2.</th>
<th>3.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMCPY</td>
<td>39</td>
<td>93</td>
<td>21</td>
<td>0271755</td>
</tr>
<tr>
<td>BCH.Gen</td>
<td>93</td>
<td>90</td>
<td>9</td>
<td>05D1689</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>40</td>
<td>97</td>
<td>21</td>
<td>0281855</td>
</tr>
<tr>
<td>BCH.Gen</td>
<td>97</td>
<td>94</td>
<td>9</td>
<td>0611789</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>41</td>
<td>101</td>
<td>21</td>
<td>0291955</td>
</tr>
<tr>
<td>BCH.Gen</td>
<td>101</td>
<td>98</td>
<td>9</td>
<td>0651889</td>
</tr>
<tr>
<td>XOR.L1.64</td>
<td>12</td>
<td>X</td>
<td>12</td>
<td>00C000C</td>
</tr>
<tr>
<td>XOR.L2.64</td>
<td>78</td>
<td>102</td>
<td>13</td>
<td>04E198D</td>
</tr>
<tr>
<td>XOR.L1.64</td>
<td>16</td>
<td>X</td>
<td>12</td>
<td>010000C</td>
</tr>
<tr>
<td>XOR.L2.64</td>
<td>82</td>
<td>106</td>
<td>13</td>
<td>0521A8D</td>
</tr>
<tr>
<td>XOR.L1.64</td>
<td>20</td>
<td>X</td>
<td>12</td>
<td>014000C</td>
</tr>
<tr>
<td>XOR.L2.64</td>
<td>86</td>
<td>110</td>
<td>13</td>
<td>0561B8D</td>
</tr>
<tr>
<td>XOR.L1.64</td>
<td>24</td>
<td>X</td>
<td>12</td>
<td>018000C</td>
</tr>
<tr>
<td>XOR.L2.64</td>
<td>90</td>
<td>114</td>
<td>13</td>
<td>05A1C8D</td>
</tr>
<tr>
<td>XOR.L1.64</td>
<td>28</td>
<td>X</td>
<td>12</td>
<td>01C000C</td>
</tr>
<tr>
<td>XOR.L2.64</td>
<td>94</td>
<td>118</td>
<td>13</td>
<td>05E1D8D</td>
</tr>
<tr>
<td>XOR.L1.64</td>
<td>32</td>
<td>X</td>
<td>12</td>
<td>020000C</td>
</tr>
<tr>
<td>XOR.L2.64</td>
<td>98</td>
<td>122</td>
<td>13</td>
<td>0621E8D</td>
</tr>
<tr>
<td>$c := \text{SKE.ENC}(sk, hd)$</td>
<td>SIMON.ECB</td>
<td>102</td>
<td>X</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>SIMON.KeyAndGo</td>
<td>6</td>
<td>126</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>SIMON.CBC</td>
<td>108</td>
<td>X</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>SIMON.KeyAndGo</td>
<td>6</td>
<td>132</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>SIMON.CBC</td>
<td>114</td>
<td>X</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>SIMON.KeyAndGo</td>
<td>6</td>
<td>138</td>
<td>5</td>
</tr>
</tbody>
</table>
\[ y'_1 \]

\( (t_1, \ldots, t_5) := G(r_1, y'_1 y'_2) \)

<table>
<thead>
<tr>
<th>SIMON.CBC</th>
<th>120</th>
<th>X</th>
<th>6</th>
<th>0780006</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMON.KeyAndGo</td>
<td>6</td>
<td>144</td>
<td>5</td>
<td>0062405</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>42</td>
<td>X</td>
<td>6</td>
<td>02A0006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>6</td>
<td>150</td>
<td>5</td>
<td>0062585</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>48</td>
<td>X</td>
<td>6</td>
<td>0300006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>6</td>
<td>156</td>
<td>5</td>
<td>0062705</td>
</tr>
<tr>
<td>y'_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRAM.Rec</td>
<td>X</td>
<td>162</td>
<td></td>
<td>0002880</td>
</tr>
<tr>
<td>SIMON.PRF1</td>
<td>162</td>
<td>X</td>
<td>7</td>
<td>0A20007</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>X</td>
<td>5</td>
<td>0480005</td>
</tr>
<tr>
<td>SIMON.PRF2</td>
<td>54</td>
<td>X</td>
<td>8</td>
<td>0360008</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>X</td>
<td>5</td>
<td>0480005</td>
</tr>
<tr>
<td>SIMON.PRF2</td>
<td>168</td>
<td>X</td>
<td>8</td>
<td>0A80008</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>X</td>
<td>5</td>
<td>0480005</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>174</td>
<td>X</td>
<td>6</td>
<td>0AE0006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>228</td>
<td>5</td>
<td>0483905</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>180</td>
<td>X</td>
<td>6</td>
<td>0B40006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>234</td>
<td>5</td>
<td>0483A85</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>186</td>
<td>X</td>
<td>6</td>
<td>0BA0006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>240</td>
<td>5</td>
<td>0483C05</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>192</td>
<td>X</td>
<td>6</td>
<td>0C00006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>246</td>
<td>5</td>
<td>0483D85</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>198</td>
<td>X</td>
<td>6</td>
<td>0C60006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>252</td>
<td>5</td>
<td>0483F05</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>204</td>
<td>X</td>
<td>6</td>
<td>0CC0006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>258</td>
<td>5</td>
<td>0484085</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>210</td>
<td>X</td>
<td>6</td>
<td>0D20006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>264</td>
<td>5</td>
<td>0484205</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>216</td>
<td>X</td>
<td>6</td>
<td>0D80006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>270</td>
<td>5</td>
<td>0484385</td>
</tr>
<tr>
<td>SIMON.CBC</td>
<td>222</td>
<td>X</td>
<td>6</td>
<td>0DE0006</td>
</tr>
<tr>
<td>SIMON.KeyAndGo</td>
<td>72</td>
<td>276</td>
<td>5</td>
<td>0484505</td>
</tr>
</tbody>
</table>

\[
z'_2 \leftarrow_R f(x_i, y_1)
\]

\[
u_1 := z'_2 \oplus t_2
\]

\[
x_i \colon= \text{ENTRPY.Stat.96x4}
\]

\[
u_1 = G'(t_3, cu_1)
\]
| SIMON.PRF2     | 150 | X  | 8   | 0960008 |
| SIMON.KeyAndGo | 258 | X  | 5   | 1020005 |
| SIMON.PRF2     | 156 | X  | 8   | 09C0008 |
| SIMON.KeyAndGo | 258 | X  | 5   | 1020005 |
| SIMON.PRF2     | 306 | X  | 8   | 1320005 |
| SIMON.KeyAndGo | 258 | X  | 5   | 1020005 |
| SIMON.PRF2     | 312 | X  | 8   | 1380005 |
| SIMON.KeyAndGo | 258 | X  | 5   | 13E0005 |
| SIMON.PRF2     | 318 | X  | 8   | 13E0005 |
| SIMON.KeyAndGo | 258 | X  | 5   | 13E0005 |
| SIMON.PRF2     | 324 | X  | 8   | 1440005 |
| SIMON.KeyAndGo | 258 | X  | 5   | 1440005 |
| SIMON.PRF2     | 330 | X  | 8   | 14A0005 |
| SIMON.KeyAndGo | 258 | X  | 5   | 14A0005 |
| SIMON.CBC      | 336 | X  | 6   | 1500006 |
| SIMON.KeyAndGo | 258 | 342| 5   | 1025585 |

\[
\begin{array}{c}
\text{SRAM.Send.96} \\
\text{SRAM.Send} \\
\text{SRAM.Send} \\
\text{SRAM.Send} \\
\text{SRAM.Send} \\
\text{SRAM.Send} \\
\text{SRAM.Send} \\
\end{array}
\]

\[
\begin{array}{c}
126  \quad \times  \quad 07E0017 \\
132  \quad \times  \quad 0840017 \\
138  \quad \times  \quad 08A0017 \\
144  \quad \times  \quad 0900017 \\
150  \quad \times  \quad 0960017 \\
156  \quad \times  \quad 09C0017 \\
54   \quad \times  \quad 0360017 \\
228  \quad \times  \quad 0E40017
\end{array}
\]
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Line</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM.Send</td>
<td>306</td>
<td>X</td>
<td>1320017</td>
</tr>
<tr>
<td>SRAM.Send</td>
<td>312</td>
<td>X</td>
<td>1380017</td>
</tr>
<tr>
<td>SRAM.Send</td>
<td>318</td>
<td>X</td>
<td>13E0017</td>
</tr>
<tr>
<td>SRAM.Send</td>
<td>324</td>
<td>X</td>
<td>1440017</td>
</tr>
<tr>
<td>SRAM.Send</td>
<td>342</td>
<td>X</td>
<td>1560017</td>
</tr>
<tr>
<td>t'₄ ← SRAM.Rec</td>
<td></td>
<td>X 348</td>
<td>0005718</td>
</tr>
<tr>
<td>t₄ = t'₄</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHKJMP.L1</td>
<td>264</td>
<td>X 14</td>
<td>108000E</td>
</tr>
<tr>
<td>CHKJMP.L2</td>
<td>348</td>
<td>0 15</td>
<td>15C000F</td>
</tr>
<tr>
<td>(y₁, sk) := (y₂, t₅)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEMCPY</td>
<td>270</td>
<td>0 21</td>
<td>10E0015</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>271</td>
<td>1 21</td>
<td>10F0055</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>272</td>
<td>2 21</td>
<td>1100095</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>273</td>
<td>3 21</td>
<td>11100D5</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>274</td>
<td>4 21</td>
<td>1120115</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>275</td>
<td>5 21</td>
<td>1130155</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>276</td>
<td>6 21</td>
<td>1140195</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>277</td>
<td>7 21</td>
<td>11501D5</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>278</td>
<td>8 21</td>
<td>1160215</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>279</td>
<td>9 21</td>
<td>1170255</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>280</td>
<td>10 21</td>
<td>1180295</td>
</tr>
<tr>
<td>MEMCPY</td>
<td>281</td>
<td>11 21</td>
<td>11902D5</td>
</tr>
<tr>
<td>JMP</td>
<td></td>
<td>X 0 16</td>
<td>0000010</td>
</tr>
</tbody>
</table>

Table A.2: AGMSY’15 mapping on ASIP
**Setup Phase**

<table>
<thead>
<tr>
<th>Server</th>
<th>$(sk, y_1) \xleftarrow{R} \text{TRNG}$</th>
<th>Device $(f(x_i, \cdot))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(sk, y_1)$</td>
<td>$\xrightarrow{z_1}$</td>
<td>$z_1 \xleftarrow{R} f(x_i, y_1)$</td>
</tr>
</tbody>
</table>

**Authentication Phase**

<table>
<thead>
<tr>
<th>Server ${ (z_1, sk, z_{old}, sk_{old}) }_i$</th>
<th>Device $(f(x_i, \cdot), sk, y_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1$</td>
<td>$(z_1', sk, z_{old}, sk_{old})$</td>
</tr>
<tr>
<td>$\delta, \text{rand}, y_2' \leftarrow {0, 1}$</td>
<td>$r_1 := G(sk, z_1' \text{rand})$</td>
</tr>
<tr>
<td>$hd_1 := \text{FE.Gen}(z_1', \delta)$</td>
<td>$c := \text{SKE.ENC}(sk, hd_1)$</td>
</tr>
<tr>
<td>$(t_1, \ldots, t_5) := G(r_1, y_1' \parallel y_2')$</td>
<td></td>
</tr>
<tr>
<td>$z_2' \xleftarrow{R} f(x_i, y_1)$</td>
<td>$z_2' \leftarrow t_2'$</td>
</tr>
<tr>
<td>$u_1 := z_2' + t_2$</td>
<td>$v_1 = G(t_3, c \parallel u_1)$</td>
</tr>
<tr>
<td>$c, y_2', t_1, u_1, v_1$</td>
<td>$r_1 := \text{FE.Rec}(z_{old}, hd_1)$</td>
</tr>
</tbody>
</table>

$$\text{hd} := \text{SKE.Dec}(sk, c)$$

$$r_1 := \text{FE.Rec}(z_1', hd)$$

$$(t_1', \ldots, t_5') := G(r_1, y_1' \parallel y_2')$$

If $t_i' = t_i$ in $1 \leq i \leq \text{num}$,

- If $v_1 = G'(t_3', c \parallel u_1)$,
  - $z_2' := u_1 + t_2'$
  - Update $(z_1, sk, z_{old}, sk_{old})$ to $(z_2', t_5, z_1, sk)$
- Else, $hd_1 := \text{SKE.Dec}(sk_{old}, c)$
  - $r_1 := \text{FE.Rec}(z_{old}, hd_1)$
  - $$(t_1', \ldots, t_5') := G(r_1, y_1' \parallel y_2')$$

Else, $t_4' \xleftarrow{R} \text{TRNG}$

$$(y_1, sk) := (y_2, t_5)$$

**Figure A.2:** The AGMSY’15 protocol [5]
Bibliography


[26] Top500, “Top500 supercomputer sites.”


[104] S. Underwood, “Mspgcc port of the GNU tools to the texas instruments MSP430 microcontrollers,” *Online in Internet: URL:*


[154] T. Pöppelmann, L. Ducas, and T. Güneysu, “Enhanced lattice-based signatures on reconfigurable hardware,” in Cryptographic Hardware and


