Instruction Set Extensions for Support of Cryptography on Embedded Systems

by

Stefan Tillich

A PhD Thesis
Presented to the Faculty of Computer Science in Partial Fulfillment of the Requirements for the PhD Degree

Assessors

Prof. Dr. Karl Christian Posch (TU Graz, Austria)
Prof. Dr. Bart Preneel (KU Leuven, Belgium)

November 2008

Institute for Applied Information Processing and Communications (IAIK)
Faculty of Informatics
Graz University of Technology, Austria

Abstract

Digital computing devices continue to be increasingly dispersed within our everyday environments. Computers are "embedded" into everyday appliances in order to serve predominantly one of two purposes: Either take over the functionality of analog electronic components or enable new services in their own right. While such digital computing capabilities are arguably a key enabler for exciting new applications, the potential hazards should not be overlooked. Problems which exist in the much more familiar domain of desktop computing (e.g., development of correct software) are now introduced into these new fields. At the same time, embedded computers also face new challenges, e.g., severe restrictions of resources like computing power, memory, and energy.

One of the more pressing problems of embedded computing is the provision of adequate security mechanisms. While there are some robust solutions available for the desktop domain, resource restrictions often prevent their direct application for embedded devices. The basic problem is constituted by the fact that modern cryptographic algorithms still present a significant overhead for such constrained systems.

As most embedded processors will be charged with the execution of cryptographic algorithms, it is worthwhile to revisit these processors' capabilities in this regard and to consider the benefits of "tweaking" their functionality towards these specific workloads. The main vehicle for such a tweaking is the addition of custom instructions into the default instruction set architecture of the processor. Such *instruction set extensions* have been highly successful in areas like multimedia and digital signal processing. In this thesis we examine instruction set extensions for cryptography, with a special focus on secret-key algorithms.

Three main goals are pursued within this thesis. The first goal is the investigation of potential new instructions (design space exploration) and the proposal of worthwhile candidates. The second goal is concerned with the efficient implementation of the proposed instructions and the evaluation of their effectiveness in a realistic setup. This activity has lead to the creation of the LEON2-CIS embedded processor, which is a variant of the SPARC V8-compatible LEON2 processor and which incorporates all of the instructions which we propose in this thesis. The LEON2-CIS is available under the GNU LGPL in order to document our efforts and to provide a basis for further research. The third goal of this thesis is concerned with strategies for securing embedded processors against the threat of implementation attacks (most importantly side channel attacks).

This thesis collects our research work from the last years, most of which has already been disseminated through academic publication. The publications have

been put into a coherent form and have been complemented with new material. In addition to documenting our work, we have strived to provide references to relevant publications by research groups dealing with related topics.

Acknowledgements

First of all I would like to thank Johann Großschädl for the very fruitful cooperation during the last years, which has manifested itself in the form of the LEON2-CIS processor. Johann's work has provided both an excellent starting point as well as a valuable complement to my own. Furthermore, I consider myself very lucky that I have been able to learn from Johann a great deal about the academic world (and especially the publishing process).

Many thanks also go to my fellow colleagues and friends who have supported me during my studies and work. I would especially like to mention my coauthors (apart from Johann) of the publications which constitute the "flesh and bones" of this thesis: Martin Feldhofer, Christoph Herbst, Stefan Mangard, Thomas Popp, and Alexander Szekely. In this regard, I would also like to thank David Canright, Dan Page, and Johannes Wolkerstorfer for their support.

Stefan Mangard and Elisabeth Oswald deserve special credit for providing guidance and support and for their patience with their younger colleagues. Furthermore, I'd like to thank the academic staff at IAIK, but especially Manfred Aigner for granting me enough leeway to finish this work and Karl Christian Posch for serving both as my advisor and assessor for this thesis. Many thanks also go to Bart Preneel for serving as second assessor and for his willingness to make the trip to Graz.

Thanks also to all who took the time to read over the drafts of this thesis and made valuable comments and suggestions: Manfred Aigner, Christoph Herbst, Mario Kirschbaum, Marcel Medwed, Thomas Popp, Thomas Plos, Jörn-Marc Schmidt, and Alexander Szekely.

Last but not least I would like to thank my family and friends, which have constantly supported me during the whole course of my studies, but especially my parents, without whom none of this work would have been possible.

Stefan Tillich
Graz, November 2008

Contents

List of Tables

List of Figures

Acronyms

ADSP	Application Domain-Specific Processor
AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Set Processor
ASM	Assembly
AVR	Advanced Virtual RISC
CM	Countermeasure
CIS	Cryptography Instruction Set
CMOS	Complementary Metal-Oxide Semiconductor
COP	Coprocessor
CPA	Carry-Propagate Adder
CPI	Coprocessor Interface
CSA	Carry-Save Adder
CTR	Counter
DES	Data Encryption Standard
DFA	Dual-Field Adder
DMRP	Data Memory Read Port
DPA	Differential Power Analysis
DPW	Datapath Width
DSA	Digital Signature Algorithm
DSP	Digital Signal Processor
DSRCP	Domain Specific Reconfigurable Cryptographic Processor
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
EDA	Electronic Design Automation
EEPROM	Electrically Erasable Programmable ROM
FIPS	Federal Information Processing Standard
FPGA	Field-Programmable Gate Array
FU	Functional Unit
GE	Gate Equivalent
GPP	General-Purpose Processor
HD	Hamming Distance
HDL	Hardware Description Language
HW	Hardware *or* Hamming Weight

ISA	Instruction Set Architecture
ISE	Instruction Set Extension
ISEC	Instruction Set Extensions for Cryptography
ISS	Instruction-Set Simulator
IU	Integer Unit
IW	Issue Width
LSB	Least Significant Bit
LUT	Look-Up Table
MAC	Message Authentication Code *or* Multiply Accumulate
MIPS	Microprocessor without Interlocked Pipeline Stages
MM	Memory Mapped
MSB	Most Significant Bit
NIST	National Institute of Standards and Technology
PKC	Public-Key Cryptography
PPG	Partial Product Generator
RAM	Random-Access Memory
RCA	Ripple-Carry Adder
RFID	Radio-Frequency Identification
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RSA	Rivest-Shamir-Adleman
SCA	Side-Channel Attack
SHA	Secure Hash Algorithm
SKC	Secret-Key Cryptography
SW	Software
TEA	Tiny Encryption Algorithm
VHDL	Very High Speed Integrated Circuit HDL
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
WCET	Worst-Case Execution Time
WDDL	Wave Dynamic Differential Logic
WSN	Wireless Sensor Network

Notation

Throughout this thesis, we make a distinction between operations (functions) and instructions. Operations are more general, e.g., the multiplication of two binary polynomials, and they are written in capital letters, e.g., GF2MUL. Instructions realize operations (or parts thereof) for a specific processor architecture. Instruction mnemonics are written in lower-case non-proportional font, e.g., `sbox4s`. Instruction operands have the same format, e.g., `rs1`.

Whenever new instructions are defined for a specific processor architecture, we follow the respective notation conventions for that architecture. For the SPARC V8 architecture, typical instructions have the following form:

```
sbox4s rs1, rs2, rd
```

The two source registers `rs1` and `rs2` are specified first, followed by the destination register `rd`. Instructions for the AVR architecture are written in this form:

```
aessbox Rd, Rr
```

Here, the destination register `Rd` is specified first, followed by the first source register `Rr` (Note that the destination register also acts as second source register.).

The finite field notations GF(q) and $\mathbb{F}_q$ are used interchangeably. Constants in the binary extension field GF(2^8) are written as two uppercase hexadecimal digits, e.g., 0E, where the most significant bit of the byte value corresponds to the coefficients of the highest power in the polynomial representation, i.e., x^7.

1 Introduction

This chapter gives a motivation of the research work of this thesis by describing the imminent security challenges faced by embedded computing devices in Sections 1.1 and 1.2. Cryptography is identified as a key enabler for security in Section 1.3. The challenges of implementing cryptography on constrained devices are outlined in Section 1.4. The approach of *instruction set extensions* is introduced as an attractive solution to the implementation problem in Section 1.5. This section also contains a brief outline of some important research work in this field and a summary of our contributions and research methodology. Section 1.6 tries to give a fairly complete overview of the research towards extensions for public-key cryptography. For secret-key cryptography, the overview is distributed amongst the following two sections, where Section 1.7 treats the work of other research groups, while Section 1.8 is concerned specifically with our contributions and results. In Section 1.9, the structure of the thesis is outlined.

1.1 The Future of Computing

About a decade ago, the term *ubiquitous computing* arose as a buzzword to describe the next model of human-computer interaction beyond the desktop paradigm. The future environment is envisioned to be filled with objects capable of information processing and interaction. Heterogeneous devices embedded in everyday objects will contribute to a concerted effort in order to provide specific services for humans. Some services will require explicit human interaction in some form or another, while other services will remain completely hidden from the beneficiaries.

One of the main enablers of this new computing paradigm is undoubtedly the

continuing advancement of the manufacturing techniques of integrated circuits. Postulated in the mid-60s, Moore's law [177]—which predicts a doubling of maximal transistor number per integrated circuit every two years—has up to now largely held up to the test of time. It is not certain how long the current pace of development can be sustained. In 2005, Gordon Moore himself predicted his law to be able to uphold for another 10 to 20 years, before the fundamental barrier (i.e., the size of atoms) would be reached [63]. Industry roadmaps [137] and upcoming technological advances (e.g., multigate devices [204]) support Moore's view. However, whether it will be possible to transfer the momentum over to a new technology remains an open question.

The increase in integration density has been supported by advances in Electronic Design Automation (EDA). Modern EDA tools allow for the creation of increasingly complex designs, e.g., heterogeneous multi-core processors like the Cell processor [122], while modern Integrated Circuit (IC) production technology affords for transistor counts well above one billion per chip [222]. As a consequence, even smaller devices like PDAs and cell phones can now be equipped with complex digital processing capabilities.

Of the billions of processors produced today, only a tiny fraction is used in conventional general-purpose computers like PCs [22]. The vast majority is built into so-called *embedded systems*, which are designed to serve a special purpose. Embedded systems are found in many modern everyday appliances like wristwatches, bike tachometers, toothbrushes, TVs, PDAs, cell phones, etc. Embedded systems are also at the heart of critical equipment like medical diagnosis and treatment devices or flight control systems.

One of the challenges for modern embedded system design lies with the implementation of whole digital systems and networks of such systems on a single chip. These design paradigms are commonly denoted as *System on Chip* and *Network on Chip*. The development and adoption of new communication standards for low-cost devices continues to allow for improved networking capabilities of embedded systems.

Exciting new application domains have arisen as a consequence of the aforementioned technological advances. One example is the domain of *Radio-Frequency Identification (RFID)*. The principal idea is to facilitate the identification of objects by attaching simple electronic devices which can deliver an ID over a wireless connection. Originally intended as a replacement for the barcode, the growing capabilities of modern RFID tags have enabled many new applications beyond simple identification.

Another example is the field of *Wireless Sensor Networks (WSNs)* which has attracted a great deal of scientific and commercial interest. The basic components are low-cost devices (often denoted sensor nodes or motes) which can interact with their environment, gather data over sensors and manipulate the surroundings over actuators, process and store data, and communicate with nearby devices. The general description of WSNs already captures many of the features required for ubiquitous computing. Possible applications range from large-scale monitoring (e.g., battlefield surveillance, detection of forest fires) and

small-scale monitoring (e.g., home intrusion detection) over tracking (e.g., goods in a warehouse) to controlling (e.g., in power plants, chemical plants).

1.2 Challenges for Ubiquitous Computing

In ubiquitous computing applications, many participating devices will have to operate for prolonged periods without the possibility of manual replenishment of their energy source. This may be due to physical unattainability (e.g., stress sensor devices cast in the concrete of buildings), undiscoverability (e.g., sensors attached to free-roaming animals), or simply due to economic reasons (e.g., too large number of devices). Similar problems are already faced by many WSN applications and have led to research into different solutions. One avenue of approaches deals with enhancements on the power supply side, such as the increase of battery capacity or the gathering of energy from the environment (*energy harvesting*). On the other side, reduction of power consumption can be effected on several levels with appropriate semiconductor technology, hardware architecture, software architecture, or overall system architecture.

Another evident challenge of ubiquitous computing is the security of the systems. Many of the devices will have wireless network interfaces for communication. Due to their broadcast nature, wireless networks are inherently susceptible to eavesdropping. Furthermore, as access to the medium is virtually unrestricted, attacks like jamming or message injection can be easily performed. Furthermore, many devices will be deployed in public areas, allowing for physical adjacency or even physical contact for attackers. This is not only relevant for general problems like vandalism or theft. The protection of sensitive data stored in the devices is of critical importance in many applications and must be ensured in the face of new attack methods (e.g., side-channel attacks, cf. Chapter 11). For example, a ubiquitous computing environment where many of the deployed devices have come under the control of a malicious entity can lead to a multitude of threats to its users, ranging from compromise of privacy to physical harm.

1.3 The Role of Cryptography

The foundation for security is laid by the application of sound cryptographic primitives (algorithms) and protocols. An important point to note is that cryptographic methods cannot generate *trust* in a system. What they can do is to shift the assumptions made by the system's user (the truster) into the system (the trustee) from unreasonable ones (e.g., no unauthorized entity will eavesdrop on the wireless medium) to reasonable ones (e.g., a specific cryptographic key is only available to authorized communication participants). More elaborated security schemes will not even rely on the full validity of these "reasonable" security assumptions. They will try to detect possible breaches and take appropriate actions to minimize the resulting damage to the overall system.

This thesis deals with the cryptographic foundation which needs to be available in the ubiquitous computing devices as an enabler of security and trust. The basic problem that is investigated is that current embedded microprocessors, which are at the heart of virtually any embedded device, are not designed to handle cryptographic workloads efficiently. The reason for this is that processors were originally conceived to perform arithmetic operations natively only on integer and real numbers. Cryptography on the other hand usually employs arithmetic on diverse finite fields which cannot be performed directly with native operations of the processor. This may be due to the size of the operands, which may well exceed the word size of the processor, and/or to the specific characteristics of the required finite field operations.

Additionally, many cryptographic primitives include functions like permutations or bit operations which do not map very well to the operations offered by processors. Fortunately, most modern cryptographers are aware of this problem and try to omit too "cumbersome" operations in the design of new cryptographic primitives. There have been even attempts to design cryptographic primitives solely based on native operations of processors, e.g., the TEA family of block ciphers [187, 256, 257]. But while they definitely have some advantages over other modern block ciphers, e.g., simplicity of implementation and small code size, empirical data indicates that performance of the TEA ciphers is not significantly better than that of several other modern block ciphers (e.g., AES) [104, 203]. The reason for this is that although ciphers like TEA use only simple operations, it is in turn necessary to perform a great number of such operations in order to gain reasonable cryptographic security (e.g., TEA has a suggested number of 64 rounds).

1.4 Implementing Cryptography on Constrained Devices

In addition to meeting performance demands, cryptography in embedded systems must also be efficient in terms of resources (e.g., working memory, code size) and energy efficient (mostly to prolong battery life). Depending on the emphasis of the application, these constraints need to be taken into account when implementing cryptographic methods. Whenever the constraints cannot be met by software optimization, the system designer has to enhance the system's hardware to increase performance and efficiency. The traditional approach is to add a *cryptographic coprocessor* to the general-purpose processor of the system and to offload the bulk of the cryptographic operations to this coprocessor.

The use of cryptographic coprocessors has several advantages. The most obvious one is that computations on the coprocessor are typically much faster than on the host processor and normally result in increased energy efficiency. Also, cryptographic implementations using a coprocessor typically require significantly less resources. Simultaneous operation of processor and coprocessor is usually possible, leading to additional speedup if the application at hand allows

concurrent execution. Another important point is that coprocessor design and integration are relatively mature and well-understood techniques. The border between host processor and coprocessor is typically clear cut, which facilitates the job of the hardware developer.

On the other hand, the coprocessor approach is not devoid of shortcomings. Cryptographic coprocessors have a fixed functionality, handling specific algorithms in specific modes of operation. Often the choice of key length and other cryptographic parameters is limited or all parameters are predetermined. Due to the clear separation of the coprocessor, facilities of the host processor (e.g., registers files, decode logic) cannot be used and need to be replicated, requiring more silicon area. Furthermore, the use of a coprocessor can result in a considerable communication overhead, which may significantly impair the overall benefit. Also, in a multi-tasking environment, different processes might compete for the coprocessor, which requires some form of arbitration.

Flexibility is normally a much desired feature of a cryptographic device in order to react to advances in cryptanalysis. On the one hand, it is possible to update the cryptographic implementation to replace or strengthen algorithms and protocols found to be too vulnerable to traditional cryptanalytic attacks. On the other hand, side-channel cryptanalysis targets the implementation itself and it might be necessary to continually adapt to new and more powerful attacks.

In this thesis, we have investigated a new strategy to fill the existing gap between state-of-the-art cryptography and embedded processing devices. The basic idea is to integrate the cryptographic support in the general-purpose processor rather than in a separate coprocessor. The new operations are available as custom instructions, thus extending the original instruction set of the processor. This approach of *instruction set extensions* has already been highly successful in other fields of application. Instruction set extensions have been designed for efficient handling of multimedia content. Prominent examples include Hewlett-Packard's Multimedia Acceleration eXtension (MAX) [155], Intel's MMX [195] and Streaming SIMD Extensions (SSE) [231], AMD's 3DNow! [1], Sun's Visual Instruction Set (VIS) [229], and Freescale's AltiVec [79]. In the domain of digital signal processing, appropriate instruction set extensions have been so successful that they have spawned a whole new branch of processors, namely the *digital signal processors (DSPs)*.

1.5 Instruction Set Extensions for Cryptography

The widespread need for security in future embedded applications has called for the adaptation of the instruction set extension approach to the domain of cryptography. The ambitious goal of this thesis is to take the "best of both worlds" from software and hardware in order to make strong cryptography available on low-cost embedded processors in a flexible and scalable way, with good performance and economical use of resources. Our work spans from the analysis of

cryptographic algorithms and design space exploration for possible extensions, over design and integration of instruction set extensions and supporting tools, to performance analysis and investigation of implementation security. Most of the practical aspects of our research have been performed in relation to a state-of-the-art embedded processor design, thus allowing for a coherent and thorough analysis.

Cryptographic primitives can be separated into the three major categories of asymmetric (public-key) algorithms, symmetric (secret-key) algorithms, and unkeyed algorithms. Symmetric and unkeyed primitives bear many similarities and are sometimes regarded as a single class. Symmetric cryptography deals with methods of securing communication, i.e., protecting the confidentiality, integrity and authenticity of messages, provided that a shared key is known by all authorized participants (and only by them). In asymmetric methods, each participant has a unique key pair, where one key is distributed publicly (the public key) and the other key is kept secret (the private key). As private keys are in the sole possession of a single entity, cryptographic operations with a private key can be distinctly attributed to an entity using the corresponding public key. Therefore, asymmetric primitives can be used to provide the service of non-repudiation, i.e., the undeniability of performed actions. This is not possible with symmetric primitives as the key is shared among several participants and actions performed with it cannot be attributed to a specific entity.

From a functional standpoint, asymmetric primitives can provide a superset of the services of symmetric primitives. However, practical implementations of asymmetric primitives are several orders of magnitude slower than their symmetric counterparts. Therefore, the primary application of asymmetric primitives lies in the additional services they can deliver. One of the most important applications of asymmetric primitives is to use them for establishing shared secrets between communicating parties. This makes for a perfect addition to the methods of symmetric cryptography.

Cryptographic protocols and applications normally use both asymmetric and symmetric primitives to provide security services. Asymmetric primitives are slow and require considerable resources. They are used rather infrequently in most scenarios (e.g., key refreshing in a regular interval). Symmetric primitives are more light-weight to implement and much faster, hence they are used much more frequently (e.g., for encryption of messages). When it comes to implementation in embedded systems, asymmetric cryptography tends to pose problems of feasibility and latency, while symmetric cryptography is more connected to challenges of adequate performance and minimizing required resources. Consequently, all solutions which try to alleviate the problems of implementing cryptography in embedded systems have to take these basic goals into account.

Our work has been accompanied by an interesting paradigm shift in industry towards configurable embedded processors. Companies have started to offer complete solutions for developing processors which are customized towards the application(s) at hand. Some of the most important examples are *Tensilica Xtensa* [230], *MIPS CorExtend* [175], *ARC ARChitect* [10], *STMicroelectron-*

ics ST200 [58], and *CoWare Processor Designer* [54]. The starting point is a base processor which can be enhanced by the embedded system developer in a more or less automated way. Custom instructions can be added either manually or automatically through analysis of application code and selection of worthwhile candidates. Additionally, other characteristics of the processor can also be configured, e.g., cache size and organization. The configured processor is then delivered by the tools in a standard hardware description language (e.g., VHDL or Verilog) which can be used for hardware implementation. Usually, a customized toolchain which incorporates the new instructions is also generated and can be used for software development. In addition to the general advantages of the instruction set extension approach, the companies claim that the reliance on the (hopefully sound) design of the base processor and the tool support for processor configuration and toolchain generation minimizes the chance for design errors and that the development time can be reduced. Currently, some companies offer different base processor designs which are already geared towards specific application domains (e.g., multimedia processing or digital signal processing).

Important providers of microprocessors have included support for security into their products, e.g., *ARM SecurCore* [11], *MIPS SmartMIPS* [174], *STMicroelectronics SmartJ* [227], *Atmel XMEGA* [17], *VIA PadLock* [250], and *Sun UltraSPARC T2* [228]. The IA-64 architecture by Intel and Hewlett-Packard, as implemented in the Itanium processor, has also been optimized to address cryptographic needs [131]. Intel has also announced instruction set support for AES and binary polynomial multiplication in future generations of their processors (with the Westmere code name) [129, 130] Unfortunately, as companies try to retain competitive advantages, many implementation details of these cryptographic extensions are unavailable to the public.

1.5.1 Previous Work

More than a decade ago, Nahum et al. were the first to suggest "to do classic RISC processor (or coprocessor) design on a large set of cryptographic software implementations" [181]. This was amongst three strategies to bring cryptographic performance up to Gigabit/s network speeds: Design of new (more efficient) cryptographic algorithms, processor parallelism, and processor enhancements. While throughput of secret-key algorithms on current desktop processors has now reached the Gigabit/s range [160, 167], the introduction of 10 Gigabit/s Ethernet [128] and work commencing on 100 Gigabit/s Ethernet [123] has continued to leave software performance trailing behind. In his PhD thesis, Jean-François Dhem was the first to propose concrete enhancements to a processor architecture (ARM7M) in order to support long integer modular multiplication as a benefit for public-key algorithms [59]. Dhem concluded that his thesis "could be continued by a similar work on emerging cryptographic algorithms like the ones on elliptic curves".

First publications proposing concrete instruction set extensions for symmetric-key primitives started to appear around 2000 [38, 84, 221]. Johann

Großschädl (in cooperation with several other researchers) dealt with many aspects of instruction set extensions for public-key cryptography. This includes the design of custom instructions and proposals for integration into standard processor architectures [90, 91, 95, 96, 98, 99, 101, 249], development and implementation of according hardware [89, 94, 97], and performance evaluation from a protocol level [150].

1.5.2 Our Contribution

The work described in this thesis complements the efforts of Johann Großschädl towards public-key extensions in many regards. Most importantly, the focus has been set on instruction set extensions for secret-key algorithms. Apart from the design and implementation of such extensions [237, 240], synergies between public-key and secret-key support have been investigated [236, 239]. Another very important point has been research towards the role of instruction set extensions in secure implementation of cryptographic algorithms, i.e., resistance against implementation attacks (most notably side-channel attacks) [238, 241, 243].

Further research work has been concerned with low-cost extensions for Elliptic Curve Cryptography [234], evaluation of performance and energy efficiency of cryptographic primitives on embedded devices [92, 103, 104, 235], design space and algorithm exploration for public-key extensions [93, 100, 102, 151], optimization of implementations of secret-key extensions [232], evaluation of proposed public-key extensions [105], and practical investigation of advanced side-channel attacks [191].

One important feature of our research has been the continuous alternation between theoretical investigation and practical implementation and verification. Most of our proposed extensions have been implemented and tested in real hardware in order to gain a deeper insight into general practicability and opportunities for further improvements. As platform we have chosen the LEON2 embedded processor, which is compliant to the SPARC V8 architecture. The Scalable Processor Architecture, Version 8 (SPARC V8) [225] is a 32-bit RISC instruction set architecture (ISA) which possesses many typical features that are common to all major ISAs for embedded processors. Another big advantage of the LEON2 is that its HDL source code is distributed under the liberal terms of the GNU Lesser General Public License (LGPL), which allows for convenient use in research projects. Practical results obtained on the LEON2 processor allow to draw conclusions which apply to most 32-bit embedded RISC processors in use today. In our practical work we have implemented the proposed custom instructions and architectural enhancements in the LEON2 processor. This applies to the secret-key extensions presented in this thesis as well as to the public-key extensions proposed by Johann Großschädl. The resulting enhanced processor has been labeled *LEON2 with Cryptography Instruction Set (LEON2-CIS)* and has been made publicly available to document and disseminate our results as well as to facilitate and foster additional research into the same direction [134].

The research work underlying this thesis has been done mainly in the years

2004 to 2008. As a very rough general outline, the work can be divided into five main activities:

Study of previous work Research into instruction set extensions for secret-key cryptography can benefit from the results of several related research directions. Most notable is work on Application-Specific Instruction Set Processors (ASIPs) and Application Domain-Specific Processors (ADSPs) which can contribute new approaches for architectural enhancements of general-purpose processors. Some concepts from the design and implementation of dedicated hardware like cryptographic processors or coprocessors can also be applied to instruction set extensions. Furthermore, we have drawn benefit from work on automatic design space exploration and side-channel attacks.

Design space exploration This activity dealt with the identification of performance bottlenecks and suitable areas for support with instruction set extensions. Methods of both manual and automatic design space exploration have been applied.

Extensions design Following the findings of design space exploration, we tried to identify specific custom instructions and architectural enhancements "optimal" under specific constraints (e.g., low memory usage, low cost, high performance). A general goal was to achieve a high degree of flexibility for cryptographic software in regard to the extensions.

Implementation and evaluation As already outlined above, the proposed extensions have been implemented on the LEON2 processor. The functionality of the custom instructions has been mapped to extended or new functional units (FUs). In order to make use of these instructions, the GNU assembler (gas) has been enhanced with new opcodes. The processor has been synthesized with standard-cell technologies in order to get an accurate estimate of the resulting hardware overhead. On the other hand, optimized cryptographic software has been benchmarked on real hardware on an FPGA prototype implementation of the enhanced LEON2.

Increasing implementation security Another body of work was concerned with strategies to enhance the implementation security of cryptographic algorithms in the face of instruction set extensions. We have investigated to what extent our proposed cryptographic enhancements can contribute to secure software implementations. Furthermore, we have tried to increase the implementation security through hardware modifications of the processor.

The next sections try to give a more detailed overview of the most important research work conducted in the field of instruction set extensions for cryptography. Section 1.6 treats support for public-key cryptography. The subsequent two sections describe the work for secret-key cryptography support: Section 1.7 features the most important results from other research groups, while Section 1.8 summarizes our contributions to this research field.

1.6 International Research Work Regarding Support for Public-Key Cryptography

1.6.1 Integer Modulo Arithmetic

Koç has examined the suitability of the Intel MMX instructions for implementing public-key cryptography [42]. He concluded that these multi-media extensions bore no benefit for this kind of workload, mainly due to the sole availability of signed integer operations. Dhem proposed ARM7M support for a $(32 \times 32 + 32 + 32)$-bit multiply-accumulate operation for long integer modular multiplication [59]. He also described a $(8 \times 32 + 8 + 32)$-bit multiply-accumulate (MAC) unit which could be used to execute the required operation in four clock cycles. Großschädl examined support for the same operation for the MIPS32 architecture [90]. Cycle-accurate simulation of a SystemC model of a MIPS32 core yielded a speedup factor of 2 for multiple-precision integer modular multiplication using the Montgomery method [176]. The algorithm considers the *coarsely integrated operand scanning (CIOS)* method [44]. A concrete implementation of a multiply-accumulate unit has been given by Großschädl et al. in [94]. Further refinements of the instructions and use of *finely integrated operand scanning (FIOS)* and *finely integrated product scanning (FIPS)* methods have been given in [98] and [95], respectively.

In a joint work with Johann Großschädl, we have investigated the energy characteristics of different algorithms for long integer modular arithmetic [92]. Using power simulation with the tool *JouleTrack* [223], we were able to show that the Karatsuba and Comba multiplication with Montgomery reduction (KCM method) required the least energy of all surveyed algorithms. On the other hand, in a follow-up work we demonstrated that the FIPS methods outperforms the KCM method in terms of pure performance for typical operand sizes used in state-of-the-art cryptography [102].

1.6.2 Non-Integer Arithmetic

Another line of work deals with support of operations in arithmetic structures other than integers. The main focus has been set on finite binary extension fields, commonly denoted as $\mathrm{GF}(2^m)$ or $\mathbb{F}_{2^m}$. Nahum et al. already suggested to add support for arithmetic in $\mathrm{GF}(2^{155})$ to RISC processors, without giving a concrete implementation [181]. Koç et al. proposed bit-level and word-level algorithms for Montgomery multiplication over binary extension fields [43]. They found that a basic operation for word-level algorithms was the multiplication of two binary polynomials. Furthermore, they suggested to add support for multiplication of two word-size binary polynomials to processors in order to speed up the proposed algorithms. The name given to this operation was MULGF2. Previously, Drescher et al. had proposed to integrate support for $\mathrm{GF}(2^m)$ multiplication into a conventional integer multiplier [62]. Their architecture is able to perform (17×17)-bit integer multiplication and finite field multiplication on fields of degree 8 or less. It has been conceived for supporting error correction

codes on digital signal processors. However, public-key cryptography requires finite field degrees larger by more than an order of magnitude. The architecture proposed by Drescher et al. does not scale well with increasing degree and is therefore not directly applicable to public-key algorithms.

Bucci presented ideas for a *dual-mode* modular multiplier, which could handle integers and binary polynomials, having cryptography as primary field of application [35]. Goodman et al. presented a reconfigurable cryptographic processor which integrated polynomials operations into an integer modular multiplier [85, 86]. The processor was denoted *Domain Specific Reconfigurable Cryptographic Processor (DSRCP)*. It includes a bit-serial multiplier for operands of up to 1,024 bits, which can be adapted in 32-bit steps to the actual operand size. The design of the DSRCP is described in detail in James Goodman's PhD thesis [87]. Savaş et al. gave *word-level* algorithms for Montgomery multiplication in GF(p) and GF(2^m) along with a multiplier suited for both types of fields [210]. The authors showed that such a *unified* multiplier only required the availability of adder cells (so-called *dual-field* adders) which could suppress the carry for GF(2^m) operations. The resulting hardware implementation can handle operands of arbitrary size. Großschädl [89] and Au et al. [20] presented optimized components for unified multipliers.

Support for the MULGF2 operation has been investigated by Großschädl et al. in the context of a 16-bit RISC processor [96]. The resulting speedup for polynomial multiplication has been shown to range between a factor of 6 to 10. A low-power multiply-accumulator design for 32-bit processors has also been presented by the same authors [97]. This unit can perform $(32 \times 32 + 64)$-bit multiply-accumulate operations for signed and unsigned integers as well as binary polynomials in a single clock cycle at a size of about 10,000 gate equivalents (GEs). An extensive treatment of extensions for arithmetic in GF(p) and GF(2^m) was given by Großschädl et al. in [101]. A total of five custom instructions was proposed to support word-level algorithms over both finite fields.

Fiskiran and Lee have investigated instruction set extensions for binary extension fields in relation to varying word size and superscalar processing [71]. They were able to demonstrate a potential speedup of 22.4 for ECC point multiplication. Furthermore, the authors showed that—when it comes to accelerating public-key algorithms with architectural enhancements—a doubled word size is to be preferred over a doubling of the execution parallelism. Eberle et al. have targeted an 8-bit microcontroller for integrating support for ECC over binary extensions fields GF(2^m) [65]. Their implementation of an ECC point multiplication over GF(2^{163}) on an enhanced ATmega128 required about 2.3 million clock cycles and was considerably faster than implementations of 160-bit ECC over GF(p) and 1,024-bit RSA, which offer approximately equivalent security.

Support for so-called *Optimal Extension Fields (OEFs)* was investigated in [99]. An OEF is a field of the form GF(p^m), where p is a pseudo-Mersenne prime with a structure $p = 2^n - c$, which fits into a single register of the target processor platform. Two custom instructions for the MIPS32 architecture were demonstrated to lead to a speedup of about 1.8 for ECC scalar multiplication

over such a field.

Our cooperative work with Johann Großschädl also touched on the field of public-key cryptography support. The main contribution dealt with very low-cost support of GF(2^m) operations, which could be integrated into the integer adder of RISC processors [234]. We demonstrated that ECC point multiplication over GF(2^{191}) accommodating an arbitrarily chosen reduction polynomial can be sped up by a factor of up to 10. For a single fixed reduction polynomial, the speedup can reach nearly 2. Other joint work has dealt with synergies of digital signal processing workloads with cryptographic operations and the comparison of two common RISC architectures (MIPS32 and ARMv5TE) for their suitability to accommodate architectural enhancements [100]. In this regard the MIPS32 architecture showed slight advantages over its competitor.

For pairing-based cryptography, instruction set support for ternary finite fields GF(3^m) has also been investigated [249]. The authors employ a so-called *tri-field* multiplier, in order to handle integer operations as well as operations in binary and ternary extensions fields.

1.7 Work on Support for Secret-Key Cryptography by Other Research Groups

1.7.1 General Permutations

Shi and Lee were the amongst the first to propose instruction set extensions for secret-key cryptography [221]. They suggested to provide architectural support for arbitrary bit-level permutations; an operation perceived to be beneficial for symmetric ciphers in order to increase diffusion[1]. The authors presented two instructions for performing fast arbitrary permutations of word-size values: While `pperm3r` can be used for permutations with repetitions, the `grp` instruction is also useable for permuting operands exceeding the word size. In a follow-up work, Lee et al. proposed a lower-cost variant of the `pperm3r` instruction (denoted `pperm`) [156]. Furthermore, the `cross` instruction was presented, which performs permutations based on the principles of a Benes network. It has also been shown that the same functionality can be achieved with the `omflip` instruction, which exploits the properties of an omega-flip network. The advantage of `omflip` is a much reduced implementation cost. Another strategy for performing permutations was discussed by McGregor and Lee [171]. The basic idea was to combine subword-level permutation (`swperm` instruction) with bit selection and permutation within subwords (`sieve` instruction) in order to achieve arbitrary bit-level permutations.

1.7.2 Broad Algorithm Support

Burke et al. conducted a detailed performance analysis of eight symmetric ciphers (seven block ciphers and one stream cipher) [38]. An important conclusion was

[1] Property to break connection between plaintext and corresponding ciphertext.

that modern block ciphers show a significant degree of parallelism which can be exploited in suitable hardware architectures. Architectural enhancements have also been proposed with the goal of a broad support of current and (possibly) future symmetric ciphers. This includes a 16-bit modular multiplication (with a fixed modulus), several rotate instructions, support for byte substitution using arbitrary tables and a bit-level permutation instruction similar in functionality to those investigated by the group around Ruby Lee. Follow-up work by Wu et al. led to the development of the *CryptoManiac* cryptographic coprocessor [262]. This processor is a *Very Long Instruction Word (VLIW)* machine which is able to execute up to four instructions per cycle. A distinguishing feature is the processor's ability to combine short-latency instructions (e.g., bitwise logical and arithmetic instructions) to be executed in a single cycle. This is supported by CryptoManiac's ability to handle instructions with up to three source operands.

Another cryptographic processor with an VLIW architecture is the *Cryptonite* processor [37]. Similar to CryptoManiac, Cryptonite's design is based on analysis of secret-key ciphers, but also on hash algorithms. Cryptonite includes two 64-bit datapaths supporting operations for cryptographic algorithms (e.g., byte permutation and rotation). Each datapath is complemented by 32 KB of dedicated local memory, which can be accessed with different address and data widths and which can be used to implement arbitrary substitution operations. The work of Ravi et al. has targeted the 32-bit extensible processor Xtensa from Tensilica [202]. The authors investigated the application of automatic generation of instruction set extensions for different cryptographic algorithms.

Fiskiran and Lee's work has shown that extended addressing modes can lead to substantial performance gains for cryptographic algorithms [76]. The same authors discuss support for table lookup to speed up various symmetric ciphers [73, 74, 75]. Their solutions consists of a number of parallel lookup tables with 256 entries of 32-bit words each, which are made available through the specialized `ptlu` instruction. The PAX processor is an interesting architecture which integrates this instruction as part of its cryptographic support [72, 78]. Nevertheless, PAX is not a pure cryptographic processor and also features general-purpose RISC functionality. A very interesting feature is the scalability of the datapath, which allows PAX processors to be configured to word sizes of 32, 64, and 128 bits.

Smyth et al. have investigated custom instructions for speeding up secret-key algorithms and hash functions [224]. They proposed generic cryptographic extensions in the form of bit-oriented and byte-oriented permutation instructions, support for multiplication and inversion in binary extension fields, and dedicated lookup tables. For support of hash functions, a so-called sequence cache with 256 words of 32 bits each is described. This sequence cache allows data access in a specific order. Dedicated instructions for Rijndael are also proposed, including explicit support for ShiftRows and a combination of SubBytes and MixColumns. For DES, instructions for bit permutation and S-box support are included. Furthermore, Smyth et al.'s processor also incorporates coprocessors for Rijndael and DES.

1.7.3 Specific Algorithm Support for the Advanced Encryption Standard

Irwin and Page have investigated extensions for the Advanced Encryption Standard (AES) in the context of the PLX general-purpose RISC processor with a 128-bit datapath [133]. However, the presented solutions do not scale down to architectures with a smaller word size. Nadehara et al. [180] and Bertoni et al. [26] focused on the AES and proposed different extensions for general-purpose processors. Nadehara et al. suggested a single custom instruction to calculate the AES round transformations for a single State byte in a dedicated functional unit. Effectively, this maps the so-called round lookup (T-lookup), which is commonly used for AES software implementations on 32-bit platforms, into hardware. In contrast, Bertoni et al. started from an AES software implementation which uses only S-box lookup tables and a transposed State representation, which has been described in [25]. They proposed both byte-oriented and word-oriented extensions for a 32-bit Intel StrongARM processor. The byte-oriented extensions have similar functionality as those of Nadehara et al., but they offer more flexibility (e.g., additional support for the key expansion). The word-oriented extensions allow for increased performance at a higher implementation cost.

Interesting insights have also been offered by the works of Schaumont et al. [213] and Hodjat et al. [119], who investigated the integration of an AES coprocessor into the 32-bit SPARC V8-compatible LEON2 processor. They showed that communication bottlenecks can easily become the dominant factor in overall system performance.

1.8 Our Contributions and Results for Secret-Key Cryptography Support

Our contributions to this research field consist of an in-depth study of the support for a single secret-key algorithm, namely the AES. The main goal was to maximize performance and to minimize implementation cost while retaining the scalability and flexibility of software implementation. We believe that in the future there will be an increasing need for constrained embedded devices to support cryptography at a low cost. Therefore we envision devices with a strong support for a *core set* of asymmetric and symmetric algorithms, which can be used to implement cryptographic protocols and to realize all required security assurances. Devices sharing support for this core set then have a cheap and effective means for secure communication. If necessary, communication with systems using other algorithms can still be achieved through software implementation. An overview of our most important contributions is given in the subsequent sections.

1.8.1 Synergies Between Secret-Key Algorithms and Public-Key Algorithms

We have investigated synergies between instruction set support for public-key cryptography and secret-key cryptography in two regards. On the one hand, we looked at the use of existing public-key extensions for secret-key algorithms. On the other hand, we tried to implement functional units which could support instructions for both kinds of cryptographic algorithms. In [236] a subset of the public-key enhancements (for ECC over binary extension fields) from Großschädl et al. [101] has been reused to speed up AES implementations. This has been possible because the AES algorithm also employs arithmetic operations over binary extension fields. We have mainly made use of the multiplication of two binary polynomials realized by the `mulgf2` instruction to implement the relatively costly MixColumns transformation of AES. In this fashion it is possible to increase the performance of an optimized software implementation by up to 25%. A limiting factor is the greatly differing field size employed by the different algorithms. While ECC uses operands of hundreds of bits in length, the AES transformations work in GF(2^8) and small extensions thereof. We integrated explicit support for GF(2^8) into an existing unified multiply-accumulate unit in order to increase the attainable speedup for AES [239]. At an additional cost of about 1.3 kGates, the AES execution can be accelerated by up to 43%.

1.8.2 Dedicated Support for AES

The central point of our work has been the architectural support for AES. Primary investigations focused on increasing memory efficiency, as memory is usually a very scarce resource in embedded systems. Furthermore, we demonstrated on our experimental platform that the memory subsystem can quickly become a performance bottleneck [240]. The T-lookup approach with several 1 KB or 4 KB tables, which is usually the fastest software implementation option on desktop processors, might end up several times slower than a memory-preserving *S-box lookup* strategy, when the available memory resources are insufficient. Moreover, several other factors can make T-lookup unattractive, e.g., the increased energy consumption of memory accesses and vulnerability against cache-based timing attacks. Our solution presented in [240] maps the S-box lookup for a single byte into a dedicated functional unit. At the minimal hardware cost of a few hundred gates, the need for memory accesses for the AES computation can be completely removed. The resulting performance increase can be up to 30%. As another advantage, the code size shrinks by up to 43%. In combination with the concepts of [236], the calculation of AES is up to 3.7 times faster than in a "pure-software" implementation.

After the encouraging results for memory efficiency, we turned our efforts towards the improvement of performance. Benchmarking quickly showed that better support for the MixColumns transformation could yield the best results. In a first step, we mapped the MixColumns transformation (and its inverse) into a dedicated hardware unit, which could produce a single-byte result per

clock cycle. In combination with the (also byte-oriented) extension for the S-box lookup from [240], the speedup already goes up to nearly 4.5 [237]. This solution offers good performance at a relatively small extra hardware cost (less than 1,000 gate equivalents). However, there was still room for improvement with extensions which were able to exploit the whole 32-bit datapath of the processor efficiently.

Nadehara et al.'s approach [180] with T-lookup support seemed suboptimal for various reasons. The most important point was that their `aesenc` instruction could only process a single byte of the State at a time. Another issue was the relatively long critical path of their proposed functional unit, which went through a multiplexer stage (for input byte selection), a hardware S-box (for SubBytes) and a GF(2^m) multiplier stage (for MixColumns).

We tried to increase datapath utilization by expanding the functionality of our byte-oriented instruction set extensions. A natural step was to simply quadruplicate the functionality of the according functional units. With one instruction, all four bytes of a 32-bit register can be substituted according to the AES S-box independently and in parallel. Another instruction interprets the contents of a register as an AES State column and performs the MixColumns transformation on it. Although these instructions make better use of the 32-bit datapath, it turns out that there is no effective way to combine their functionality in order to yield a fast AES implementation. The reason for this problem is the neglect of the ShiftRows transformation.

On 32-bit processors, each of the four columns of the 128-bit AES State is normally packed into a single register. The SubBytes and MixColumns have hardware support to transform the State columns effectively, but this does not help in the manipulation of the rows of the State. Therefore, ShiftRows must be done explicitly in software, which is very costly. Alternatively, the storage of the State could be alternated in each round from column-oriented for MixColumns to row-oriented for ShiftRows. But this would be even more inefficient than explicit ShiftRows.

In their work performed at about the same time, Bertoni et al. experienced the same problems with word-oriented AES extensions. Their solution is to add non-standard access to the four registers containing the State, so that one byte from each registers could be read in and assembled into a 32-bit word [26]. However, such a solution has rather unfavorable properties for implementation in a processor's register file.

Our solution to the problem with ShiftRows requires no modifications to the register file and is therefore much easier to implement. The basic idea is to exploit both existing read ports of the register file to access two AES State columns at a time. In this fashion, the ShiftRows transformation can be performed implicitly with SubBytes and MixColumns without extra computational cost. Moreover, this functionality can be hardwired into the according functional units—therefore resulting in virtually no extra hardware cost—while retaining full flexibility for various AES implementation variants.

With our word-oriented extensions, the resulting performance gain reaches an factor of up to 10, i.e., improvement by an order of magnitude. Moreover,

the code becomes much more compact with reduction of over 80%. The implementation cost is about 3 kGates. For example, a complete AES-128 encryption with a precomputed key schedule can be done with less than 200 clock cycles, which is about twice as fast as the approach of Nadehara et al. Also, our extensions provide support for the key expansion, which increases the key agility of implementations. A complete encryption with on-the-fly key expansion can be performed in 255 clock cycles with a code of 65 instructions (i.e., a code size of 260 bytes).

Another advantage of the proposed custom instructions is their eligibility for superscalar processing. Our theoretical evaluation has shown that a 4-way superscalar processor equipped with three instances of each of the proposed functional units, could reach a fourfold increase in performance (i.e., about 50 clock cycles per encryption). This performance is achievable with a single data memory read port. The maximum degree of parallelism would be exploited with a 6-way superscalar processor with four instances of the AES extensions and two data memory read ports. Such a machine could do the complete encryption in 37 clock cycles. However, this 25% increase in performance would bring about a 50% increase of the size of the processor core.

Our practical results compete very well with previous ones from extended general-purpose processors and cryptographic processors, especially when the low demand for resources is taken into account. The performance of our single-scalar extended processor is only surpassed by processors with large dedicated lookup memories, which are very costly in terms of silicon area and relatively hard to implement. The cycle count performance of the aforementioned hypothetical 4-way superscalar processor is amongst the best to be reported for any general-purpose or cryptographic processor.

We have also designed instruction set extensions for AES for the popular AVR 8-bit architecture [242]. Our proposed extensions unify three strategies in order to increase performance: Native support for required operations (i.e., GF(2^m) arithmetic, S-box lookup), execution of several AES transformations in a single instruction, and simultaneous operation on two State bytes. With an additional cost of less than 800 gates the AES performance can be increased by a factor of 3 for encryption and 3.6 for decryption.

1.8.3 Protection Against Side-Channel Attacks

Since the first publications on side-channel attacks (SCAs) by Kocher et al. about a decade ago [147], the topic of implementation security has attracted an ever increasing interest from the cryptographic research community. The principal problem is that a concrete implementation of a cryptographic algorithm normally leaks information about intermediate results via various physical values. Prominent examples of such physical values are execution time [147], power consumption [148], and electromagnetic emanations [201, 81]. Capture and subsequent analysis of these values allows to draw conclusions about the actual intermediate results which occurred during the cryptographic computations and in turn recover the key. One interesting variant of timing analysis in the context

of software implementations is the cache-based timing attack [23, 194, 248].

A great bulk of research work is available, which discusses various forms of SCA attacks and/or appropriate countermeasures for both software and hardware implementations. In the context of software implementation on general-purpose processors, the most important approaches are algorithmic masking (e.g., [88]) and randomized execution (e.g., [170]). Moreover, the efficient application of *secure logic styles* (e.g., [197, 244, 246]) is still an open research problem. In any case, the advent of cryptography instruction set extensions has added a new facet for research into SCA countermeasures.

We have investigated the challenges and opportunities for SCA countermeasures for processors with cryptography extensions. We have shown that ISEs help to protect software implementations against implementation attacks [243]. The use of dedicated instructions for the S-box lookup can both prevent cache-based timing attacks on the AES rounds and power analysis of the AES key expansion [164]. However, protection against advanced power analysis methods demands a dramatic overhead in terms of execution time. In order to increase the protection against power analysis by a factor of about 10^4, the performance degrades by a factor of about 100 (under conservative assumptions). This has shown that software countermeasures alone might not be sufficient to protect devices against SCA. Therefore, we have also considered to equip the implementation of the enhanced processor with hardware protection measures [238]. We presented three approaches for increasing implementation security, which had a relatively moderate impact on AES performance, with an increase in cycle count from about 17% to 206%. Our approaches rely mainly on the application of so-called secure logic styles, which provide protection at the hardware cell level. Depending on the attainable security and implementation details, the hardware overhead amounts to about 28 kGates. Although this is a considerable increase in silicon area, our solution is relatively easy to implement as it leaves most of the original processor's hardware untouched.

In [241] we have performed a practical evaluation of software countermeasures on a typical 8-bit smart card. Although we were able to conduct successful power analysis attacks on a protected AES implementation, our results indicated that the examined countermeasures can be expected to add significant protection. The actual protection factor (i.e., the increase in the number of required power traces) will therefore lie somewhere between the conservative theoretical estimations of [243] and the practical results from [241].

1.9 Organization of This Thesis

In this thesis we have strived to provide an exhaustive description of all our research results relevant to the large topic of cryptography instruction set extensions. Most of these results have already been disseminated through publication in international conferences and journals. Due to space restrictions, these publications could only deal with specific results of our work. We hope that this thesis can complement our dissemination efforts by providing a more complete

and coherent view on this interesting research topic.

This thesis is organized in the following way: In Chapter 2, we give a short overview of the most important concepts of modern cryptography and state-of-the-art algorithms and protocols. Topics of special interest for embedded systems will be highlighted. We then review the implementation options for cryptographic algorithms on embedded systems in Chapter 3. This encompasses both pure-software and pure-hardware implementations as well as hardware/software codesign approaches. We expand on the approach of instruction set extensions for cryptography. The subsequent technical Chapters 4 to 13 largely correspond to individual contributions to conferences and journals. The original publications are cited in parentheses. We describe our low-cost extensions for public-key cryptography in Chapter 4 ([234]). This chapter illustrates how even tiny changes to a processor's architecture can lead to significant improvements. In Chapter 5 we demonstrate the versatility of custom instruction for public-key cryptography by applying them to speed up AES ([236]). Subsequently, we investigate the benefits of custom instructions for AES on 32-bit platforms with an analysis of memory benefits in Chapter 6 ([240]). More extensive support, different trade-off variants, and a thorough performance analysis are then presented in Chapter 7 ([237]). In Chapter 8, the synergies of secret-key and public-key support are investigated with the design of a flexible functional unit for multiply-accumulate operations ([239]). The focus is shifted to small 8-bit architectures in Chapter 9, where again support for AES is investigated ([242]). An evaluation and comparison of S-box hardware design options for implementation in a standard-cell technology follows in Chapter 10 ([232, 233]). This is not only relevant for the optimized implementation of our proposed instruction set extensions, but should also serve as a useful guideline in regard to many different AES hardware implementation options. The last chapters are dedicated to secure implementations in the face of side-channel attacks. Chapter 11 investigates the use of our proposed AES extensions for realizing effective SCA countermeasures ([243]). In Chapter 12 we deal with the question, how the implementation security can be further enhanced by hardware modifications ([238]). The effectiveness of some general software countermeasures in practically evaluated in Chapter 13 ([241]). We summarize the main contributions of this thesis and draw conclusions in Chapter 14.

2

Concepts and Methods of Modern Cryptography

Most public-key algorithms and also an increasing number of modern secret-key algorithms are defined in terms of arithmetic operations in finite algebraic structures. This chapter summarizes the most important properties of finite groups and finite fields as far as they relate to cryptographic algorithms. Based on this outline, state-of-the-art cryptographic algorithms are described. Emphasis will be given to Elliptic Curve Cryptography and the Advanced Encryption Standard, as they exhibit very favorable properties for implementation in constrained embedded devices.

2.1 Mathematical Preliminaries

A finite group consists of a finite set and an operation which maps two elements of the set to another element of the set. The group is hence said to be closed under the operation. Additionally, the operation must be associative, an identity element must exist, and each element must have an inverse element. If the operation is also commutative, the group is a *commutative group* or *Abelian group*. Commonly, the operations are denoted as addition or multiplication, which are used in additive groups and multiplicative groups, respectively. However, the actual operations can differ significantly from traditional addition and multiplication of integers or real numbers.

The *order* of a finite group is defined as the number of elements which it contains. When the operation is continually applied to a chosen element (say a) of the group (i.e., starting with the element a as initial sum or product, a

is added or multiplied continually), the resulting elements form a subgroup and a is denoted as generator of this subgroup. The order of the chosen element is defined to be the order of the subgroup. Hence, a generator of the finite group has the same order as the group. Groups which have a generator are denoted as *cyclic groups*.

A finite field features a finite set of elements and two operations, which we will denote as addition and multiplication. The set and each operation form finite Abelian groups, with the exception that the identity element of addition has no multiplicative inverse. Additional requirements are that the identity elements of addition and multiplication may not be equal and that multiplication is distributive over addition. If multiplication is not commutative and not each element has a multiplicative inverse, the structure is called a finite ring.

Finite fields are often denoted as Galois fields (GF). A finite field with q elements is denoted as $\mathrm{GF}(q)$ or $\mathbb{F}_q$. A fundamental result of group theory states that the order q of a finite field must be the power of a prime, i.e., $\mathrm{GF}(p^m)$ with p prime and $m \in \mathbb{N}$, see [158]. Finite fields of degree $m > 1$ are also called extension fields and their elements can be represented as polynomials of degree smaller than m with coefficients in $\mathrm{GF}(p)$. Finite fields of the same size are isomorphic, i.e., they can be transformed into one another, whereby their structure is preserved. This property of finite fields can be exploited for choosing computationally efficient representations on different platforms. Further details on finite fields can be found in standard literature, e.g., [158].

2.2 Principles of Public-Key Cryptography

In public-key cryptosystems, each participant is in possession of an individual pair of keys, i.e. the private and public key. While the private key must never be revealed, the public key can be distributed freely. The public key can be used to encrypt messages which are intended for the holder of the private key. Likewise, the private key can be used to digitally sign data and the signature can be verified with the public key. As the public key pair can be associated with a specific entity, it is possible to attribute specific actions involving the private key (e.g., digital signature generation) to that entity. This property cannot be achieved with secret-key methods.

All public-key algorithms make use of so-called *trapdoor one-way functions*. Such functions are deemed as hard to invert (hence one way) except where some additional information (trapdoor information) is known. This trapdoor information is used to generate the private key. The public key is connected to the private key and it is used to calculate the "easy" direction of the trapdoor one-way function. The private key can be used to calculate the "hard" inverse function. The relation of the private and public key allows to build schemes for encryption and decryption of messages as well as for digital signature generation and verification. The trapdoor one-way functions used in most common public-key cryptosystems are defined as arithmetic operations in finite groups or finite fields.

2.2.1 RSA

For RSA [208], modular exponentiation in a finite ring is used as one-way function. The finite ring is defined by a residue system modulo a product of two large primes. The elements of this residue system can be represented by the non-negative integers smaller than the modulus and the two operations are defined as modular addition and multiplication. The trapdoor information of RSA is the factorization of the modulus. The trapdoor one-way function is modular exponentiation with a public exponent. This function can be inverted by modular exponentiation with a corresponding private exponent[1], which can be derived from the public exponent if the factorization of the modulus is known.

2.2.2 Digital Signature Algorithm (DSA)

Similar to RSA, the DSA [184] works with modular exponentiation. As DSA uses a prime modulus, its underlying structure is a finite field. The public key is the exponentiation of the generator of a large cyclic subgroup of the multiplicative group with a private exponent. The trapdoor information is the discrete logarithm (i.e., the used private exponent) of the public key.

2.2.3 Elliptic Curve Cryptography (ECC)

The trapdoor one-way function of ECC [145, 173] is a *point multiplication* (*scalar multiplication*) on an elliptic curve. An elliptic curve is an additive Abelian group with the set of curve points and a corresponding point addition operation. For ECC, elliptic curves are defined over various finite fields. Elliptic curves will be discussed in a more formal way in Section 2.3.

2.2.4 Other Public-Key Cryptosystems

In addition to ECC, the use of hyperelliptic curves has been proposed for cryptography [146]. Hyperelliptic Curve Cryptography (HECC) allows for even smaller operands than ECC but the implementation of the arithmetic operations tends to be more complex.

The NTRU cryptosystem has been published by Hoffstein et al. in 1998 [120]. It relies on operations in the N-th degree truncated polynomial ring $\mathbb{Z}[x]/(x^N - 1)$.

Another modern cryptosystem is XTR, which also relies on the difficulty of the discrete logarithm problem [157]. XTR uses traces in order to calculate powers of finite field elements.

[1] In non-optimized implementations which do not make use of the Chinese Remainder Theorem (CRT).

2.3 Details on Elliptic Curve Cryptography

An elliptic curve over a field can be formally defined as the set of all solutions $(x, y) \in \mathbb{K} \times \mathbb{K}$ to the general (affine) Weierstraß equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.1}$$

with the coefficients $a_i \in \mathbb{K}$. If $\mathbb{K}$ is a finite field GF(q), then the set of all pairs (x, y) satisfying Equation (2.1) is also finite.

The set of all solutions $(x, y) \in \text{GF}(q) \times \text{GF}(q)$, together with an additional special point $\mathcal{O}$, which is called the *point at infinity*, forms an Abelian group whose identity element is $\mathcal{O}$. The group operation is the addition of points, which can be realized with addition, multiplication, squaring and inversion in GF(q). Depending on the type of the underlying finite field, a variety of algorithms for point addition exists, where each requires a different number of those field operations. If, e.g., the points on the elliptic curve are represented in *projective coordinates*, then the number of field inversions is reduced at the expense of additional field multiplications, see [112].

If the finite field is a binary extension field GF(2^m), then Equation (2.1) can be simplified to

$$y^2 + xy = x^3 + ax^2 + b \quad \text{with} \quad a, b \in \text{GF}(2^m) \tag{2.2}$$

The other common choice for the underlying finite field are prime fields GF(p), where p is a large prime. For this case (in fact for all GF(p^m) with $p > 3$) the general Weierstraß equation becomes

$$y^2 = x^3 + ax + b \quad \text{with} \quad a, b \in \text{GF}(p) \tag{2.3}$$

All EC cryptosystems are based on an computation of the form $Q = k \cdot P$, with P and Q being points on the elliptic curve and $k \in \mathbb{N}$. This operation is called point multiplication or scalar multiplication and is defined as adding P exactly $k - 1$ times to itself: $k \cdot P = P + P + \cdots + P$. The execution time of the scalar multiplication is crucial to the overall performance of EC cryptosystems. Scalar multiplication in an additive group corresponds to exponentiation in a multiplicative group. The inverse operation, i.e., to recover k given P and $Q = k \cdot P$, is denoted as the elliptic curve discrete logarithm problem (ECDLP), for which no subexponential-time algorithm has been discovered yet. More information on EC cryptography is available from various sources, e.g., [31, 112].

2.4 Modern Secret-Key Cryptography Algorithms

In secret-key cryptosystems, a key is always shared by two or more participants. This is especially useful for securing communication between these participants (e.g., to assure confidentiality and data integrity). As the key is in possession of

several entities, operations performed with it cannot be attributed to a specific entity in the way as public-key methods allow.

This section describes the most commonly used secret-key cryptographic primitives. A description of unkeyed hash algorithms is also included, as they bear many similarities with block ciphers. This selection is a limited excerpt of available block ciphers, hash functions, and stream ciphers. We will also present a short historical overview of the algorithms and their use in standards and applications.

2.4.1 Block Ciphers

As their name suggests, block ciphers operate on blocks of a fixed size. Each secret key defines a bijective mapping between input and output blocks of the same size. This mapping can also be seen as a permutation on the set of input blocks. Thus, the same input block will always yield the same output block, provided that the same secret key is used.

The Data Encryption Standard (DES) algorithm, which was approved in 1976 by the US National Institute of Standards and Technology (NIST) [183] has become the most important block cipher for several decades. Due to its limited key size of 56 bits and advances in technology, brute-force attacks became feasible with relatively low budget in the late 1990s. Therefore, DES was extended to Triple-DES in 1999. Two years later, DES was superseded by the Advanced Encryption Standard (AES) which offers larger key sizes (128, 192, and 256 bits). The Rijndael algorithm was selected as the winner of an open selection process initiated by NIST and has been standardized as the AES algorithm in the year 2001 [185]. During this selection process the four algorithms MARS [39], RC6 [207], Serpent [8], and Twofish [217] have also been selected as finalists with no discovered severe weaknesses. Hence, these algorithms are also worth to be considered in special environments like embedded systems and WSNs. RC6 has for example been shown to have very favorable properties on embedded processors with limited memory resources [104]. In comparison to Rijndael, which has the advantage of its efficient implementation on various platforms, Serpent has a higher security margin but is in turn much slower. Other interesting block ciphers for application in embedded systems are the Tiny Encryption Algorithm (TEA) [256] and its extensions (XTEA, XXTEA) [187, 257], Skipjack [182], and IDEA [154]. TEA is based on very simple and fast operations which require very little code size. Skipjack is used in TinySec [141] applications whereas IDEA is used in PGP.

2.4.2 Hash Functions

Hash functions belong to the category of unkeyed cryptographic primitives and are mainly used for providing data integrity and message authentication. The input of these primitives is a potentially large message which is condensed to a relatively short value. This so-called hash value can be seen as a fingerprint of

the message. One important property of a hash function is the collision resistance which means that it is computationally infeasible to find any two messages that produce the same hash value. This allows to detect data manipulation due to technical reasons or due to an attacker. A hash can be used to build a message authentication code (MAC) which allows to simultaneously verify data integrity and authenticity of a message. Due to the demanding design of hash functions regarding computing power and memory resources it might be favorable to minimize the use of hash functions and consider the use of block ciphers in an appropriate mode of operation.

For a long time, the design of cryptographic hash functions has received relatively little research effort. Most of the MD4-family hash functions were believed to be secure. So far, the best known hash functions belong to the MD4 family of hash functions: MD4 itself [205], MD5 [206], the SHA variants (SHA-1, SHA-256, SHA-384, and SHA-512) [186], and the RIPEMD variants (with 128, 160, 256, and 320 bits output) [33, 61]. All designs base more or less on the (today insecure) MD4 algorithm which was invented by Ron Rivest in 1990. Whereas MD5 is used only in a few applications, SHA-1 is widely employed in many security protocols and applications like SSH, S/MIME, SSL, TLS, and IPSec and the keyed-hash message authentication code (HMAC-SHA1) [153].

Since the publication of an attack against a reduced version of the SHA-1 algorithm [253], the cryptographic community continues to improve the attacks (e.g., [40]) and endeavors to design new, more secure hash algorithms. So far NIST recommends to use the more secure SHA-2 variants (SHA-256, SHA-384, SHA-512) or to switch to alternative designs like Whirlpool, which is standardized in ISO/IEC 10118-3 [135], or Tiger [7].

Some newer hash function designs include Grindahl, VSH, LASH, Radio-Gatun, and Maelstrom. The "Hash Function Zoo" website contains a comprehensible list of old hash functions and also more recent proposals [121]. Recently, NIST has announced that a selection process similar to that of the AES algorithm is planned for a new hash standard (SHA-3). The process will be a strong driver for new research in the field of hash functions.

2.4.3 Stream Ciphers

The name stream cipher derives from that fact that such a type of secret-key cipher encrypts the plaintext bits one at a time in a stream-like fashion. This is done mostly by applying an XOR operation with a key stream which depends on the current internal state. The main difference to block ciphers is that a stream cipher does not operate on large blocks of fixed length, but on individual bits. However, depending on the mode of operation, a block cipher can also be operated as a stream cipher. Typically, stream ciphers require less hardware resources and have a higher data throughput. However, stream ciphers tend to have a somewhat bad reputation because of security problems that arise when they are used incorrectly and because the most famous stream cipher RC4 has been shown to have some weaknesses. Although RC4 is widely applied in security protocols like SSL and WEP, its usage is not recommended in new applications.

Other stream ciphers are currently used in GSM technology, e.g., A5/1 and A5/2 (both of which are broken).

From 2004 to 2008, the European Network of Excellence ECRYPT has organized a contest called eSTREAM [66] in order to identify new stream ciphers for future standardization. In addition to a high security level, the ciphers should be suited for fast implementation in software (profile 1) or efficient implementation in hardware (profile 2). After three phases, the final portfolio chosen by eSTRAM consists of eight algorithms (four of each profile). For the software profile, HC-128, Rabbit, Salsa20/12, and SOSEMANUK were chosen. For the hardware profile, the portfolio consists of F-FCSR-H (v2) , Grain (v1), MICKEY (v2), and Trivium.

2.5 Description of the Advanced Encryption Standard

The block cipher Rijndael was designed by Joan Daemen and Vincent Rijmen [57]. It has an iterated round structure and its block size and key size can be selected independently in the range of 128 bits to 256 bits in 32-bit increments. The number of rounds depends on the chosen values for block size and key size and varies between 10 and 14.

A subset of Rijndael has been chosen by NIST to become the new Advanced Encryption Standard (AES) algorithm in the year 2001. AES is defined in the Federal Information Processing Standard FIPS-197 [185] to replace the ageing Data Encryption Standard (DES) [183]. The block size of AES has been fixed to 128 bits and the choice of key size has been narrowed down to 128, 192, and 256 bits. The corresponding numbers of rounds for these key sizes are 10, 12, and 14, respectively. In the following, we limit the scope to AES, but most descriptions apply to Rijndael as well.

The AES algorithm is very flexible and can be implemented very efficiently in software and hardware. AES is well-suited for software implementation both on small 8-bit microcontrollers as well as on more powerful 32-bit processors. Dedicated hardware implementations of AES can be scaled in size and speed to satisfy specific constraints. The free availability of the algorithm paved the way for deployment of AES in numerous standards like wireless LAN according to the IEEE802.11i standard [127], IPSec [143], and TLS [60].

One of the strengths of AES lies in its simplicity which was one of its main design principles [57]. This simplicity is achieved by the reliance on a small set of basic operations and the utilization of symmetry properties at different levels.

2.5.1 Principal Structure

All operations of the AES algorithm transform a block of data, which is referred to as the *State*. The State is organized as a 4×4 matrix of bytes. At the start of encryption, the 128-bit State is initialized with the plaintext. Then, the round transformations are applied to modify the 128-bit State and to finally

yield the output ciphertext. Some of these transformations are dependent on the cipher key, which thus determines the actual mapping from plaintext blocks to ciphertext blocks. Decryption is similar to encryption: The State is initialized with the ciphertext and the inverse transformations are applied in reverse order.

The non-linear, linear, and key-dependent transformations in each round are defined as operations over various finite fields and rings ($\mathbb{F}_2$, $\mathbb{F}_{2^8}$, $\mathbb{F}_{2^8}[t]/(g(t))$). The field $\mathbb{F}_{2^8}$ is defined by the reduction polynomial $f(x) = x^8+x^4+x^3+x+1$ and the ring $\mathbb{F}_{2^8}[t]/(g(t))$ by the reduction polynomial $g(t) = t^4 + 1$. The operations are called SubBytes, ShiftRows, MixColumns, and AddRoundKey for encryption and InvSubBytes, InvShiftRows, InvMixColumns, and AddRoundKey for decryption. All operations process the bytes of the State either individually, row-wise, or column-wise. Before the first round of encryption, an initial AddRoundKey is performed and in the last round of encryption the MixColumns operation is omitted.

In each round, a 128-bit *round key* is combined with the State in the AddRoundKey transformation. All round keys are derived from the cipher key in an operation called *key expansion* or *key scheduling*. For Nr rounds, AES requires $Nr + 1$ round keys, i.e., one key per round and an extra one for the initial AddRoundKey. The full list of all round keys is denoted as *key schedule*.

The structure of AES is shown in Figure 2.1, which includes the pseudocode for encryption. The constant Nr contains the number of rounds. The array w contains the key schedule in the form of 32-bit words. A range of words from x to y of this array is addressed in the form $w[x..y]$. Consequently, $w[0..3]$ contains the first round key, $w[4..7]$ the second round key, etc.

The pseudocode for AES decryption is shown in Figure 2.2. The shown structure simply uses the inverse transformations in the inverse order of encryption and applies the round keys accordingly in the inverse order. In can be seen that within a single round, the order of inverse transformations in decryption (InvShiftRows, InvSubBytes, AddRoundKey, InvMixColumns) is different from the order of the respective transformations in encryption (SubBytes, ShiftRows, MixColumns, AddRoundKey).

For some implementations it can be favorable if decryption uses the same order of (inverse) transformations as encryption. In AES, this can be achieved with the so-called *Equivalent Inverse Cipher* structure which is shown in Figure 2.3.

Note that the transformations InvShiftRows and InvSubBytes as well as AddRoundKey and InvMixColumns are pairwise swapped. As InvShiftRows and InvSubBytes are independent of each other (byte rotation vs. byte substitution), they can be swapped without constraints. In order to exchange the order of AddRoundKey and InvMixColumns it is necessary to modify the concerned round keys by applying the InvMixColumns transformation to them.

2.5.2 Round Transformations

In this Section, the round transformations of AES are described in more detail. Also common implementation options will be given.

```
AES_encrypt(byte in[16], byte out[16], word w[4*(Nr+1)])

begin
  byte state[4,4];
  state = in;

  AddRoundKey(state, w[0..3]);

  for round = 1 step 1 to Nr-1
    SubBytes(state);
    ShiftRows(state);
    MixColumns(state);
    AddRoundKey(state, w[4*round..4*round+3]);
  end

  SubBytes(state);
  ShiftRows(state);
  AddRoundKey(state, w[4*Nr..4*Nr+3];

  out = state;
end
```

Figure 2.1: Pseudo code for AES encryption.

SubBytes

The SubBytes transformation is the only non-linear operation of AES. It processes the 16 bytes of the State individually with the help of a single substitution table. This table is defined by an inversion in the binary extension field $\mathbb{F}_{2^8}$, followed by an affine transformation. This byte substitution is often referred to as *S-box lookup*. The operation of SubBytes is depicted in Figure 2.4.

There are different possibilities to implement the AES S-box. In software, a table containing the 256 different values is normally used. The table can be part of the executable and stored in the program memory. Alternatively, the S-box table can be calculated at runtime and stored in the data memory. The SubBytes transformation can also be combined with MixColumns with the help of larger lookup tables (T-tables). For hardware implementations there is a much larger design space. An analysis of the different design options is done in Chapter 10.

ShiftRows

The ShiftRows transformation is a simple operation which rotates each row of the State to the left using a specific byte offset. The offset is given by the row index (starting at 0), which means that the first row is not rotated at all and the last row is rotated by three bytes. The principal functionality is shown in Figure 2.5.

```
AES_decrypt(byte in[16], byte out[16], word w[4*(Nr+1)])

begin
  byte state[4,4];
  state = in;

  AddRoundKey(state, w[4*Nr..4*Nr+3]);

  for round = Nr-1 step -1 to 1
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(state, w[4*round..4*round+3]);
    InvMixColumns(state);
  end

  InvShiftRows(state);
  InvSubBytes(state);
  AddRoundKey(state, w[0..3];

  out = state;
end
```

Figure 2.2: Pseudo code for AES decryption.

The implementation in software is reduced to an appropriate addressing of the bytes in the subsequent operation. Normally, ShiftRows is done in combination with the S-box or T-table lookup. In hardware, the architecture determines whether ShiftRows can be implemented as a simple wiring or whether an appropriate addressing of the State storage is required.

MixColumns

MixColumns operates on each column of the State separately. Each column is interpreted as a polynomial of degree 3 with coefficients from the field $\mathbb{F}_{2^8}$, i.e., $\mathbb{F}_{2^8}[t]/g(t)$ with $g(t)$ as reduction polynomial. This operation is illustrated in Figure 2.6. Elements of the field $\mathbb{F}_{2^8}$ are usually represented as binary polynomials of degree smaller or equal to 7. We write constants from this field in hexadecimal notation, where each bit stands for a coefficient of the binary polynomial. For example, $0B = (00001011)_2 = x^3 + x + 1$

MixColumns is defined as a multiplication of this polynomial with the constant polynomial $(03 \cdot t^3 + 01 \cdot t^2 + 01 \cdot t + 02)$ modulo the polynomial $g(t) = t^4 + 1$. The operation can also be expressed as a matrix multiplication of a constant 4×4 matrix with the input column. The matrix multiplication for MixColumns can be seen in Equation 2.4, where a_0 to a_3 are the $\mathbb{F}_{2^8}$ coefficients of the input column and b_0 to b_3 are the $\mathbb{F}_{2^8}$ coefficients of the output column. The inverse

```
AES_decryptEQ(byte in[16], byte out[16], word w[4*(Nr+1)])

begin
  byte state[4,4];
  state = in;

  AddRoundKey(state, w[4*Nr..4*Nr+3]);

  for round = Nr-1 step -1 to 1
    InvSubBytes(state);
    InvShiftRows(state);
    InvMixColumns(state);
    AddRoundKey(state, w[4*round..4*round+3]);
  end

  InvSubBytes(state);
  InvShiftRows(state);
  AddRoundKey(state, w[0..3];

  out = state;
end
```

Figure 2.3: Pseudo code for AES decryption with Equivalent Inverse Cipher structure.

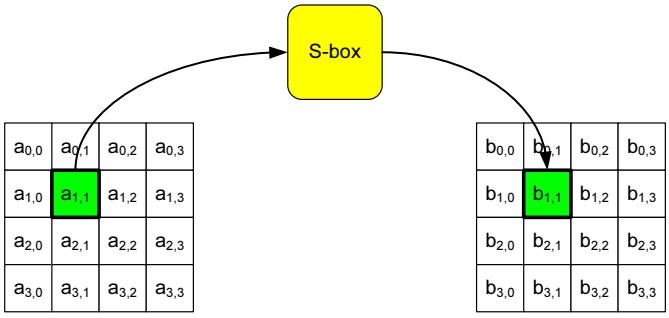

Figure 2.4: SubBytes transformation.

matrix used in InvMixColumns is shown in Equation 2.5.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{2.4}$$

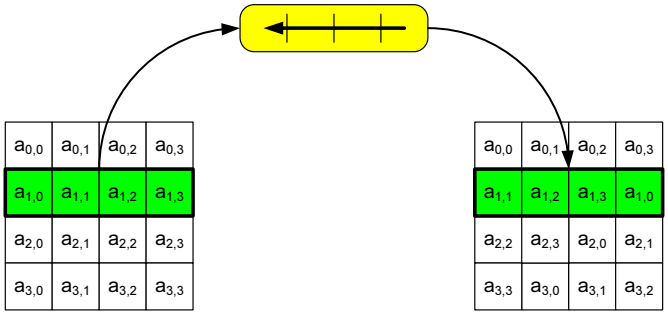

Figure 2.5: ShiftRows transformation.

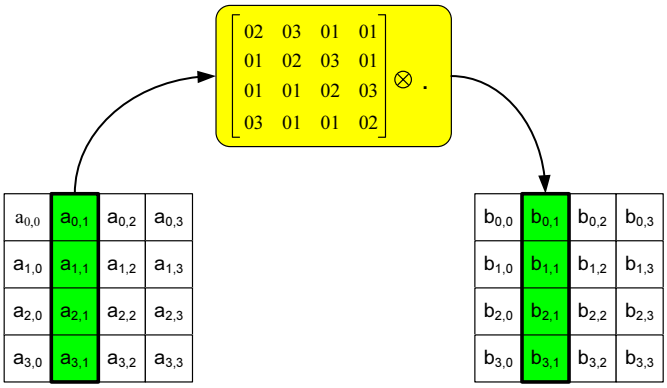

Figure 2.6: MixColumns transformation.

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{2.5}$$

MixColumns is normally the most expensive operation is software, as multiplication in binary extension fields is not supported by most processors. Therefore, each multiplication of elements of $\mathbb{F}_{2^8}$ needs to be realized with a number of shift and XOR instructions. The cost for InvMixColumns is even higher, as the according inverse matrix contains larger constants which require an increased computational effort (cf. Equation 2.5). MixColumns and its inverse can be implemented with table lookups by using the approach with T-tables [57]. For 32-bit platforms, Bertoni et al. have proposed to speed up MixColumns and InvMixColumns through row-wise processing [25]. Hardware implementations of MixColumns tend to be very efficient as they consist mainly of a relatively small number of XOR gates.

The three operations SubBytes, ShiftRows, and MixColumns constitute the

substitution-permutation network of the AES algorithm. SubBytes makes up the substitution part which is intended to minimize the information about the cipher key contained in the ciphertext (confusion). ShiftRows and MixColumns build the permutation part which is expected to break the link between the plaintext and its corresponding ciphertext (diffusion).

AddRoundKey

AddRoundKey simply adds a round key to the State. It is an XOR operation over all 128 bits. In each round, a new round key is used for this transformation. The AddRoundKey transformation is depicted in Figure 2.7. The $Nr + 1$ round keys are derived iteratively from the cipher key.

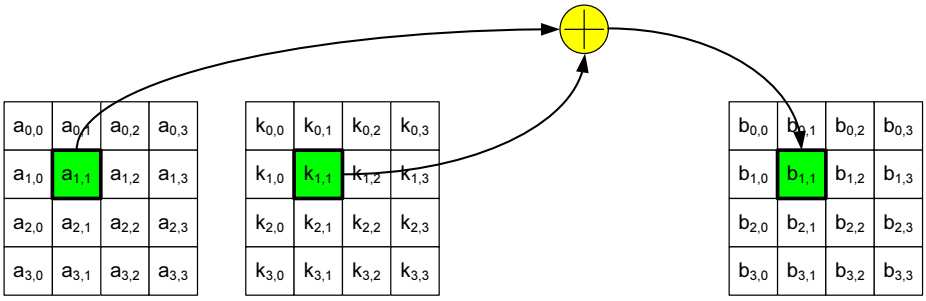

Figure 2.7: AddRoundKey transformation.

Key Expansion

The key expansion mainly consists of XOR operations. Additional operations include the application of the S-box, a byte-wise rotation and addition of some round constants (*Rcon*). The pseudocode for the key expansion is shown in Figure 2.8, where Nk is the number of 32-bit words in the cipher key, i.e., 4, 6, or 8.

The key schedule (i.e., the list of all round keys) is initialized with the cipher key. The rest of the key schedule is then derived successively from the existing part. The first round key equals the first 128 bits of the cipher key. The course of the key expansion varies slightly with the size of the cipher key.

There are two main implementation options to handle the generation and use of round keys. When the same cipher key is used repeatedly and the system has sufficiently fast memory resources, then it is common to use a *precomputed key schedule*. The other possibility is the *on-the-fly key expansion* where the current round key is calculated on demand during the encryption or decryption operation. This variant allows to save memory resources but may in turn reduce the cipher's performance.

```
AES_expand_key(word key[Nk], word w[4*(Nr+1)], Nk)

begin
  word temp
  i = 0

  while (i < Nk)
    w[i] = key[i]
    i = i+1
  end while

  i = Nk

  while (i < 4*(Nr+1))
    temp = w[i-1]
    if (i mod Nk = 0)
      temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
    else if (Nk > 6 and i mod Nk = 4)
      temp = SubWord(temp)
    end if
    w[i] = w[i-Nk] xor temp
    i = i + 1
  end while
end
```

Figure 2.8: Pseudo code for AES key expansion.

3

Cryptography and Instruction Set Extensions

In this Chapter, we give an overview of the principal implementation approaches for cryptography in embedded systems. The design and implementation of instruction set extensions for cryptography is then described in more detail. This starts from the identification of software bottlenecks and candidate instructions, over the development and integration of appropriate functional units into the processor, down to adaptation of the toolchain and performance evaluation on a prototyping platform.

3.1 Implementation Options for Cryptographic Algorithms

This section will give an overview of the general approaches for implementing cryptographic algorithms. Subsequently, all options will be evaluated in regard to constrained embedded systems.

Implementations of cryptographic algorithms can be roughly classified into five general categories. The following list cites them with decreasing amount of dedicated cryptographic hardware.

- Application-Specific Integrated Circuit (ASIC)
- Application-Specific Instruction Set Processor (ASIP) and Application Domain-Specific Processor (ADSP)
- General-Purpose Processor with Coprocessor (GPP+COP)

- General-Purpose Processor with Instruction Set Extensions (GPP+ISE)
- General-Purpose Processor (GPP)

An *Application-Specific Integrated Circuit* consists only of hardware dedicated to one or several specific algorithms. *Application-Specific Instruction Set Processors* and *Application Domain-Specific Processors* are processors with an instruction set architecture (ISA) specifically tuned for a concrete application or application domain, which may also feature additional optimizations of the microarchitecture. A cryptographic *Coprocessor* can be attached to a *General-Purpose Processor* in order to enhance the performance of specific algorithms. *Instruction Set Extensions* feature an even tighter coupling of application-specific hardware to the general-purpose processor by providing new instructions in addition to the original ISA. The last design option is an unmodified general-purpose processor where all algorithms are executed using the defined ISA. All these approaches have different characteristics regarding *performance*, *memory requirements*, *flexibility/scalability*, *energy efficiency*, and *cost*.

The border between coprocessor and instruction set extension is not completely sharp as coprocessor solutions might also be accompanied by the addition of custom instructions (e.g., for controlling the coprocessor's functionality). A good criterion to categorize a specific solution is to look at the control mechanism of the cryptographic computations. If this control is mainly external to the processor pipeline, the solution is rather a coprocessor. A typical pattern for a coprocessor usage just consists of a few simple steps: Configure coprocessor functionality (e.g., select encryption or decryption), load input data, start operation, and retrieve results. On the other hand, if the pipeline retains fine-grain control over the cryptographic computations, the solution is rather in the domain of instruction set extensions. Typically, the custom instructions perform relatively small steps of the algorithm (e.g., part of a round transformation of a block cipher).

On desktop systems, performance is measured in terms of throughput and is traditionally the main criterion in cryptographic systems (i.e., "the faster, the better"). This stance is especially typical for cryptographic software for mainly two reasons. First, resources like working memory, program memory, and available energy are practically unbounded. Second, although the speed and execution parallelism of desktop processors have been increasing steadily, the performance of cryptographic software continues to lag behind the performance of network connections. For example, the throughput for the AES block cipher on a modern desktop CPU currently peaks in the 1-2 Gbit/s range [160, 167], while Ethernet has already reached throughputs of 10 Gbit/s [128], with work already commencing on speeds up to 100 Gbits/s [124].

However, in embedded systems throughput is not the single goal of optimization. These systems normally have very limited memory and energy resources which need to be used carefully by a cryptographic implementation (and in fact by any application running on the embedded device). The most important goal is to reach an adequate throughput sufficient for the particular application at

hand with minimal use of memory and energy resources. Furthermore, as most embedded devices must be inexpensive, the additional cost for cryptographic hardware must henceforth be low. Another important aspect for embedded systems is flexibility and scalability of cryptographic processing.

Flexibility refers to a system's capability to process a variety of tasks from an application domain, e.g., support of different ECC parameters, or even from different applications domains, e.g., multimedia, digital signal processing, and cryptography. The required degree of flexibility is highly dependent on the intended application of the system. While some embedded systems with fixed processing requirements (e.g., within household appliances) have a limited need for flexibility, other systems (e.g., multi-purpose smart-cards, set-top boxes, game consoles, cell phones, PDAs) may require a very high flexibility.

Scalability is concerned with the provision of different variants of a specific cryptographic algorithm. A typical example is a block cipher with a varying key size and number of rounds, as it is offered by many modern algorithms. A scalable solution can be adapted to the use of different key sizes, which offer a varying degree of security.

Energy efficiency refers to the amount of processed data in relation to the required energy. As a general rule, energy efficiency becomes better with a rising degree of dedicated hardware. However, this is only true within the domain for which the system is built. If, for example an ASIP is forced to do general-purpose processing, its energy efficiency may fall well below that of a GPP. Energy efficiency is strongly related to performance, as fast processing of an algorithm also reduces the total amount of energy spent on the task [92].

When considering cost, one has to distinguish different contributing factors; mainly design cost and implementation cost. Design cost refers to the cost for building or purchasing the solution. While GPPs, GPPs with ISE and GPPs with coprocessors may be readily available as intellectual property (IP) cores, ASIP/ADSPs and ASICs normally require a greater design effort. Implementation cost refers to the cost for realizing the system. Solutions involving GPPs may already be available as relatively cheap chips, whereas application-specific solutions will most likely require custom implementation by the system manufacturer.

In summary, embedded devices require cryptographic implementations which offer good properties in the following regards:

- Throughput
- Memory usage (both working memory and program memory)
- Flexibility/scalability
- Energy efficiency
- Cost

Finding a satisfying mix of all these properties is a very difficult task. Figure 3.1 gives a rough overview of the design space for cryptographic systems. The

five key characteristics are also shown for the different design approaches. It is however important to note, that this assessment reflects only a typical trend and is not an absolute measure. It may well be that a specific GPP with ISE may outperform a GPP with coprocessor or that the cost of a specific ASIP is higher than that of an ASIC.

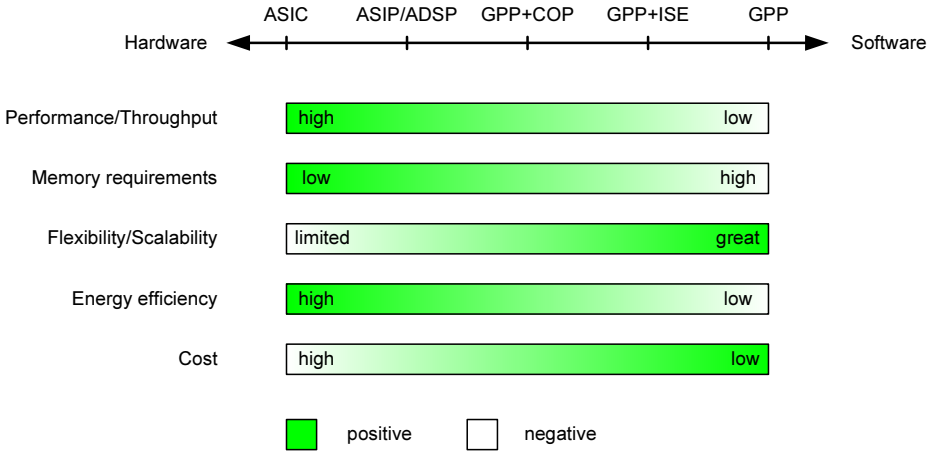

Figure 3.1: Design criteria for cryptographic systems.

3.1.1 Application-Specific Integrated Circuit

ASICs are custom-designed and custom-built chips which comprise a dedicated functionality. They are usually added to a system to satisfy performance demands and/or to increase the energy efficiency for a specific workload. There have been numerous publications about ASIC implementation of cryptographic algorithms. Fabricated ASICs for public-key algorithms (e.g., [21]) and secret-key algorithms (e.g., [107, 179, 220]) have been presented. Furthermore there exist high-speed implementations of hash functions (e.g., [56]). In addition, some ASICs have been designed to cater for complete security protocols like IPSec [109].

For embedded systems the prospect of increased energy efficiency is a very big advantage of ASIC implementation. However, the stringent development and implementation cost of cryptographic ASICs is contrary to the low-cost criterion. Furthermore, an additional chip could lead to an unacceptable enlargement of the embedded device. Multi-chip packaging could be used to circumvent space restrictions, but this would again increase the cost of the device. Therefore, cryptographic ASICs are not an optimal choice for most embedded devices.

3.1.2 Application-Specific Instruction Set Processors / Application Domain-Specific Processors

Two prominent examples of ASIP architectures for symmetric cryptography are the Cryptonite processor [37, 189] and the CryptoManiac processor [254, 262]. For asymmetric algorithms there have also been proposed architectures [64]. In [144], a processor for both symmetric and asymmetric cryptography is presented. A methodology for designing ASIPs through gradual refinement has been presented in [211, 212].

Goodman et al. presented a reconfigurable cryptographic processor which integrated polynomial operations into an integer modular multiplier [85, 86]. The processor was denoted *Domain Specific Reconfigurable Cryptographic Processor (DSRCP)*. It includes a bit-serial multiplier for operands of up to 1,024 bits, which can be adapted in 32-bit steps to the actual operand size. The design of the DSRCP is described in detail in Goodman's PhD thesis [87].

Another interesting architecture is the PAX processor [72, 78]. Although labeled "cryptographic processor", PAX also features a number of general-purpose RISC instructions in addition to instructions for cryptography. A unique feature of PAX is its datapath scalability. The datapath width can be configured to 32, 64, or 128 bits.

ASIPs normally offer greater flexibility than ASICs and there is also a broader tool support available for their development. However, compared to a general-purpose processor, the scope of applications is still narrow.

3.1.3 General-Purpose Processor with Coprocessor

A large number of designs and implementations of cryptographic coprocessors have been reported in the literature. As practically every cryptographic design can be used as coprocessor by adding some control logic, these solutions can vary considerably in their approach. The spectrum of functionality ranges from basic modular arithmetic to full-blown ASIPs in the form of a coprocessor. Normally the designs are specialized to either symmetric cryptography [69, 117, 118, 200, 247] or asymmetric cryptography [2, 68, 264].

Some publications have shown that the use of coprocessors can suffer from a serious disadvantage. The processing speed of the coprocessor can be severely limited by the data rate of its connection to the general-purpose processor. In [119] and [213], it was shown for an AES coprocessor that even dedicated high-speed interfaces may well be nearly two orders of magnitude slower than the coprocessor itself. This result provides a strong argument for a tighter coupling of the custom hardware to the processor's pipeline whenever there is the need for a substantial interaction between GPP and cryptographic hardware.

3.1.4 General-Purpose Processor with Instruction Set Extensions

Instruction set extensions for cryptography are a hybrid approach lying between pure-software solutions and coprocessor systems. In the current literature there is a distinction between two basic methodologies for selecting an efficient set of custom instructions. The first approach leaves instruction selection up to the designer while the second approach relies on automatic selection via benchmarking of program code which is "typical" for the targeted application domain. Different approaches for automatic selection have been described in [50, 51, 142, 202]. The Swiss Federal Institute of Technology Lausanne (EPFL), partly in cooperation with the University of California, Irvine has also released a number of publications regarding automatic generation of instruction set extensions [13, 29, 30, 198, 199].

On the one hand, automatic selection of custom instructions can yield substantial improvements when the benchmarked code represents the requirements of the application domain in a sufficient manner. But in the general case, where the designer is not familiar with the application domain, this automatic selection may well fall short of the optimal solution. A good example of this problem is the case of extensions for Elliptic Curve Cryptography, where a relatively slow pure software solution can be sped up to match the performance of the best algorithm, if both are using instruction set extensions (cf. Chapter 4). If just the fastest algorithm is used for benchmarking, the resulting extensions may not be able to deliver the optimal performance improvement for the application domain.

Much research has been done involving manual design of instruction set extensions. Chapter 1 has given a fairly extensive overview of the available work in this area. Although manual design of extensions takes more time than automatic selection, its results are more likely to be nearer to the optimal solution for the application domain. A good example for this are the AES extensions for 32-bit processors described in Chapter 7.

3.1.5 General-Purpose Processor

Efficient software implementations of cryptographic algorithms are the subject of a large number of publications. In the field of symmetric cryptography, the selection process for the Advanced Encryption Standard (AES) by the U.S. National Institute of Standards and Technology (NIST) has drawn much attention from the cryptographic research community. There have been numerous publications during the AES selection process which have focused on efficient software implementations of the candidate algorithms, e.g., [9, 83, 110, 216].

After the selection of Rijndael as the AES algorithm in 2001 [185], efficient software implementations on various platforms have received considerable attention [12, 25, 82]. The AES algorithm is assumed to be in widespread use throughout the next decades.

The reduced resource demand of Elliptic Curve Cryptography (ECC) [145,

173] in comparison to traditional asymmetric cryptosystems like RSA [208] and DSA [184] makes it an attractive alternative for embedded and constrained devices. ECC has been the topic of a great number of publications regarding both efficient algorithms, e.g., [111, 161, 188, 255, 260], and implementations on different platforms, e.g., [34, 77, 106, 113, 114, 116, 261]. An important property of ECC is the large degree of freedom offered to the system designer. On the one hand, mathematical properties of different elliptic curves and underlying fields can be exploited to allow efficient implementations on specific platforms, e.g., [106, 261]. On the other hand, this can pose problems when a device needs to support various sets of parameters.

3.2 Design and Implementation Approach for Instruction Set Extensions

As already mentioned, the design of instruction set extensions is normally motivated by the fact that the original Instruction Set Architecture (ISA) of a processor and the implementing microarchitecture are deemed inadequate for an application or a class of applications. The primary goal is normally the increase of performance, but the reduction of memory requirements could also be an important decision factor.

The first step in the design is the study of typical software implementations on the processor at hand in order to identify the main bottlenecks. Typically, such bottlenecks can be removed successfully when most of the time is spent in small sections of the code. The focus should be kept on the original application with several implementation options and not set on a single option. Specifically, it should be avoided to concentrate the study on measures to increase performance of the fastest software implementation for the given architecture. In any case, the survey should include a generic implementation, even if its performance in pure software is less than that of an optimized version. Such a generic implementation tends to offer much broader opportunities for acceleration with instruction set extensions. In Chapter 4 such a strategy is shown to be very beneficial as both generic and optimized implementations can benefit from the extensions. In the end, the generic implementation with instruction set extensions yields similar performance as the optimized version.

Once suitable instructions and microarchitectural changes have been identified, their effects on performance can be estimated with the help of instruction-set simulators (ISS) which can incorporate the proposed changes. Such simulations can already deliver results for performance, memory requirements, and possibly energy consumption and facilitate the selection between different solutions. If several solutions promise equal benefits, then the decision can be made according to additional factors, e.g., cost of implementation.

With the desired functionality of the instruction set extensions established, the custom instructions need to be integrated into the processor. Which changes need to be made depends very much on the nature of the new instructions.

Common modifications encompass the decode logic and the functional units (FUs). The decode logic needs to recognize the new operation codes (opcodes) of the added instructions and set control signals accordingly. The actual work of the instructions is then done within modified or new FUs. Other changes might be the addition of custom registers or alternate access modes to the existing register file. The number of read and write ports of the register file might also be affected by the changes.

During the development of the FUs, it should already be possible to get a good estimation of the feasibility of implementation under given constraints (e.g., silicon area, allowable critical path delay). If an option turns out to be too costly it could be abandoned and an alternative from the previous design space exploration could be chosen instead.

When instructions are added to the ISA of a processor, it is also necessary to enhance the toolchain (compiler, assembler, linker) so that software developers can make use of the new functionality. The easiest option is to integrate support for the new instructions into the assembler, as it usually offers a much simpler structure than the compiler. The custom instructions can then be used in assembly programs as well as in higher-level languages (e.g., C) which allow inline assembly constructs. An according modification of the compiler offers more comfort to the software designer but is also more complicated and error-prone. With the help of a modified toolchain, applications which use the extensions can be translated into object code. The final code size of the application can be extracted from the object code and for most instances, the RAM size can also be estimated.

The enhanced processor can be implemented in hardware (e.g., FPGA prototyping, standard-cell synthesis). This yields very accurate figures for area and timing behavior for the given technologies and might at least allow estimations for different technologies. Software which makes use of the instruction set extensions can be run on the hardware in order to gather accurate performance figures. The RAM size can be determined accurately and also the power and energy consumption can be estimated.

We have followed the outlined approach to design and implement various instruction set extensions for ECC and AES on different platforms. Our main target was the SPARC V8-compatible LEON2 embedded processor [80], where we enhanced the GNU toolchain (gcc and binutils) to enable software development with the cryptography extensions. A version of the LEON2 with all our added enhancements has been made available under the LGPL in the context of our project *Instruction Set Extensions for Cryptography (ISEC)* [134]. This processor variant is denoted LEON2-CIS, where CIS stands for *Cryptography Instruction Set*.

4

Instruction Set Extensions for Public-Key Cryptography

In Section 1.6 we have already given an overview of research work on extensions for public-key cryptography. In this chapter we will describe the most important results and present the contributions of our work.

A common challenge of implementing public-key algorithms in software is that they require the processing of very large numbers. For public-key cryptosystems based on the integer factorization problem (e.g., RSA) or the discrete logarithm problem (e.g., DSA), the required arithmetic operations are on large integers (typically between 1,024 and 3,072 bits). Cryptosystems based on the elliptic curve discrete logarithm problem use significantly smaller values (typically between 160 and 250 bits), but these are still much larger than the native word size of embedded general-purpose processors. Therefore, it is necessary to represent these large values in the form of multiple words. Arithmetic operations performed on such values are hence denoted as multiple-precision operations (where precision refers to the maximal operand size which the processor can handle natively).

Analysis of the workload of software implementations of public-key algorithms has shown that most time is spent in a few small code sections. Typically, these code sections are the inner loops found in multiple-precision operations. Therefore, most architectural enhancements for public-key cryptography have focused on speeding up these critical portions of the program code. Another goal was to make use of the similarities of the critical operations of the main public-key cryptosystems in order to design a small set of custom instructions to support a broad range of these systems.

4.1 Related Work on Public-Key Extensions

After Dhem proposed the integration of a $(32 \times 32 + 32 + 32)$-bit multiply-accumulate operation for long integer modular multiplication [59], Großschädl et al. continued work into this direction by investigating different algorithms for multiple-precision modular multiplication [90, 95, 98].

Our cooperation with Johann Großschädl centered around the evaluation of the energy efficiency of algorithms for multiple-precision multiplication, reduction, and modular multiplication [92, 102].

For non-integer arithmetic, Nahum et al. generally suggested support for binary extension fields GF(2^m) [181] and Koç and Acar identified multiplication of two word-size binary polynomials (MULGF2) as a worthwhile target for optimization [43].

While Drescher et al. were the first to propose integration of GF(2^m) multiplication into conventional integer multipliers, Bucci proposed a *dual-mode* modular multiplier specifically for cryptographic applications. Similar concepts were used by Goodman et al. for their Domain Specific Reconfigurable Cryptographic Processor (DSRCP) [85, 86].

Savaş et al. developed *word-level* algorithms for Montgomery multiplication in GF(p) and GF(2^m) and also proposed a multiplier suited for both types of fields [210]. They also introduced the concept of *dual-field* adder cells, which can suppress carry propagation for GF(2^m) operations.

Großschädl et al. investigated the efficiency of a `mulgf2` instruction on a 16-bit platform [96] and proposed a low-power unified multiply-accumulator design for 32-bit processors [97]. Arithmetic support for both GF(p) and GF(2^m) was considered by Großschädl et al. in [101]. The authors proposed a set of seven instructions working on a 72-bit accumulator (four for integer arithmetic, two for binary polynomials, and one for general accumulator manipulation).

While Fiskiran and Lee examined the efficiency of instruction set extensions for GF(2^m) for varying word sizes and processor parallelism [71], Eberle et al. considered similar support for 8-bit microcontrollers [65].

Support for less common finite fields like Optimal Extension Fields and ternary extension fields GF(3^m) has also been investigated in [99] and [249], respectively.

Our joint work with Johann Großschädl investigated low-cost support for GF(2^m) operations [234], which is described in the remainder of this chapter, and the synergies of support for digital signal processing and cryptography [100].

4.2 Low-Cost Instruction Set Extensions for ECC over GF(2^m)

Elliptic Curve Cryptography (ECC) is especially attractive for devices which have restrictions in terms of computing power and energy supply. The efficiency of ECC implementations is highly dependent on the performance of arithmetic operations in the underlying finite field. In this section we present a simple

architectural enhancement to a general-purpose processor core which facilitates arithmetic operations in binary extension fields GF(2^m). A custom instruction for a multiply step for binary polynomials has been integrated into a SPARC V8 core, which subsequently served to compare the merits of the enhancement for two different ECC implementations. One was tailored to the use of GF(2^{191}) with a fixed reduction polynomial. The speed of this implementation was almost doubled while its code size was reduced. The second implementation worked for arbitrary binary fields with a range of reduction polynomials. The flexible implementation was accelerated by a factor of nearly 10.

Binary extension fields GF(2^m) allow efficient representation and computation on a general-purpose processor which does not feature a hardware multiplier. They are therefore well suited to be used as the underlying field of an elliptic curve cryptosystem.

ECC implementations require several choices of parameters regarding the underlying finite field (type of the field, representation of its elements, algorithms for the arithmetic operations) as well as the elliptic curve (representation of points, algorithms for point arithmetic). If some of these parameters are fixed, e.g., the field type, then implementations can be optimized yielding a considerable performance gain. Such an optimized ECC implementation will mainly be required by constrained end devices in order to cope with their limited computing power.

Various standards have included recommendations for specific sets of parameters, e.g., ANSI X9.62 and X9.63 [6, 5], NIST FIPS-186 [184], IEEE 1363 and 1363a [125, 126], and SEC 2 [45]. As research in ECC advances, new sets of parameters with favorable properties are likely to become available and recommended. Therefore, not all end devices will use the same set of parameters. Server machines which must communicate with many different clients will therefore have a need for flexible and yet fast ECC implementations.

We propose a simple extension to a general-purpose processor to accelerate the arithmetic operations in binary extension fields GF(2^m). Our approach concentrates on a very important building block of these arithmetic operations; namely the multiplication of binary polynomials, i.e., polynomials with coefficients in GF(2) = $\{0, 1\}$. If this binary polynomial multiplication can be realized efficiently, then multiplication, squaring and inversion in GF(2^m) and in turn the whole ECC operation is made significantly faster.

Two forms of a multiply-step instruction are proposed, which can be implemented and used separately or in combination. These instructions perform an incremental multiplication of two binary polynomials by processing one or two bits of one polynomial and accumulating the partial products. A modified ripple-carry adder is presented which facilitates the accumulation with little additional hardware cost.

The proposed custom instructions have merits for implementations which are optimized for specific binary finite fields with a fixed reduction polynomial. Also, flexible implementations which can accommodate fields of arbitrary length with a range of reduction polynomials benefit from such instructions. Note that both

such types of implementations are general enough to support different elliptic curves and EC point operation algorithms.

Section 4.3 outlines important aspects of modular multiplication in GF(2^m). Section 4.4 describes the proposed custom instructions in detail. A possible implementation of the extensions with a modified ripple-carry adder is presented in Section 4.5. Section 4.6 gives evaluation results from our implementation. Finally, a summary and conclusions are given in Section 4.7.

4.3 Arithmetic in Binary Extension Fields

A common representation for the elements of a binary extension field GF(2^m) is the polynomial basis representation. Each element of GF(2^m) can be expressed as a binary polynomial of degree smaller than m.

$$a(x) = \sum_{i=0}^{m-1} a_i \cdot x^i = a_{m-1} \cdot x^{m-1} + \cdots + a_1 \cdot x + a_0 \quad \text{with} \quad a_i \in \{0,1\} \quad (4.1)$$

A very convenient property of binary extension fields is that the addition of two elements is done with a simple bitwise XOR, which means that the addition hardware does not need to deal with carry propagation in contrast to a conventional adder for integers. The instruction set of virtually any general-purpose processor includes an instruction for the bitwise XOR operation.

When using a polynomial basis representation, the operations in GF(2^m) can be defined as polynomial addition and multiplication modulo an *irreducible polynomial* $p(x)$ of degree m. This reduction is only required in multiplication, as addition is carry-less and already yields a fully reduced element of GF(2^m). A *multiplication in GF(2^m)* can be performed in two steps[1]:

1. Multiplication of two binary polynomials of degree up to $m-1$, resulting in a product polynomial of degree up to $2m-2$[2].

2. Reduction of the product from the first step modulo the irreducible polynomial $p(x)$, yielding an element of GF(2^m) in fully reduced form.

First, we discuss the implementation options for the multiplication of binary polynomials (i.e., step one) only. Then, the reduction for the result of step one is outlined.

In software, the simplest way to implement the multiplication of two binary polynomials $a(x), b(x) \in$ GF(2^m) is by means of the so-called *shift-and-xor method* [219]. Several improvements of the classical shift-and-xor method have been proposed [113]; the most efficient of these is the *left-to-right comb method*

[1] These two steps can also be interleaved, depending on the characteristics of the implementation.

[2] Note that, unlike in integer multiplication, the coefficient for degree $2m-1$ is always zero due to the missing carry propagation.

by López and Dahab [162], which employs a look-up table to reduce the number of both shift and XOR operations.

A completely different way to realize the multiplication of binary polynomials in software is based on the MULGF2 operation as proposed by Koç and Acar [43]. The MULGF2 operation performs a word-level multiplication of binary polynomials, similar to the $(w \times w)$-bit MUL operation for integers, whereby w denotes the word size of the processor. More precisely, the MULGF2 operation takes two w-bit words as input, performs a multiplication treating the words as binary polynomials, and returns a $2w$-bit word as result[3]. All standard algorithms for multiple-precision arithmetic of integers can be applied to binary polynomials as well, using the MULGF2 operation as a subroutine [96]. Unfortunately, most general-purpose processors do not support the MULGF2 operation in hardware, although a dedicated instruction for this operation is simple to implement [181]. It was shown by Großschädl [97] that a conventional integer multiplier can be easily extended to support the MULGF2 operation, without significantly increasing the overall hardware cost. On the other hand, Koç and Acar [43] describe two efficient techniques to "emulate" the MULGF2 operation when it is not supported by the processor. For small word sizes (e.g., $w = 8$), the MULGF2 operation can be accomplished with help of look-up tables. The second approach is to emulate MULGF2 using shift and XOR operations (see [43] for further details).

In the following, we briefly describe an efficient word-level algorithm for multiple-precision multiplication of binary polynomials with the help of the MULGF2 operation. We write any binary polynomial $a(x) \in \mathrm{GF}(2^m)$ as a bit-string of its m coefficients, e.g., $a(x) = (a_{m-1}, \ldots, a_1, a_0)$. Then, we split the bit-string into $s = \lceil m/w \rceil$ words of w bits each, whereby w is the word size of the target processor. These words are denoted as $\tilde{a}_i$ (for $0 \leq i < s$), with $\tilde{a}_{s-1}$ and $\tilde{a}_0$ representing the most and least significant word of $a(x)$, respectively. In this way, we can conveniently store a binary polynomial $a(x)$ in an array of s single-precision words (unsigned integers), i.e., $a(x) = (\tilde{a}_{s-1}, \ldots, \tilde{a}_1, \tilde{a}_0)$. Based on the MULGF2 operation, a multiple-precision multiplication of binary polynomials can be performed according to Algorithm 1 (replicated from [96]). The tuple $(\tilde{u}, \tilde{v})$ represents a double-precision quantity of the form $u(x) \cdot x^w + v(x)$, i.e., a polynomial of degree $2w - 1$. The symbols $\otimes$ and $\oplus$ denote the MULGF2 and XOR operation, respectively. In summary, Algorithm 1 requires to carry out s^2 MULGF2 operations and $2s^2$ XOR operations in order to calculate the product of two s-word polynomials. We refer to [96] for a detailed discussion of this algorithm.

Once the product $a(x) \cdot b(x)$ has been formed, it must be reduced modulo the irreducible polynomial $p(x) = x^m + \sum_{i=0}^{m-1} p_i \cdot x^i$ to obtain the final result (i.e., a binary polynomial of degree up to $m - 1$ as a representative of a fully reduced element of $\mathrm{GF}(2^m)$). This reduction can be implemented very efficiently when $p(x)$ is a sparse polynomial, which means that $p(x)$ has few non-zero coefficients p_i. In such a case, the modular reduction requires only a few shift and XOR

[3] Note that the highest bit of the result is always zero.

Algorithm 1 Multiple-precision multiplication of binary polynomials [96].

Require: Two binary polynomials, $a(x) = (\tilde{a}_{s-1}, \ldots, \tilde{a}_1, \tilde{a}_0)$ and $b(x) = (\tilde{b}_{s-1}, \ldots, \tilde{b}_1, \tilde{b}_0)$, each represented by an array of s single-precision (i.e., w-bit) words.
Ensure: Product $r(x) = a(x) \cdot b(x) = (\tilde{r}_{2s-1}, \ldots, \tilde{r}_1, \tilde{r}_0)$ ($\tilde{r}_{2s-1}$ is always 0).
1: $(\tilde{u}, \tilde{v}) \leftarrow 0$
2: **for** i from 0 by 1 to $s-1$ **do**
3: **for** j from 0 by 1 to i **do**
4: $(\tilde{u}, \tilde{v}) \leftarrow (\tilde{u}, \tilde{v}) \oplus (\tilde{a}_j \otimes \tilde{b}_{i-j})$
5: **end for**
6: $\tilde{r}_i \leftarrow \tilde{v}$
7: $\tilde{v} \leftarrow \tilde{u}$, $\tilde{u} \leftarrow 0$
8: **end for**
9: **for** i from s by 1 to $2s-2$ **do**
10: **for** j from $i-s+1$ by 1 to $s-1$ **do**
11: $(\tilde{u}, \tilde{v}) \leftarrow (\tilde{u}, \tilde{v}) \oplus (\tilde{a}_j \otimes \tilde{b}_{i-j})$
12: **end for**
13: $\tilde{r}_i \leftarrow \tilde{v}$
14: $\tilde{v} \leftarrow \tilde{u}$, $\tilde{u} \leftarrow 0$
15: **end for**
16: $\tilde{r}_{2s-1} \leftarrow \tilde{v}$
17: **return** $r(x) = (\tilde{r}_{2s-1}, \ldots, \tilde{r}_1, \tilde{r}_0)$

operations and can be highly optimized for a given irreducible polynomial [112, 113, 219]. Most standards for ECC, such as from ANSI [4] and NIST [184], propose to use sparse irreducible polynomial like trinomials or pentanomials. On the other hand, an efficient word-level reduction method using the MULGF2 operation was introduced in the previously mentioned paper [96]. The word-level method also works with irreducible polynomials other than trinomials or pentanomials, but requires that all non-zero coefficients (except p_m) are located within the least significant word of $p(x)$, i.e., $p_i = 0$ for $w \leq i < m$. We used the trinomial $p(x) = x^{191} + x^9 + 1$ for our ECC implementations, which satisfies this condition for a word size of $w = 32$.

4.4 Proposed Multiply-Step Instructions

Our enhancement is basically the addition of one or two custom instructions which can be realized with relatively little additional hardware. The basic idea is to provide a multiply-step instruction for multiplication of binary polynomials. With a given word size w of the processor, a multiplication of two w-bit binary polynomials yielding a $2w$-bit result can be implemented efficiently with the proposed instructions. This word-size multiplication of binary polynomials corresponds to the MULGF2 operation mentioned in Section 4.3, which is an

important building block for arithmetic operations in the field GF(2^m).

The SPARC V8 Architecture Manual [225] defines a multiply-step instruction for integer multiplication (`mulscc`) and our proposed instructions are a modification thereof. The instruction `mulscc` processes one bit of the multiplier and adds the resulting partial product to a 64-bit accumulator realized by two hardware registers. In the following descriptions, the register naming conventions of SPARC V8 will be used.

In order to perform a complete multiplication of two 32-bit binary polynomials with a series of multiply-step instructions, three registers have to be employed. However, each instruction can only have two source operands. We follow the SPARC V8 solution for `mulscc` by using the special register `%y` implicitly in our multiply-step instructions. The other two registers can then be specified as the instruction's source operands. In our code examples, we use the registers `%o0` and `%o1`. Two of the registers form a logical 64-bit accumulator that holds the intermediate product during multiplication. The low word or the accumulator is fixed to be the register `%y`. The high word is contained in `%o0` in our examples. The *multiplier* is loaded into the low word of the accumulator (i.e., `%y`) at the beginning of the multiplication. The *multiplicand* resides in `%o1` during the whole course of the multiplication.

We now describe our two proposed multiply-step instructions for binary polynomials. First, we give a detailed description of the functionality of a single invocation of the respective instruction. We denote the involved operands in a generic fashion with `rs1` and `rs2` for the two source registers and `rd` for the destination register. Then we describe how the multiply-step instruction can be used to perform a complete multiplication with sample code. In this code, the aforementioned registers `%o0` and `%o1` will be used.

4.4.1 Single-Bit Variant

The first proposed instruction is named `mulgfs` and is only a slight variation of the `mulscc` instruction. It performs a step of a binary polynomial multiplication by processing a single bit of the multiplier. The instruction format is:

```
mulgfs rs1, rs2, rd
```

Apart from the two source registers (`rs1` and `rs2`) and the destination register (`rd`), the instruction implicitly works on the special register `%y`. The registers `rs1` and `%y` logically form a 64-bit accumulator register, where the first register contains the 32 most significant bits while the second registers contains the 32 lower bits[4].

The `mulgfs` instruction assumes that the accumulator contains the *intermediate product* in the higher part and the *unprocessed bits of the multiplier* in the lower part[5]. A `mulgfs` instruction performs the following steps:

[4] On other architectures, different approaches may be favorable, e.g., on a MIPS architecture the multiplication registers `%hi` and `%lo` could be implicitly used as accumulator.

[5] Note that while the multiplication progresses, the intermediate product gets longer while the unprocessed multiplier gets shorter.

1. The last bit of `%y` (i.e., the least significant unprocessed bit of the multiplier) is examined. A partial product (denoted as A) is set to the value of `rs2` (i.e., the multiplicand), if the bit is one. Otherwise A is set to all zeros. This is the contribution to the high word of the new intermediate product due to this bit of the multiplier.

2. The logical 64-bit accumulator (consisting of `rs1` and `%y`) is shifted one bit to the right. As a consequence, the bit of the multiplier processed in the previous step is shifted out.

3. The partial product A is XORed (i.e., added) to the high word of the accumulator and the result is stored in `rd`. The accumulator is now formed by `rd` and `%y` and contains the updated intermediate product and the remaining bits of the multiplier.

A possible implementation of the `mulgfs` instructions with dedicated XOR logic can be seen in Figure 4.1. The input value of the accumulator (`rs1` and `%y`) is shifted right by one and `rs2` is conditionally added to the high word. The 64-bit result is written to the accumulator (consisting of `rd` and `%y`).

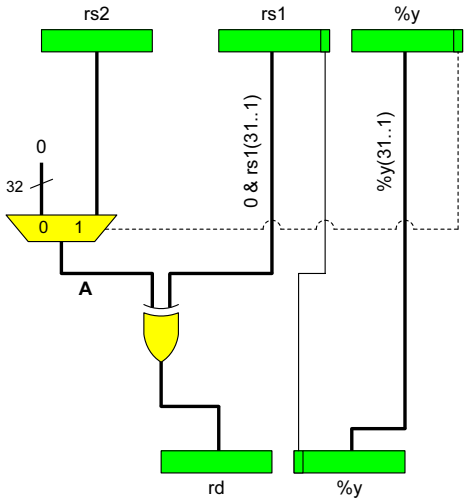

Figure 4.1: Implementation of the `mulgfs` instruction for the SPARC V8 architecture.

By using the `mulgfs` instruction repeatedly, the MULGF2 operation can be implemented efficiently. The instruction operands are mainly fixed during a single multiplication. The first source register (`rs1`) and the destination register (`rd`) are kept identical as long the 64-bit accumulator is read and written[6]. The second source register (`rs2`) has to contain the multiplicand. The multiplier

[6] As the `rd` register can be specified individually, the high word of the accumulator could be chosen differently after each `mulgfs` instruction. However, the common practice is to keep the high word of the accumulator in the same registers by using the same register for `rs1` and `rd`.

must be loaded into the low word of the accumulator (i.e., `%y`) at the beginning of multiplication. Figure 4.2 shows how the MULGF2 operation can be implemented with the `mulgfs` instruction.

```
! pre: Multiplier in %o0, multiplicand in %o1
! post: Product in %o0,%o1

mov %o0, %y   ! Load multiplier into %y (low word of accu)

! 32 iterations of mulgfs
! remark: %g0 is always zero
mulgfs %g0, %o1, %o0    ! High word of accu starts at zero
mulgfs %o0, %o1, %o0
! 30 x mulgfs %o0, %o1, %o0
! [omitted for compactness]

! Last iteration (33) only shifts result right by one
mulgfs %o0, %g0, %o0

! Result resides in %o0,%y
! Put least significant word of result into %o1
mov %y, %o1
```

Figure 4.2: Using `mulgfs` to implement multiplication of 32-bit binary polynomials (MULGF2 operation).

At the beginning, the multiplier is loaded into `%y`. Then, the `mulgfs` instruction is executed 32 times to process each bit of the multiplier in `%y`. In each execution of the `mulgfs` instruction, the value in the accumulator (`%o0` and `%y`) is shifted right by one. After 32 `mulgfs` instructions, the value in the accumulator must be shifted right by one to obtain the correct 64-bit result.

4.4.2 Double-Bit Variant

The second proposed instruction, named `mulgfs2`, is a variation of the `mulgfs` instruction, and it processes two bits of the multiplier simultaneously. Its format is:

```
mulgfs2 rs1, rs2, rd
```

The `mulgfs2` instruction executes the following steps:

1. The last bit of `%y` (i.e., the least significant unprocessed bit of the multiplier) is examined. If the bit is one, a partial product (denoted as B) is set to the value of `rs2` (i.e., the multiplicand) shifted right by one. Otherwise B is all zeros. This is the contribution to the high word of the new intermediate product due to this bit of the multiplier.

2. The second-lowest bit of `%y` (i.e., the penultimate bit of the multiplier) is examined. If it is one, a second partial product (denoted as A) is set to the value of `rs2` (i.e., the multiplicand). Otherwise A is all zeros. This is the contribution to the high word of the new intermediate product due to this bit of the multiplier.

3. The logical 64-bit accumulator (consisting of `rs1` and `%y`) is shifted two bits to the right. Consequently, the two processed bits of the multiplier are shifted out.

4. The partial products A and B are XORed to the high word of the accumulator and the result is stored in `rd`. Furthermore, the least significant bit of `rs2` (i.e., the multiplicand) is added to the most significant bit of the low word of the accumulator if the multiplier bit processed in step 1 is one. The accumulator is now formed by `rd` and `%y` and contains the updated intermediate product and the remaining bits of the multiplier.

In this fashion two partial products can be generated and added to the intermediate product in the accumulator with a single instruction. A possible implementation of the `mulgfs2` instruction for a SPARC V8 general-purpose processor can be seen in Figure 4.3. The input value of the accumulator (`rs1` and `%y`) is shifted right by two and `rs2` is twice conditionally added to the high word. One of the two conditional additions (the one for the lowest multiplier bit) also involves the most significant bit of the low word of the new accumulator value. This can be seen in Figure 4.3 as the logical AND of the least significant bits of `rs2` and `%y` and the subsequent XOR with the penultimate bit of `rs1`. The 32-bit three-input XOR can be realized with the modified ripple-carry adder presented in Section 4.5.

A multiplication of two binary polynomials (MULGF2 operation) is done in the same way as described in the Section 4.4.1 with the exception that the 32 subsequent `mulgfs` instructions are replaced by 16 `mulgfs2` instructions. This is shown in Figure 4.4.

If the `mulgfs` instruction is available, then the final shift can be done with a single instruction. If only the `mulgfs2` instruction is available, the final shift of the 64-bit accumulator must be done with conventional bit-test and shift instructions on the two respective 32-bit registers. On a SPARC V8 processor this requires four instructions.

4.5 Possible Implementation with Modified Ripple-Carry Adder

The paper of Großschädl [97] presents the design of a so-called unified multiply-accumulate unit that supports the MULGF2 operation. The efficiency of that design is based on the integration of polynomial multiplication into the datapath of the integer multiplier. On the other hand, the datapath for our proposed multiply-step instructions can be integrated into the ALU adder and does not

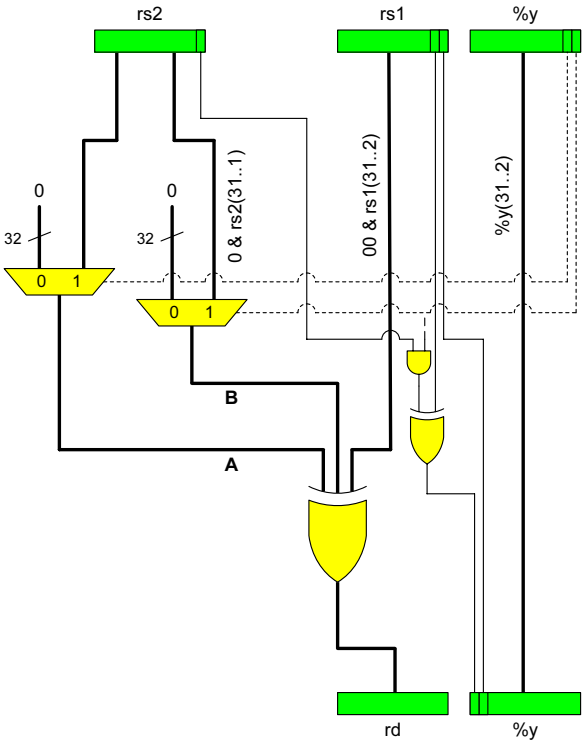

Figure 4.3: Implementation of the `mulgfs2` instruction for the SPARC V8 architecture.

require a multiplier. Alternatively, some dedicated logic (mainly consisting of XOR gates) could be added to the ALU of the processor.

For SPARC V8 cores, the implementation of our extension is relatively easy, as those cores already feature a multiply-step instruction for integer arithmetic. In comparison to the work in [97], the multiply-step instructions offer a tradeoff of hardware cost against speed.

The simplest way to implement adders in general-purpose processors is in the form of ripple-carry adders (RCAs). For instance the SPARC V8 LEON2 processor, which we have used for our evaluation, employs such an adder. Principally, ripple-carry adders consist of a chain of full adder cells, where each cell takes three input bits (usually labeled a, b and c_{in}) and produces two output bits with different significance (sum and c_{out}). The cells are connected via their carry signals, with the c_{out} of one stage serving as c_{in} input for the next higher stage.

A conventional ripple-carry adder takes two n-bit values and a carry-in bit and produces an n-bit sum and a carry-out bit which can be seen as the $(n+1)$-th bit of the sum. To generate a bit of the sum vector, each full adder cell performs a logical XOR of its three inputs a, b and c_{in}. This property can be

```
! pre: Multiplier in %o0, multiplicand in %o1
! post: Product in %o0,%o1

mov %o0, %y   ! Load multiplier into %y

! 16 iterations of mulgfs2
! remark: %g0 is always zero
mulgfs2 %g0, %o1, %o0   ! High word of accu is zero
mulgfs2 %o0, %o1, %o0
! 14 x mulgfs2 %o0, %o1, %o0
! [omitted for compactness]

! mulgfs shifts result right by one
mulgfs %o0, %g0, %o0

! Result resides in %o0,%y
! Put least significant word of result into %o1
mov %y, %o1
```

Figure 4.4: Using `mulgfs2` to implement multiplication of 32-bit binary polynomials (MULGF2 operation).

exploited to perform a bitwise logical XOR of three n-bit vectors with a slightly modified ripple-carry adder. As explained in Section 4.3, this XOR conforms to an addition of the three vectors if they are interpreted as binary polynomials.

The modification consists of the insertion of multiplexers into the carry-chain of the ripple-carry adder as illustrated in Figure 4.5. The *insert* control signal selects the carry value which is used. If *insert* is 0, the adder propagates the carry signal, selecting cp_i as c_{i+1}. In this mode the adder performs a conventional integer addition, setting s and c_{out} accordingly. If *insert* is 1, the carry is not propagated, but the *cins* vector is used to provide the c_i inputs for the full adder cells. The *sum* vector s is calculated as the bitwise logical XOR of the vectors a, b and *cins*. The value of c_{out} is not relevant in this mode and is forced to 0[7]. In Figure 4.5, the bits with the same significance of the three vectors are grouped together with braces. The carry input of the rightmost full adder cell receives c_{in} for integer addition and the least significant bit of the *cins* vector for addition of binary polynomials. The *insert* signal of the modified adder therefore switches the circuit between the functionality of an integer adder and a 3:1 compressor for binary polynomials.

Ripple-carry adders have the disadvantage that the delay of carry propagation can be rather high. If the carry propagation path of the adder constitutes the critical path, then a faster adder design can be used to achieve better timing

[7]This suppression can be omitted for a simple RCA design. However, cascading (e.g., for a carry-select adder) is simplified.

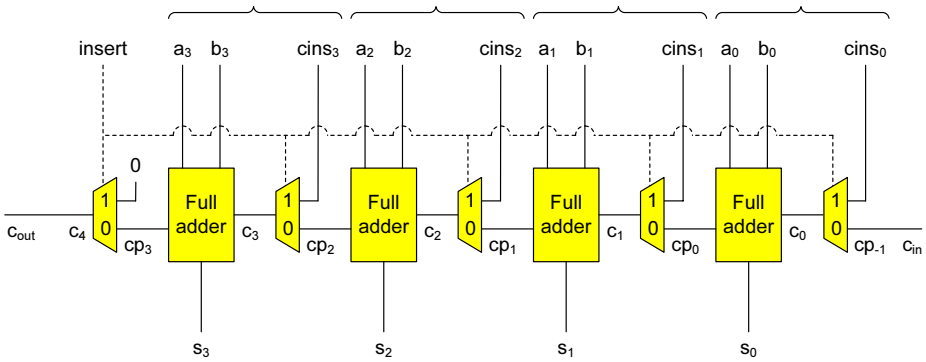

Figure 4.5: A 4-bit modified ripple-carry adder.

results for the whole design.

A good source for high-speed adder designs is the book of Chandrakasan [46]. A carry-select adder, for instance, can be adapted to accommodate addition of binary polynomials. A carry-select adder consists of several short adders which add different chunks of the input vectors. For each chunk there are two adders: One for each of the two possible values of the incoming carry from the next lower chunk. Once this carry arrives, it selects the correct sum and the next carry via multiplexers. If ripple-carry adders are used for the addition of the chunks, then the modifications for polynomial addition can be integrated into the carry-select adder by modifying one of the two adders for each chunk according to Figure 4.5[8]. Such a modified carry-select adder is shown in Figure 4.6.

4.6 Experimental Results

Both `mulgfs` and `mulgfs2` have been implemented in the SPARC V8-compliant LEON2 processor [80]. The size for both instruction and data cache have been set to 4 KB. A cycle counter register, whose content is incremented each clock cycle, has also been added to the LEON2 to facilitate the measurement of the execution time of software routines. An XSV-800 Virtex FPGA prototyping board [263] has been used to implement the extended processor for verification of the design and for obtaining timing result for different realizations of ECC operations.

The ECC parameters given in Appendix J.2.1 of the ANSI standard X9.62 [4] have been used. The elliptic curve is defined over the binary finite field GF(2^{191}) with the reduction polynomial $p(x) = x^{191} + x^9 + 1$. Most of the examined implementation variants use a multiplication of two binary polynomials (MULGF2 operation) as a building block for GF(2^m) operations where the size of the binary polynomials equals the word-size w of the LEON2 processor, namely 32

[8] In our case, we have modified the adder with carry input 0, but it would also be possible to modify the adder with carry input 1.

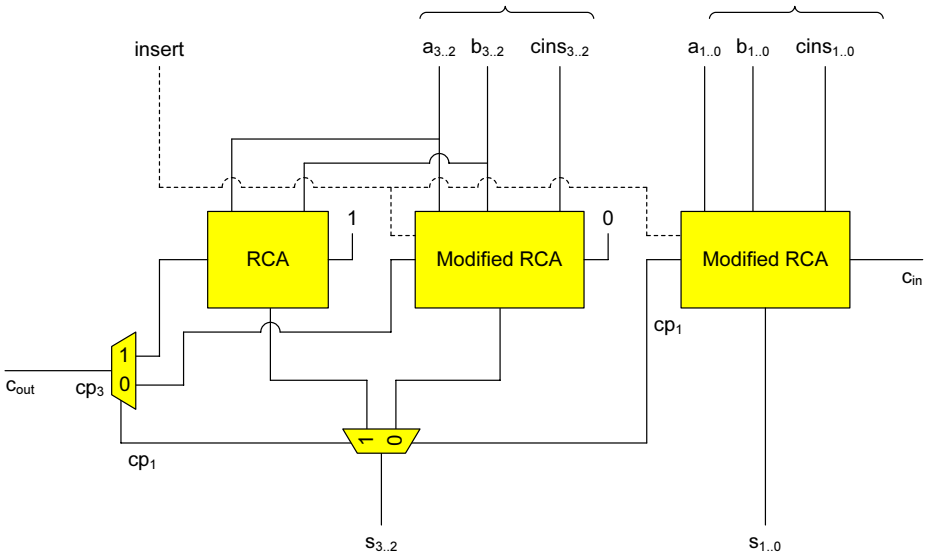

Figure 4.6: A 4-bit modified carry-select adder using modified ripple-carry adders as building blocks.

bits.

Two principal implementations of ECC operations have been employed for evaluation of the merits of the proposed multiply-step instructions. The first uses the left-to-right comb method with a look-up table containing 16 entries, as mentioned in Section 4.3, for polynomial multiplication and shift and XOR instructions for reduction. This implementation is tailored to the use of GF(2^{191}) with the above reduction polynomial and therefore especially suited for constrained devices. The different variants used for evaluation are denoted with the prefix OPT. The second implementation can work in a binary extension field of arbitrary length with any reduction polynomial which fulfills the requirement that it has only non-zero coefficients for powers smaller than w. Such an implementation is favorable for machines which need to handle elliptic curves over various finite fields. These variants are based on the MULGF2 operation as a building block for all GF(2^m) multiplication, squaring and reduction operations. They vary only in the implementation of the MULGF2 operation and are denoted with the prefix FLEX. All OPT and FLEX implementations use the method described by López and Dahab to perform an elliptic curve point multiplication [161].

4.6.1 Hardware Cost

We estimated the cost in terms of hardware for our proposed instructions at the example of the SPARC V8-compatible LEON2 processor. To this end, we implemented the different instructions in the processor and synthesized the integer

unit (IU), i.e. the 5-stage processor pipeline, of each processor version using a UMC 0.13 µm standard-cell library. The `mulgfs` instruction has been realized with a simple modification of the decode logic, while the heart of the `mulgfs2` instruction is realized with dedicated XOR gates. This additional functional unit for the `mulgfs2` instruction basically consists of 32 3-input XORs and a 32-bit 2-input multiplexer. Its cost can be estimated to about 200 gate equivalents (GEs). Our synthesis results are given in Table 4.1.

Table 4.1: Area and delay of LEON2-CIS's IU with various extensions.

	Area			**Delay**
Variant	μm^2	Gate equiv.	Norm.	ns
IU without extensions	67,887.94	13,106	1.00	5.00
IU with `mulgfs`	68,212.80	13,169	1.00	5.00
IU with `mulgfs2`	66,833.86	12,903	0.98	5.00
IU with `mulgfs` & `mulgfs2`	66878.78	12,919	0.99	5.00

For all combinations of extensions, the cost of additional hardware is negligible. The IUs which include the `mulgfs2` instruction is even smaller than the reference version without extensions[9].

4.6.2 Performance

Table 4.2 presents the performance (in clock cycles) of multiplication and squaring in GF(2^{191}) and of a complete elliptic scalar multiplication for the three variants of the flexible implementation. The first column (FLEX1) gives the results for the pure software variant, where the MULGF2 operation has been implemented with shift and XOR instructions. The second and third column list the performance for adapted versions, where the word-size polynomial multiplication (MULGF2 operation) has been optimized. FLEX2 refers to the variant which made use of the `mulgfs` instruction as described in Section 4.4.1. The results for FLEX3 are for an implementation which utilizes both `mulgfs` and `mulgfs2` instructions as outlined in Section 4.4.2. Both FLEX2 and FLEX3 necessitated only minor changes to the code of FLEX1.

Table 4.2: Execution times of important operations for ECC over GF(2^{191}) for the FLEX variants.

Operation	**FLEX1** cycles	**FLEX2** cycles	**FLEX3** cycles
GF(2^{191}) multiplication	15,344	2,306	1,620
GF(2^{191}) squaring	5,335	691	476
EC scalar multiplication	22,485,650	3,260,478	2,319,558

[9] The synthesis process does not allow to pinpoint the source of the area savings. We assume that our modifications of the decode logic to accommodate the new instructions allow for a more optimal synthesis result.

The performance of the EC scalar multiplication from Table 4.2 are a representative measure to compare the overall speed of the three implementations. The use of the `mulgfs` instruction alone (FLEX2) yields a speedup factor of nearly 7 over the pure software version. If both multiply-step instructions are available (FLEX3), the speedup factor is nearly 10.

The optimized implementation in pure software (OPT1) can be enhanced with the proposed multiply-step instructions. GF(2^{191}) multiplication which uses the `mulgfs` and `mulgfs2` instructions is faster than the multiplication of the original software implementation. Table 4.3 lists the performance of the three versions, where OPT2 uses just the `mulgfs` instruction and OPT3 makes use of both `mulgfs` and `mulgfs2` instructions to speed up GF(2^{191}) multiplication.

Table 4.3: Execution times of important operations for ECC over GF(2^{191}) for the OPT variants.

Operation	**OPT1** cycles	**OPT2** cycles	**OPT3** cycles
GF(2^{191}) multiplication	3,182	2,076	1,500
GF(2^{191}) squaring	273	273	273
EC scalar multiplication	3,909,690	2,706,560	2,054,282

Note that the performance for the GF(2^{191}) multiplication for OPT2 and OPT3 is better than that of FLEX2 and FLEX3 because the former use a reduction step which is tailored to the reduction polynomial $p(x) = x^{191} + x^9 + 1$.

EC scalar multiplication is sped up by a factor of about 1.45 with the `mulgfs` instruction and by about 1.9 through the use of both `mulgfs` and `mulgfs2` instructions. Additionally, FLEX2 is about 16% faster than OPT1 and FLEX3 is about 40% faster.

4.6.3 Memory Requirements

Table 4.4 compares the size of the code and data sections of a SPARC executable which implements the full EC scalar multiplication for the OPT and FLEX variants. The executables have been obtained by linking the object files for each implementation without linking standard library routines. The size of the code and data sections have subsequently been dumped with the GNU *objdump* tool. As the values for OPT2 and OPT3 and those for FLEX2 and FLEX3 are nearly identical, only one implementation of each group has been listed exemplarily.

The executables of the FLEX2 and FLEX3 implementations are only half the size of OPT1. This is mainly because OPT1 uses a hard-coded look-up table for squaring and also features larger subroutines. OPT2 and OPT3 have 40% smaller code sections and a 33% smaller executable compared to OPT1. In addition, OPT1 uses a look-up table for GF(2^{191}) multiplication which is calculated on-the-fly and requires additional space in the RAM. This memory requirement is eliminated in OPT2, OPT3 and all FLEX variants.

The OPT variants are the most likely candidates for usage in devices which

Table 4.4: Memory requirements of the OPT and FLEX variants of EC scalar multiplication.

Memory usage	**OPT1** bytes	**OPT2/3** bytes	**FLEX1** bytes	**FLEX2/3** bytes
Code section size	4,928	2,920	3,904	2,592
Data section size	1,024	1,024	264	264
Total size (executable)	5,952	3,944	4,168	2,856
Additional RAM	384	none	none	none

are constrained regarding their energy supply. To compare OPT1 with the enhanced versions OPT2 and OPT3, it is important to note that load and store instructions normally require more energy than other instructions on a common microprocessor; see e.g., the work of Sinha et al. [223]. Based on that fact it can be reasoned that OPT2 and OPT3 have a better energy efficiency than OPT1 for two reasons: They have better performance and do not use as many load and store instructions because they perform no table look-ups for field multiplication.

4.7 Summary and Conclusions

We presented an extension to general-purpose processors which speeds up ECC over GF(2^m). The use of multiply-step instructions accelerates multiplication of binary polynomials, i.e., the MULGF2 operation, which can be used to realize arithmetic operations in GF(2^m) in an efficient manner. We have integrated both proposed versions of the multiply-step instruction into a SPARC V8-compliant processor core. Two different ECC implementations have been accelerated through the use of our instructions. The implementation optimized for GF(2^{191}) and a fixed reduction polynomial has been sped up by a factor of about 1.9 while the size of its executable and its RAM usage have been reduced. The flexible implementation, which can cater for different field sizes m and an important set of reduction polynomials, was accelerated by an factor of over 10. Additionally, the enhanced flexible version could outperform the original optimized implementation by 40%. All enhancements required only minor changes to the software code of the ECC implementations.

The benefits for devices constrained in available die size and memory seem especially significant, as our multiply-step instructions require little additional hardware and reduce memory demand regarding both code size and RAM requirements at runtime. Additionally, the implementations which use our instructions are likely to be more energy efficient on common general-purpose processors.

5

Accelerating AES Using Instruction Set Extensions for Elliptic Curve Cryptography

Instruction-level support for finite field operations like the one presented in [101] and in Chapter 4 have been set in the context of public-key cryptography. As modern secret-key algorithms also use finite field operations as building blocks, the question appears whether such support can also be used in the secret-key setting. In this chapter we demonstrate that this is indeed the case by showing the use of word-level binary polynomial multiplication for acceleration of the Advanced Encryption Standard (AES) algorithm [185].

As already outlined in Section 2.5, a considerable fraction of the computation effort of a software implementation of AES is spent in the MixColumns and InvMixColumns transformations. Consequently, these transformations are a worthwhile target for optimization as demonstrated by the approaches of T-table lookup [57] or Bertoni et al.'s alternative representation of the AES State [25]. The main reason for the relatively poor performance of MixColumns implementations lies in the fact that the required multiplications in the binary extension field GF(2^8) are not supported by modern processors and need to be emulated by shift and XOR instructions.

Instruction set extensions for Elliptic Curve Cryptography (ECC) include support for arithmetic in large binary extension fields, e.g. [101]. In this section, we analyze how well these custom instructions are suited for accelerating a software implementation of AES on 32-bit platforms. Taking fast AES implementations for 32-bit processors as reference, we are able to obtain speedup factors of 1.34 for encryption and 1.2 for decryption.

5.1 Implementing AES on 32-bit Processors

On 32-bit platforms, most of the AES operations can be implemented with table lookups using the T-table approach [57]. A set of T-tables can be used to implement a specific part of the AES algorithm. For each such part, there is a choice between the use of a single table of 256 entries of 32-bit words or a set of four such tables, i.e., a size of 1 KB or 4 KB, respectively. The three additional tables in the set of four tables are just rotated versions of the original table. Therefore, a single T-table is sufficient if the necessary rotations are executed at runtime.

The part of AES most worthwhile to be implemented with T-tables is the combination of SubBytes, ShiftRows and MixColumns, which is used in normal encryption rounds. The SubBytes and ShiftRows transformations in the final round can also be implemented with another set of T-tables, but the potential speedup is rather small. Similarly, InvSubBytes, InvShiftRows and InvMixColumns can be realized with T-tables. However, in such a case it is necessary to employ the *equivalent inverse cipher structure*[1] (cf. Section 2.5). InvSubBytes and InvShiftRows in the final decryption round can also be done with T-tables. The use of the equivalent inverse cipher structure necessitates a more complex key expansion, as most of the round keys (except the first and last) must be transformed with InvMixColumns. When a precomputed key schedule is employed, the additional transformations normally pose no problem, as the costly key expansion is only done once per cipher key. However, if on-the-fly key expansion is to be used, AES decryption with T-tables for the rounds can become rather inefficient. The InvMixColumns operation in the key expansion can also be implemented with another set of T-tables.

The minimal size of lookup tables for a software AES implementation (without resorting to bit-slicing techniques) is 256 bytes for SubBytes and InvSubBytes, respectively. In principle, the byte substitutions could be calculated on-the-fly through their defining arithmetic operations: Inversion in $GF(2^8)$ and affine transformation. However, this would be very slow on conventional processors, which are not fit for arithmetic in binary extension fields. Therefore, lookup of the S-box remains the only practical solution.

The rest of the AES round transformations can be calculated with reasonable computational effort. SubBytes and InvSubBytes can be combined with ShiftRows and InvShiftRows, respectively, if the bytes are arranged accordingly after substitution. Such a combined operation is possible as SubBytes and ShiftRows are consecutive operations and their order of execution can be switched arbitrarily. The combination delivers the shifting of the rows at no additional cost. AddRoundKey can be realized with a few XOR instructions, which are found on virtually all microprocessors.

The bit-slicing technique can be applied in order to get AES implementations which do not require any lookup tables. Each AES State is distributed

[1] Otherwise, the costly operations of InvSubBytes and InvMixColumns would be separated by the AddRoundKey transformation and it would not be possible to perform them with T-table lookup.

amongst a number of registers where each register contains parts of a number of different States. The transformations themselves are expressed as logical operations. Multiple AES operations can be performed simultaneously and the average cost per block is comparable to a conventional implementation provided that the word size of the processor is sufficiently large [149, 167, 169]. The absence of data-dependent table lookups makes bit-slicing implementations resistant against cache-based timing attacks. However, the latency for a single AES operation is very large. This is especially a problem in situations where it is not possible to parallelize the AES operations, e.g., in CBC mode encryption. As bit-slicing leads to a less general solution for realizing AES, we will concentrate on conventional implementations.

Hence, depending on the implementation strategy, AES encryption requires between 256 bytes (just one S-box table) and 8 KB (two sets of T-tables of 4 KB each) of lookup tables. For AES decryption, the range goes from 256 bytes up to a maximum of 12 KB. Depending on the acceptable code size, the T-tables can be statically included in the code section of the program or generated at runtime. In the first case, the tables reside in the program memory of the processor while in the second case, they are placed in the working memory.

As will be shown in Chapter 6, the performance of AES implementations with T-tables is highly dependent on the properties of the memory subsystem of the processor. Especially on systems with slow memory and no or minimal cache, it can be faster to calculate the AES round transformations directly.

Another important design aspect is the storage of the State on 32-bit architectures. At the beginning of encryption or decryption, the State is filled with the plaintext or ciphertext. Herein, the first four bytes of the input make up the first column of the State, the next four bytes the second column, etc. On 32-bit processors, four bytes are usually packed into a 32-bit word in order to increase utilization of registers and the datapath. A common choice is to hold the four columns of the State in four 32-bit registers. We will denote an AES implementation with such a storage strategy as *column-oriented*. The well-known AES implementation of Brian Gladman [82] is an example of a column-oriented implementation.

The MixColumns and InvMixColumns operations interpret the State bytes and State columns as elements of binary extension fields and require operations which are normally not supported by common microprocessors. When these transformations are calculated by the processor, the finite field operations must be realized with instructions for logical operations, shifting and integer arithmetic. Consequently, a considerable part of AES is spent on calculating the MixColumns and InvMixColumns operations.

Bertoni et al. [25] have presented an alternate way for calculating MixColumns and its inverse on 32-bit platforms. Their strategy requires that the rows of the State are held in 32-bit words instead of the columns[2]. The key advantage of this method is the possibility to multiply all four bytes of each word simultaneously with the same constant from GF(2^8) without the need to shift the

[2] An alternative view is that the columns of the *transposed State matrix* are held in registers.

results into place. Although this strategy requires a transposition of the State matrix at the beginning and end of AES, a transposition of the cipher key and a more complex key expansion, the whole AES operation is commonly faster than a column-oriented implementation. The performance gains are especially significant for decryption, because InvMixColumns is much easier to calculate with the rows of the State than with the columns. The algorithms for calculating MixColumns and InvMixColumns using the State columns and State rows, as well as possible optimizations using ECC instruction set extensions will be discussed in the next section.

5.2 Optimizing AES Using Instruction Set Extensions

MixColumns and InvMixColumns require addition and multiplication of elements of the binary extension field GF(2^8) and of polynomials over GF(2^8). Addition in GF(2^8) is defined as a bitwise XOR. Multiplication in GF(2^8) can be seen as multiplication of binary polynomials (i.e., coefficients mod 2), followed by a reduction with an irreducible polynomial. Arithmetic with polynomials over GF(2^8) follows the conventional rules for polynomials, using addition and multiplication in GF(2^8) for the coefficients.

In the context of Elliptic Curve Cryptography, various instruction set extensions for arithmetic in binary extension fields GF(2^m) have been proposed. The word-level multiplication of binary polynomials has been identified as one of the key operations by Koç et al. in [43], where this operation was denoted as MULGF2. In [101], a small set of instructions (including one for MULGF2) for the MIPS32 architecture has been presented and their impact on ECC implementations over GF(p) and GF(2^m) has been evaluated. We have used three of these instructions to speed up AES implementations. Table 5.1 lists the instruction names used for MIPS32 in [101] and the mnemonics we have used for our SPARC implementation along with a short functional description. We will employ the SPARC names in the following.

All three instructions work on a dedicated accumulator whose size must be at least twice the word size, i.e., in our case at least 64 bits.

Table 5.1: The ECC instruction set extensions used to speed up AES.

Instruction name		
SPARC	MIPS32 [101]	**Description**
`gf2mul`	`mulgf2`	Multiply two binary polynomials
`gf2mac`	`maddgf2`	Same as `gf2mul` with addition of result to accumulator
`shacr`	`sha`	Shift lower word out of accumulator

The instructions `gf2mul` and `gf2mac` interpret the two operands as binary polynomials, multiply them, and put the result in the accumulator. They differ in that `gf2mul` overwrites the previous accumulator value while `gf2mac` adds

the polynomial product to it. The `shacr` instruction writes the lowest word of the accumulator to a given destination register and shifts the accumulator value to the right by a distance of 32 bits. All timing estimations for code snippets presented in this chapter are based on the following properties of the SPARC V8 architecture:

- No rotate instruction is available in the architecture. Rotation is done with two shifts and an OR/XOR instruction.
- In order to set a constant value with more than 13 bits in a register, two instructions are required.
- There are enough free registers to hold up to three constant words throughout the calculation of MixColumns or InvMixColumns.

5.2.1 Column-Oriented Implementation

For MixColumns and InvMixColumns, each new column can be calculated separately from the old column. This property is used if the four columns of the State are held in separate 32-bit words. The following code calculates MixColumns for a single State column in a conventional fashion. At the beginning, the input column is held in the variable `column` and at the end, the transformed column is written into this variable.

```
01  word double, triple;
02  double = GFDOUBLE(column);
03  triple = double ^ column;
04  column = double ^ ROTL(triple, 8) ^
             ROTL(column, 16) ^ ROTL(column, 24);
```

Figure 5.1: MixColumns for a single state column (conventional).

The operator ^ denotes bitwise XOR. The function GFDOUBLE interprets the four bytes of `column` as four elements of GF(2^8) and doubles them individually. The function ROTL rotates the word to the left by the given number of bits. The basic idea behind the code is that each byte of the resulting column consists of a weighted sum of the four bytes of the old column. Multiplication of all four bytes with the GF(2^8) constants 02 and 03 is done in line 2 and 3 and the result is stored in `double` and `triple`, respectively. In line 4, the bytes are rotated into the correct positions and summed up.

As will be shown in detail in Section 5.2.2, the function GFDOUBLE requires about 10 instructions. The function ROTL takes between one and three instructions. The actual number depends on whether the processor features a dedicated rotate instruction. As this is not the case for the SPARC V8 architecture, we will consider the cost of ROTL to be three instructions in the following. Logical

operations like the XORs in line 4 are considered to map to a single instruction. The calculation of a single column requires one GFDOUBLE, three ROTL and four XOR operations, which results in a total instruction count of 23 for the code in Figure 5.1.

When the ECC instruction set extensions listed in Table 5.1 are available, it is possible to calculate a column much faster. In order to do this, we use the definition of MixColumns in terms of a polynomial multiplication [57]. More precisely, MixColumns can be described as a multiplication of two polynomials of degree 3 with coefficients in GF(2^8). The input column is interpreted as the first polynomial, whereas the second polynomial is fixed to the value of $03 \cdot t^3 + 01 \cdot t^2 + 01 \cdot t + 02$. The following code calculates MixColumns for a single column.

```
01  word mask, low_word, high_word;
02  mask = column & 0x80808080;     // Extract MSBs
03  mask = mask >> 7;
04  GF2MUL(column, 0x01010302);     // Polynomial mult.
05  GF2MAC(mask, 0x00011a1b);       // Coefficient reduction
06  SHACR(low_word);                // Degrees 0-3 of poly.
07  SHACR(high_word);               // Degrees 4-6 of poly.
08  column = low_word ^ high_word;  // Polynomial reduction
```

Figure 5.2: MixColumns for a single state column (using extensions).

If the instruction set extensions are available, the three functions GF2MUL, GF2MAC, and SHACR directly map down to the corresponding processor instructions. The rest of the code consists of simple logical operations.

The main idea behind this code is illustrated in Figure 5.3. There are three phases in the whole calculation:

- Polynomial multiplication
- Reduction of polynomial coefficients
- Polynomial reduction

Line 4 performs the multiplication of the input column with the constant polynomial $01 \cdot t^3 + 01 \cdot t^2 + 03 \cdot t^1 + 02$. Note that as the bytes of the column represent the polynomial coefficients with ascending degree (i.e., the byte at the word's most significant position is the coefficient of t^0) [57], the coefficients of the constant polynomial have been rearranged accordingly to yield a product with ascending coefficient degree. The polynomial multiplication puts the first block of coefficients in Figure 5.3 in the accumulator. Note that the coefficients are displayed separately, but that they are in fact all contained as additive contributions in the multiplication result.

The coefficients si, which have been multiplied with the value 03 or 02, may no longer be in the reduced form of binary polynomials of degree smaller or equal to 7, but can be of larger degree. More specifically, $3\,si$ or $2\,si$ are no longer in reduced form if (and only if) the most significant bit (MSB) of si is 1. Such values are called residue values and they can be reduced to a degree smaller or equal to 7 by adding the reduction polynomial of the finite field, which is $x^8 + x^4 + x^3 + x + 1$ (0x11b) in the case of AES.

In order to achieve the reduction of the GF(2^8) coefficients, we add a reduction value ri for each coefficient $3si$ and $2si$. In the case that the MSB of si is 1, the according ri is set to 0x11b, otherwise ri is set to 0. The reduction values are shown in the middle of Figure 5.3 with corresponding coefficients and reduction values marked in the same color. After the addition of the reduction values to the result of the polynomial multiplication, all coefficients are fully reduced. The reduction values ri can be calculated by extraction of the corresponding MSBs of the coefficients si (lines 2 and 3) and the multiplication of these values with the constant 0x00011a1b (line 5). This constant is the sum of 0x11b aligned to the two lower bytes of the word, and the multiplication generates the reduction values ri in the required positions. In line 5, the reduction values are also added to the previous multiplication result in the accumulator by the GF2MAC operation.

In line 6 and 7, the polynomial is read into two variables and in line 8 the reduction of the polynomial is performed. Due to the special nature of the reduction polynomial $p(t) = t^4 + 1$, the coefficients for degrees 4 to 6 must be added to the coefficients of degree 0 to 2, respectively (the coefficient for degree 3 stays unchanged). This is easily done by an XOR of the low and the high word of the accumulator.

The calculation of a single column for MixColumns shown in Figure 5.1 therefore requires thirteen instructions. This includes the generation of the three constant values which are used in the process (0x80808080, 0x01010302, and 0x00011a1b). But as these values only need to be generated once per MixColumns, there is an average number of 8.5 instructions required to calculate a single column[3].

The optimizations for InvMixColumns work in a similar fashion, with the exception that the reduction values ri are generated with an additional GF2MUL operation by performing the polynomial multiplication with the highest three bits of each coefficient alone. The according code is shown in Figure 5.4. It takes approximately 16 instructions to calculate one column, which is much faster than the conventional approach.

Lines 2 to 10 generate the reduction bits and put them at the correct position in a single word. In line 12, these reduction bits are used to perform the coefficient reduction on the result of line 11. The rest of the code is similar to the one for MixColumns.

[3]The cost is 13 instructions for the first column (including constant generation) and $(13 - 6) = 7$ instructions for the remaining three columns. This gives a total of $13 + 3 \cdot 7 = 34$ instructions with an average of $34/4 = 8.5$ instruction per column.

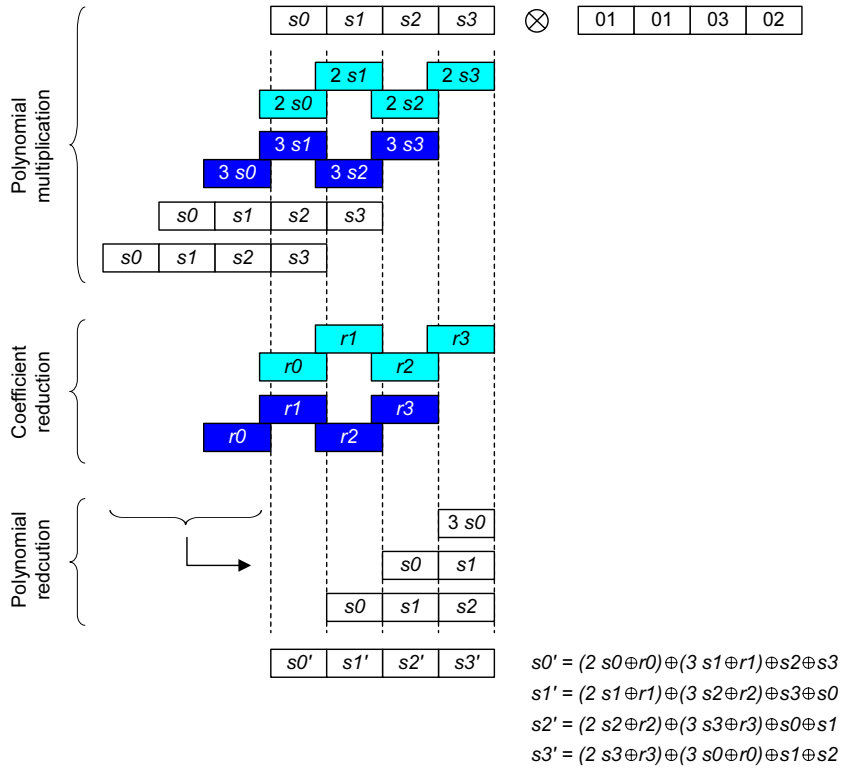

Figure 5.3: Polynomial multiplication and reduction to yield a column after MixColumns.

5.2.2 Row-Oriented Implementation

In a row-oriented AES implementation [25] the MixColumns and InvMixColumns operations are calculated together for the complete State. The strength of this method lies in the possibility to reuse intermediate results for all four columns of the State. This advantage is especially significant in the relatively complex InvMixColumns operation. The conventional row-oriented MixColumns uses four GFDOUBLE operations, while InvMixColumns requires seven. Figure 5.5 depicts the code for a conventional implementation of the GFDOUBLE operation.

Here, the reduction information is extracted from `poly` in lines 2 to 4. The actual doubling of the four bytes takes place in line 5. Afterwards, the reduction is performed (line 6). This version of GFDOUBLE requires 10 instructions, but the reuse of the bitmasks in consecutive GFDOUBLE operations leads to a lower instruction count. There are four consecutive doublings in both MixColumns and InvMixColumns, which can be done in an average of 7 instructions each when

```
01  word mask, low_word, high_word;
02  mask = column & 0xE0E0E0E0;       // Get highest 3 bits
03  GF2MUL(mask, 0x090D0B0E);         // Mask poly. mult.
04  SHACR(low_word);
05  SHACR(high_word);
06  low_word = low_word ^ high_word;  // Mask poly. red.
07  high_word = high_word << 24;      // Shift mask
08  low_word = low_word >> 8;         // Shift mask
09  mask = low_word ^ high_word;      // Assemble mask
10  mask = mask & 0x07070707;         // Select red. bits
11  GF2MAC(column, 0x090D0B0E);       // Polynomial mult.
12  GF2MAC(mask, 0x11b);              // Coefficient red.
13  SHACR(low_word);                  // Degrees 0-3 of poly.
14  SHACR(high_word);                 // Degrees 4-6 of poly.
15  column = low_word ^ high_word;    // Polynomial reduction
```

Figure 5.4: InvMixColumns for a single state column (using extensions).

```
01  word mask;
02  mask = poly & 0x80808080;
03  mask = mask >> 7;
04  mask = mask * 0x1b                // reduction mask
05  poly = (poly & 0x7f7f7f7f) << 1;
06  poly = poly ^ mask;
```

Figure 5.5: GFDOUBLE for row-wise MixColumns/InvMixColumns (conventional).

the bitmasks are reused[4].

With the help of the `gf2mul`, `gf2mac` and `shacr` instructions, the GFDOUBLE operation can be done slightly faster than with conventional instructions. Such an optimized version of GFDOUBLE is shown in Figure 5.6.

The reduction information is extracted in a similar manner, but doubling and reduction are done with GF2MUL and GF2MAC, respectively. The optimized version takes about 7 instructions. When reusing the bitmask in four consecutive doublings, the average instruction count goes down to approximately 5.5[5].

[4]With 10 instruction for the first GFDOUBLE and $(10 - 4) = 6$ instructions for the remaining three GFDOUBLE, the total cost is $10 + 3 \cdot 6 = 28$ instructions. The average cost per GFDOUBLE is hence $28/4 = 7$ instructions.

[5]Using 7 instructions for the first GFDOUBLE and $7 - 2 = 5$ instructions for the remaining three GFDOUBLE, the total cost is $7 + 3 \cdot 5 = 22$ instructions and the average cost per GFDOUBLE is $22/4 = 5.5$ instructions.

```
01  word mask;
02  mask = poly & 0x80808080;
03  mask = mask >> 7;    // reduction mask
04  GF2MUL(poly, 0x2);
05  GF2MAC(mask, 0x11b); // GF(2^8) coefficient reduction
06  SHACR(poly);
```

Figure 5.6: GFDOUBLE for row-wise MixColumns/InvMixColumns (using extensions).

5.3 Practical Results

In order to implement and evaluate our new AES implementation approaches, we used a version of the LEON2-CIS [134] which includes the required ECC extensions. The processor configuration includes a (32×32)-bit unified integer/polynomial multiply-accumulate unit with a 72-bit accumulator (including 8 guard bits for integer multiply-accumulate). At the heart of this multiply-accumulate unit is a (32×16)-bit unified integer/polynomial multiplier. The processor has been implemented on the GR-PCI-XC2V FPGA board [196].

On this version of the LEON2-CIS, a `gf2mul` instruction executes in three cycles, while a `gf2mac` instruction takes only a single cycle[6]. The `shacr` instruction always finishes in one cycle.

In order to estimate the hardware cost of the extensions, we have compared the synthesis results of a "conventional" LEON2 featuring a conventional (32×16)-bit integer multiplier to our LEON2-CIS variant. The latter requires about 5,5 kGates more than the reference version, whereby the added functionality encompasses not only all the extensions from [101], but also a signed multiply-accumulate instruction. Unfortunately, the LEON2 does not offer a configuration with a (32×16)-bit multiply-accumulate unit, so the sole cost of the instructions for binary polynomials cannot be determined easily.

We have made tests with Gladman's AES code [82] using it both as reference as well as an instance of a column-oriented implementation. However, Gladman's code only allows to optimize the calculation of a single column and not of the whole MixColumns transformation. Therefore, we have implemented our own version of a column-oriented AES which is more easily optimized. Furthermore, we have implemented our own version of a row-oriented AES following the ideas from [25]. Our column-oriented and row-oriented versions are written in C and support both encryption and decryption both for a precomputed key schedule as well as for on-the-fly key expansion. Moreover, all versions feature a conventional implementation with native SPARC V8 instructions and an optimized implementation where MixColumns and InvMixColumns make use of the

[6] As `gf2mac` instructions write their result exclusively to the dedicated accumulator registers, it is possible that other instructions are executed in parallel, provided that they do not depend on the result of `gf2mac`. This behavior is similar to the one of MIPS32 processors [101].

ECC instruction set extensions.

Timing measurements have been done using the integrated cycle counter of the LEON2-CIS. The code which performs the measurements has been derived from Gladman's code. In order to get a fair comparison of the different implementation options, we have used a processor configuration with a very large instruction and data cache (4 sets with 16 KB each, organized in lines of 8 words)[7]. The results therefore reflect performance in an environment with fast memory access or with "perfect" cache.

5.3.1 Precomputed Key Schedule

Table 5.2: Execution times of AES-128 encryption, decryption and key expansion.

Implementation	**Key expansion** cycles	**Encryption** cycles	**Decryption** cycles
Gladman NOTABLES	522	1,860	3,125
Column-oriented	497	1,672	2,962
Row-oriented	738	**1,636**	**1,954**
Gladman NOTABLES optimized	522	1,755	1,906
Column-oriented optimized	497	**1,257**	1,576
Row-oriented optimized	738	1,502	**1,567**
Speedup		23.1%	19.8%

Table 5.2 lists the timing results for AES-128 encryption and decryption when using a precomputed key schedule. The time for doing the key expansion is also stated. The speedup is calculated between the best conventional implementation and the best optimized implementation (best performance marked in bold). The row-oriented AES is best for both conventional encryption and decryption. For the optimized variants, the column-oriented implementation is best for encryption, while the performance for decryption is nearly identical for the column-oriented and row-oriented version.

5.3.2 On-the-fly Key Expansion

The timing results in Table 5.3 refer to AES-128 encryption and decryption with on-the-fly key expansion. As Gladman's code does not support this mode, only the results for our column-oriented and row-oriented version are stated. Note that the last round key is supplied to the decryption routine, so that it does not have to perform the whole key expansion at the beginning of decryption.

For conventional encryption, the column-oriented AES is slightly better, while for decryption, the row-oriented version is fastest. For the versions which use the ECC extensions, the column-oriented AES is better for both encryption

[7] This was done in order to prevent disturbing cache effects with one-way associative caches, where the performance of different implementations depends on the actual memory addresses of their stack variables.

Table 5.3: Execution times of AES-128 encryption and decryption with on-the-fly key expansion.

Implementation	**Encryption** cycles	**Decryption** cycles
Column-oriented	**2,254**	3,357
Row-oriented	2,328	**2,433**
Column-oriented optimized	**1,674**	**2,018**
Row-oriented optimized	2,230	2,176
Speedup	25.7%	17.0%

and decryption. The speedup is again calculated considering the best conventional and optimized version.

5.3.3 Code Size and Side-Channel Attacks

The code size for the implementations ranges between 2.5 KB and 3.5 KB, where the optimized variants are always smaller than the non-optimized ones. Note however, that the implementations have been optimized for speed and not for code size. Savings through optimization go up to 15% (for column-wise decryption with precomputed key schedule).

The susceptibility to side-channel attacks is not changed through the use of the instruction set extensions. It is therefore necessary to integrate countermeasures into a system which calculates AES using the presented methods, if resistance against side-channel attacks is required.

5.4 Summary and Conclusions

In this chapter we have demonstrated the use of instruction set extensions originally designed for elliptic curve cryptography for the acceleration of software implementations of AES. Although not specifically designed for that purpose, the use of the three instructions `gf2mul`, `gf2mac` and `shacr` allows performance gains of up to 25%. This speedup can be considered as "free" on processors which already feature these instructions. Generally, the column-oriented AES implementations can be optimized very well with the instruction set extensions.

6

Analysis of the Impact of Instruction Set Extensions on Memory Efficiency

This chapter investigates the interplay of memory efficiency and performance in the implementation of AES. We demonstrate that there is no single fastest implementation option and we show the importance of taking the cache architecture of the processor into account. A small custom instruction for the AES S-box lookup is introduced which is fit to increase the speed of encryption by a factor of over 1.4 and to reduce code size by 30–40%.

6.1 Memory Considerations for AES Implementations

As already mentioned in Section 5.1, AES can be implemented with large lookup tables (T-tables) in order to perform most of the round transformations. Such *T-lookup implementations* are usually considered to yield the highest performance. Nevertheless, T-lookup implementations can suffer from a number of drawbacks. When memory resources are at a premium, the use of large tables is not desirable. Moreover, the performance of a lookup table-based implementation is highly dependent on memory and cache performance. In Section 6.3 we demonstrate that, for small cache sizes, the performance of AES with large lookup tables can be much worse than that of an implementation which calculates most round transformations (We will denote such implementations as *calculated AES implementations*). Another problem of large lookup tables is an increased factor of cache pollution. This means that each execution of AES will throw out a large number of cache lines from other tasks. If these tasks continue

they will have to fetch their data from main memory again, thus leading to a degradation in overall performance. Another issue for AES decryption is the necessity to use a much more complex key expansion if T-lookup is employed. More specifically, the key expansion requires the transformation of nearly all round keys with InvMixColumns, which is a very costly operation.

For calculated AES implementations there are a number of design options on 32-bit processors. The 16 bytes of the State can be stored in four 32-bit registers, where they can either hold the columns or rows of the State matrix. Another option is to either precompute and store all round keys (precomputed key schedule) or to calculate the round keys during AES encryption or decryption (on-the-fly key expansion). The first option occupies more memory and may also require more memory accesses while the second option saves memory at the cost of additional operations in encryption and decryption in order to calculate the round keys.

We propose a custom instruction for performing the non-linear byte substitution of SubBytes and InvSubBytes in a small dedicated hardware unit (S-box unit). This way, we can completely eliminate the requirement of memory-resident lookup tables. Note that lookup tables can also be eliminated without additional hardware by using bit-slicing techniques [149, 167, 169], but reasonable performance can only be reached when multiple blocks can be processed at the same time. In contrast to bit-slicing techniques, our solution retains the full flexibility of the AES implementation, so that peak performance can be reached irrespective of the concrete application scenario.

The implementation details of the new `sbox` instruction are described in Section 6.2. With this instruction it is possible to implement AES with very few memory accesses. If there are enough spare registers to store the State and round key, then the only memory accesses required are the loading of the input data and cipher key and the storing of the result[1]. Popular RISC architectures for embedded systems like ARM, MIPS and SPARC offer large enough register files to allow such implementations.

By eliminating the need for lookup tables, all possible threats through cache-based side-channel attacks are also removed [23, 28, 147, 194, 248]. Cache pollution is kept to a minimum and the performance of AES becomes much more independent of the cache size as shown in Section 6.3. Another advantage of our proposed extension is the reduction of energy dissipation. Memory accesses are normally the most energy-intensive instructions [223], and hence their minimization can lead to a substantial energy saving.

6.2 Custom Instruction for S-box Lookup

In order to implement the AES S-box lookup, a number of different hardware designs have been proposed in the literature. An analysis of the implementation characteristics of the most important approaches will be given in Chapter 10.

[1] Provided that an implementation with on-the-fly key expansion is used.

For our implementation we have chosen the implementation of Wolkerstorfer et al. [259]. It can perform S-box lookup for both encryption and decryption, is relatively small, and can be easily implemented with any standard cell library. We wanted to achieve a high degree of flexibility, and therefore we have designed the new instruction such that it can be used for both column-oriented and row-oriented implementations. In SPARC notation, the `sbox` instruction has the following format:

```
sbox rs1, imm, rd
```

The `sbox` instruction operates on a single byte of the source register `rs1` and replaces one byte from the destination register `rd` with the substituted byte. The immediate value `imm` provides additional configuration information for the performed operation. More specifically, `imm` determines the source and destination bytes from `rs1` and `rd`, respectively. Furthermore, the direction of the substitution (S-box or inverse S-box) is also determined by the value of `imm`. More formally, the `sbox` instruction performs the following steps:

1. Select one of the four bytes in the source register (`rs1`), depending on the immediate value (`imm`).
2. Depending on `imm`, perform substitution with the S-box (for encryption and key expansion) or with the inverse S-box (for decryption).
3. Replace one of the four bytes in the destination register (`rd`) with the substituted value from step 2, as indicated by `imm`. The other three bytes in `rd` remain unchanged.

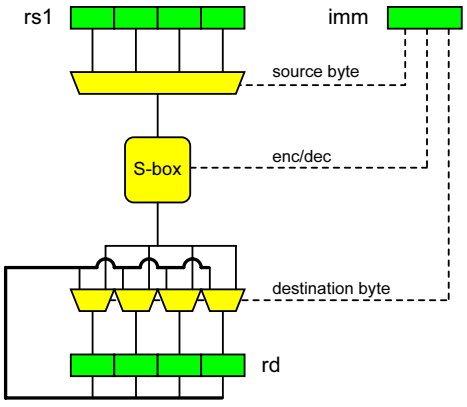

Figure 6.1: Functionality of the `sbox` instruction.

Figure 6.1 illustrates the operation of the `sbox` instruction. It requires the values from both of the registers `rs1` and `rd` to produce the 32-bit result: One byte is taken from `rs1` and transformed with the S-box and three bytes are taken

from `rd`. The access to `rd` can be easily realized as the second operand of the `sbox` instruction is always an immediate value. As a consequence, the second read port of the register file is not occupied and can be used to read in the value of the destination register `rd`. The `sbox` instruction is therefore easy to integrate into most architectures for embedded processors like ARM, MIPS, and SPARC as they all have instruction formats with two source registers.

Our instruction supports both encryption and decryption and can be used to perform all byte substitutions in all AES rounds as well as in the key expansion. It is possible to select the source byte in `rs1` and the destination byte in `rd` in a manner so that the SubBytes and ShiftRows transformation can be done simultaneously. The same applies for the InvSubBytes and InvShiftRows operations in decryption.

Figure 6.2 shows how SubBytes and ShiftRows can be realized with 16 `sbox` instructions.

```
! First column
sbox C0, 0x100, R0
sbox C1, 0x111, R0
sbox C2, 0x122, R0
sbox C3, 0x133, R0

! Second column
sbox C1, 0x100, R1
sbox C2, 0x111, R1
sbox C3, 0x122, R1
sbox C0, 0x133, R1

! Third column
sbox C2, 0x100, R2
sbox C3, 0x111, R2
sbox C0, 0x122, R2
sbox C1, 0x133, R2

! Fourth column
sbox C3, 0x100, R3
sbox C0, 0x111, R3
sbox C1, 0x122, R3
sbox C2, 0x133, R3
```

Figure 6.2: SubBytes and ShiftRows realized with the `sbox` instruction.

The four columns of the input State are contained in the registers `C0` to `C3`, while the transformed State columns are stored in the registers `R0` to `R3`. The second operand of each instruction is a three digit hexadecimal value. The most significant digit indicates the direction of the substitution and is always set to 1 for the forward S-box. The middle digit indicates the byte of the source register

`Cx` to be transformed, while the last digit indicates the byte which is replaced in the destination register `Rx` (0 for the most significant byte up to 3 for the least significant byte).

Similarly, the SubWord and RotWord operations in the key expansion can be implemented with four `sbox` instructions as given in Figure 6.3.

```
! OUT = SubWord(RotWord(IN))
sbox IN, 0x103, OUT
sbox IN, 0x110, OUT
sbox IN, 0x121, OUT
sbox IN, 0x132, OUT
```

Figure 6.3: SubWord and RotWord using the `sbox` instruction.

6.3 Influence of Cache Size on Performance

In order to demonstrate that an AES implementation with large lookup tables does not necessarily deliver the best performance, we have compared implementations with different sizes of lookup tables on an extended LEON2 with different cache sizes. The influence of cache size on the performance of AES has already been studied by Bertoni et al. [24]. Their work assumes that the cache is large enough to hold all lookup tables. In this section we will examine the situation where the cache may become too small to hold the complete tables.

In our experiments, we have varied the size of the data and instruction cache from 1 KB to 16 KB (both caches always had the same size). The implementations which use T-lookup are based on the well-known and referenced AES code from Brian Gladman [82], whereby we have used a total size of 1 KB, 4 KB, and 8 KB for the lookup tables, respectively. We have compared the achieved performance to two AES implementations which calculate all round transformations except SubBytes. In one case, a 256-byte lookup table (only S-box lookup) is used, and in the other case, our `sbox` instruction is employed. Figure 6.4 shows the performance for encryption, while Figure 6.5 depicts the results for decryption.

The performance of the lookup implementations is very bad for small cache sizes. For encryption, the usage of the `sbox` instruction yields a similar performance as the use of big lookup tables on a processor with very large cache. In decryption, T-lookup implementations become faster at cache sizes of more than 4 KB. This is due to the fact, that the InvMixColumns transformation is rather complex to calculate and therefore T-lookup becomes more efficient than calculation for large cache sizes. The main result of our experiments is that the performance of implementations using the `sbox` instruction is almost independent from the cache size. On the other hand, the performance of T-lookup implementations depends heavily on the size of the cache.

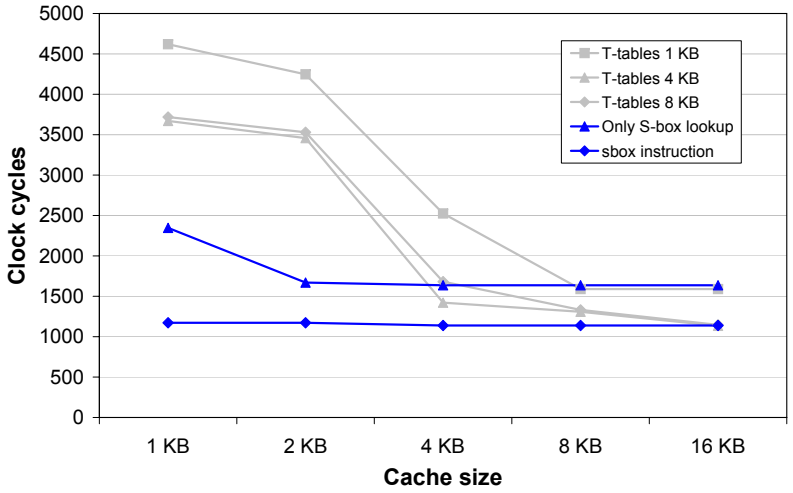

Figure 6.4: Performance of AES-128 encryption in relation to cache size.

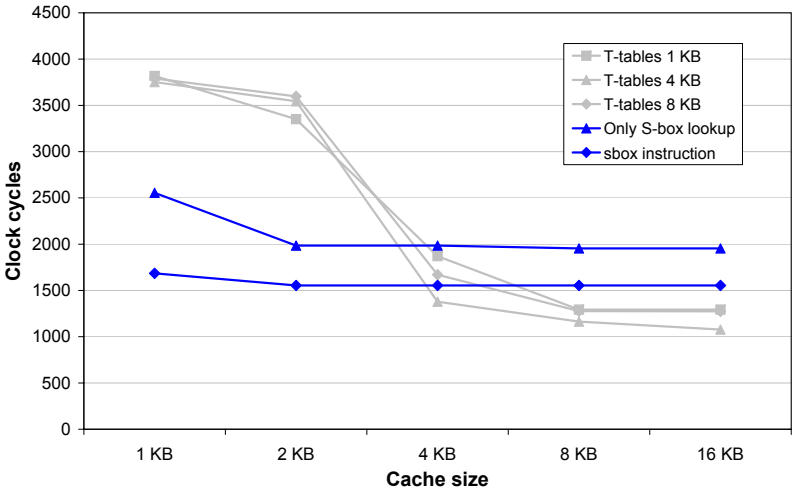

Figure 6.5: Performance of AES-128 decryption in relation to cache size.

6.4 Comparison of Calculated AES Implementations

The previous section has shown that the performance of AES implementations using T-lookup varies greatly with cache size. In this section we aim to highlight the benefits of using the `sbox` instruction in settings where T-lookup is not an option, e.g., due to limited memory. To analyze the performance, we

have compared a calculated AES implementation (without extensions) to one that uses our proposed `sbox` instruction. We have determined both the gain in performance as well as the reduction in code size. All comparisons have been done for both precomputed key schedule and on-the-fly key expansion.

The `sbox` instruction performs the AES S-box lookup in hardware in a single clock cycle, while a conventional calculated implementation requires a number of instructions, which increases both the execution time and the size of the executable. In systems with small cache, the speedup factor for the implementation with `sbox` instruction will be higher than in systems with large cache, mainly because the performance of the calculated software implementation (without extensions) degrades due to cache misses in the instruction cache. Therefore, we have used a LEON2 system with large caches (16 KB each for data and instruction cache) since we are primarily interested in the speedup due to the `sbox` instruction (and not due to less cache misses).

We have also tested a third implementation that uses both the `sbox` instruction as well as the `gf2mul`/`gf2mac` instructions used for MixColumns implementation as described in Chapter 5. All three implementations have been written in C and inline assembly has only been used to execute the custom instructions. For on-the-fly key expansion, we have also tested an assembler-optimized implementation which uses both the `sbox` and `gf2mul`/`gf2mac` instructions. This variant makes optimal use of the large register file offered by the SPARC V8 architecture and performs only a minimal number of memory accesses (8 loads for plaintext and key, 4 stores for ciphertext), which cannot be reduced further.

6.4.1 Area Overhead Estimation

We have integrated our proposed extension into the LEON2 embedded processor and prototyped it in a Xilinx Virtex2 XC2V3000 FPGA. In Sections 6.4.2 and 6.4.3 we give the practical results we have achieved by comparing an AES implementation which uses our `sbox` instruction with pure-software implementations. Our implementations used a key size of 128 bits, but the results also apply to larger key sizes. We have prototyped the extended LEON2 on an FPGA board, where the timing results have been obtained with the help of a cycle counter which is integrated in the processor.

In order to estimate the area overhead due to our extensions, we have synthesized the functional unit presented in [259] using a 0.35 µm CMOS standard cell library. The required area amounted to approximately 400 NAND gates, which is negligible compared to the size of the processor. When synthesized for the Xilinx Virtex2 XC2V3000 FPGA, the extended LEON2 (with 1 KB instruction and 1 KB data cache) required 4,274 slices and 5 Block RAMs.

6.4.2 Performance

Table 6.1 contains the timings for AES encryption and decryption with a precomputed key schedule. The use of the `sbox` instruction yields a speedup of 1.43 for encryption and 1.25 for decryption, respectively. It can also be seen

that the key expansion is accelerated by the use of our proposed extension. Table 6.2 states the timing results for an on-the-fly key expansion. The figures for decryption assume that the last round key is directly supplied to the AES decryption function. The speedup for encryption and decryption is about 1.43 and 1.3, respectively.

If both the `sbox` and the `gf2mul` instruction are used, the speedup lies between 1.79 for AES decryption with precomputed key schedule and 3.68 for AES encryption with on-the-fly key expansion.

Table 6.1: Execution times of AES-128 encryption, decryption and key expansion.

	Key exp.	**Encryption**		**Decryption**	
Implementation	cycles	cycles	speedup	cycles	speedup
No custom instructions	738	1,636	1	1,954	1
`sbox` instruction	646	1,139	1.43	1,554	1.25
`sbox` & `gf2mul` instruction	345	807	2.02	1,087	1.79

Table 6.2: Execution times of AES-128 encryption and decryption with on-the-fly key expansion.

	Encryption		**Decryption**	
Implementation	cycles	speedup	cycles	speedup
No custom instructions	2,254	1	2,433	1
`sbox` instruction	1,576	1.43	1,866	1.3
`sbox` & `gf2mul` instruction	868	2.59	1,126	2.16
`sbox` & `gf2mul` instr. (optimized)	612	3.68	881	2.76

6.4.3 Code Size

Savings in code size are mainly due to the fact that the lookup tables for SubBytes and InvSubBytes can be omitted thanks to the `sbox` instruction. Another positive effect of this instruction is that the code for SubBytes and ShiftRows and their inverses becomes more compact. The figures for the implementation with a precomputed key schedule are stated in Table 6.3. They have been obtained with the *objdump* tool from the executable. The code size shrinks by 32% for encryption and by 36% for decryption. Table 6.4 specifies the code sizes for AES with on-the-fly key expansion. Savings in code size range from nearly 43% for decryption to more than 37% for encryption. With the `gf2mul` instruction, even more than 76% of the program memory can be saved.

6.5 Summary and Conclusions

In this chapter we have presented an inexpensive extension to 32-bit processors which improves the performance of AES implementations and leads to a

Table 6.3: Code size of AES-128 encryption and decryption with precomputed key schedule.

	Encryption		Decryption	
Implementation	bytes	reduction	bytes	reduction
No custom instructions	2,168	0%	2,520	0%
`sbox` instruction	1,464	32.4%	1,592	36.8%
`sbox` & `gf2mul` instr.	680	68.6%	792	68.5%

Table 6.4: Code size of AES-128 encryption and decryption with on-the-fly key expansion.

	Encryption		Decryption	
Implementation	bytes	reduction	bytes	reduction
No custom instructions	1,656	0%	2,504	0%
`sbox` instruction	944	42.9%	1,564	37.5%
`sbox` & `gf2mul` instruction	628	62.0%	764	69.4%
`sbox` & `gf2mul` instr. (optimized)	480	71.0%	596	76.1%

reduction in code size. With the use of the proposed `sbox` instruction, all data-dependent memory lookups can be removed and the overall number of memory accesses can be brought to an absolute minimum. This instruction has been designed with flexibility in mind and delivers compact AES implementations with good performance even if cache is small and memory is slow. In our practical work we have observed a speedup of up to 1.43 while code size has been reduced by over 40%. The relative performance gain is much higher on processors with small cache size.

Furthermore, the `sbox` instruction also makes an AES implementation immune to cache-based side-channel attacks since it eliminates the table lookups in the SubBytes operation. The extra hardware cost of the `sbox` instruction amounts to only 400 gates.

7

Instruction Set Extensions for AES on 32-Bit Architectures

Following the encouraging results for supporting AES described in Chapters 5 and 6, a performance-oriented investigation of possible instruction-set support seemed definitely promising. When we started our work, the topic of instruction set extensions for AES on general-purpose processors has been largely untouched (with the exception of the work of Nadehara et al. [180] and Ravi et al. [202]). There had been a number of dedicated cryptographic processors featuring support for AES (e.g., [262, 189]), but these architectures were not fit for general-purpose processing. Moreover they emphasized broad support of cryptographic algorithms over optimization towards specific ones.

Considering the available research into architectural AES support, we decided that dedicated support of AES on general-purpose processors definitely required a more thorough investigation. Our first goal was to demonstrate that instruction set extensions generally represent an interesting point in the design space of secret-key cryptographic implementations, offering a good balance of performance, flexibility and cost. Our second goal was to continue and improve the existing research work. Finally, we hoped that our results could serve as an starting point for designers wishing to integrate AES support into a general-purpose processor.

7.1 Related Work on Extensions for AES

This section gives details of related work on the support of AES in application-specific and general-purpose processors. A comparison of the respective perfor-

mance figures with those of our approach is given in Table 7.6 in Section 7.6.

Burke et al. have developed custom instructions for several AES candidates [38]. They have proposed a 16-bit modular multiplication, bit-permutation support, several rotate instructions, and an instruction to facilitate address generation for memory table lookups. In a follow-up work, Wu et al. have designed CryptoManiac, a cryptographic coprocessor. CryptoManiac is a *Very Long Instruction Word (VLIW)* processor able to execute up to four instruction per cycle [262]. Additionally, short latency instructions (e.g., bitwise logical and arithmetic instructions) can be combined to be executed in a single cycle. To support this feature, instructions have up to three source operands.

The Cryptonite crypto-processor is a VLIW architecture with two 64-bit datapaths [189]. It features support for AES through a set of special instructions for performing byte-permutation, rotation and xor operations. The main part of AES is done with help of parallel table lookup from dedicated memories.

Fiskiran and Lee have investigated the inclusion of hardware lookup tables as a measure to accelerate different symmetric ciphers including AES [75]. They propose inclusion of on-chip scratchpad memory to support parallel table lookup. Examined are datapath widths of 32, 64 and 128 bits with 4, 8 and 16 tables, respectively, whereby each table contains 256 entries of 32-bit words (i.e., each table has a size of 1 KB).

Extensions for PLX—a general-purpose RISC architecture—have been proposed by Irwin and Page [133]. In their work they also examined the usage of the multimedia extensions of a PLX processor with a 128-bit datapath in order to implement AES with a minimal number of memory accesses. However, the presented concepts can hardly be adapted to 32-bit architectures.

Automatic generation of instruction set extensions for cryptographic algorithms (including AES) has been investigated by Ravi et al. using the 32-bit Xtensa processor from Tensilica [202]. Nadehara et al. proposed a single custom instruction which calculates most of the AES round transformations for a single State byte [180]. Their approach maps the round lookup (T-lookup) into a dedicated functional unit. Bertoni et al. have proposed several instructions for AES and have published implementation details and estimated performance figures for an Intel StrongARM processor [26].

Schaumont et al. [213] and Hodjat et al. [119] have investigated the addition of an AES coprocessor to the 32-bit LEON2 embedded processor. Performance for a memory-mapped approach and a connection through a dedicated coprocessor interface (CPI) has been reported. An AES operation was one to two orders of magnitude slower in relation to the mere time required by the coprocessor.

7.2 General Description of our Extensions

We designed several custom instructions to increase the performance of AES software implementations. These instructions have been developed for 32-bit processors with a RISC-like instruction format with two input operands and one output operand. All important 32-bit RISC architectures, such as SPARC, MIPS

and ARM, adhere to this three-operand format. Our instructions do not require special architectural features like dedicated look-up tables or non-standard register files, which makes their integration into general-purpose RISC architectures relatively easy. An integration into extensible processors like Tensilica's Xtensa [230] should also be straightforward. Furthermore, all of our instructions have been designed with the goal to keep the critical path of a concrete hardware implementation as short as possible.

The custom instructions can be categorized as byte-oriented or word-oriented, depending on whether a single byte or four bytes of output are produced. All instructions calculate parts of AES round transformations, yielding either one or four transformed bytes as result. The targeted AES round transformations are SubBytes, ShiftRows, and MixColumns, as well as their respective inverses. Moreover, the custom instructions also support the SubWord-RotWord operation of the key expansion.

7.2.1 Byte-Oriented AES Extensions

The byte-oriented instructions have fixed types of source operands. The first source operand is a register, while the second source operand is always an immediate value. This immediate value is used to configure the operation of the instruction. The single-byte result is written to a byte of the destination register, while the other three bytes retain their previous value. As the second source operand is an immediate value, the second read port of the register file is not occupied and can hence be used to load the value of the destination register. In this way, the old value from the destination register can be combined with the single-byte result, producing the complete 32-bit result of the instruction. This result is written back to the register file.

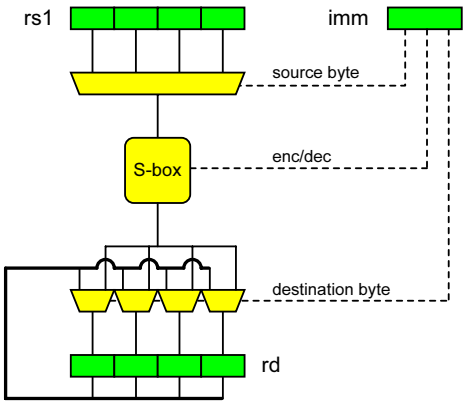

Figure 7.1: Functionality of the `sbox` instruction.

The `sbox` instruction has been proposed in Chapter 6 with the motivation to reduce the memory requirements of AES implementations. For convenience,

its functionality is again depicted in Figure 7.1. The format of the instruction is given as follows:

```
sbox rs1, imm, rd
```

The `sbox` instruction transforms one byte of the source register (`rs1`) with the AES S-box or inverse S-box and writes the resulting byte into the destination register (rd). The immediate value (`imm`) is used to select the source byte from the source register, the transformation (S-box or inverse S-box) and the destination byte. With this instruction, both the SubBytes and the ShiftRows transformation can be implemented very efficiently. The `sbox` instruction also accelerates the SubWord-RotWord operation in the AES key expansion.

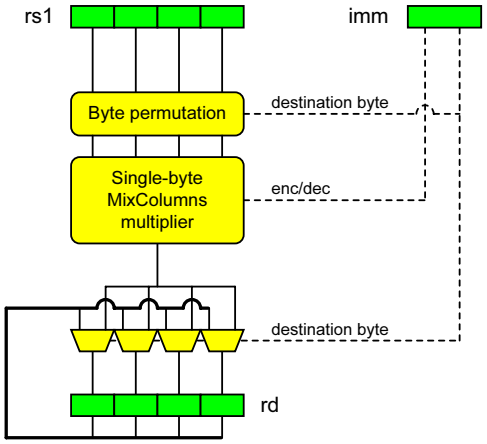

Figure 7.2: Functionality of the `mixcol` instruction.

The `mixcol` instruction is proposed as a complement to the `sbox` instruction. As the name implies, it performs a part of the MixColumns or InvMixColumns transformation. The instruction's format is:

```
mixcol rs1, imm, rd
```

Figure 7.2 shows the functionality of this instruction. The `mixcol` instruction uses the value in the source register (`rs1`) as input column and produces a single byte of the resulting column after the MixColumns operation. In this case, the immediate value sets the operation (MixColumns or InvMixColumns) and selects one of the four possible destination bytes. The destination byte determines a specific byte permutation as input to the single-byte MixColumns multiplier and the location of the resulting byte in the destination register. The complete resulting column can, therefore, be acquired with four executions of the `mixcol` instruction. As MixColumns and especially InvMixColumns are relatively costly in software, this instruction can lead to considerable speedups.

7.2.2 Plain Word-Oriented AES Extensions

The word-oriented instructions always produce a 32-bit result which is stored in the destination register. The most trivial approach for designing such word-oriented instructions is to simply quadruple the functionality of the byte-oriented extensions.

The `sbox4` instruction has the following format:

```
sbox4 rs1, imm, rd
```

This instruction substitutes all four bytes of the first source register and places them into the destination register. A byte-wise rotation can optionally be performed on the result. The immediate value selects whether S-box or inverse S-box are used for substitution and sets the rotation distance for the result. The optional rotation is useful for row-oriented AES implementations, where ShiftRows can be performed with no additional cost. Moreover, the SubWord-RotWord operation of the key expansion is supported by the `sbox4` instruction. The operation of `sbox4` is shown in Figure 7.3.

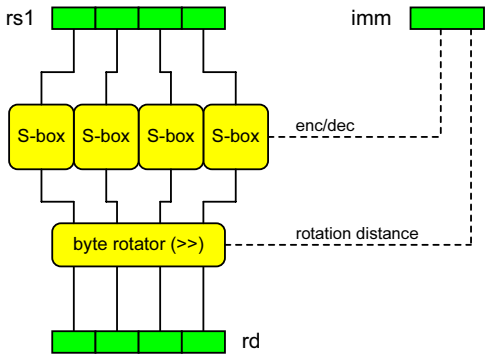

Figure 7.3: Functionality of the `sbox4` instruction.

The `mixcol4` instruction calculates all four result bytes of the MixColumns or InvMixColumns operations. The instruction format is as follows:

```
mixcol4 rs1, imm, rd
```

As illustrated in Figure 7.4, the input column is taken from the first source register while the immediate value as second operand just selects the operation (encryption or decryption).

However, as our performance evaluation in Section 7.4 shows, these simplistic instructions yield sub-optimal results. In Section 7.2.3 we analyze the problem and we introduce a slight modification of the extensions in Section 7.2.4. With these modifications, the extensions are able to deliver a very satisfactory support for AES.

Although the `sbox4` and `mixcol4` instructions cannot be combined optimally, they can be of use for trading performance for implementation cost. When used

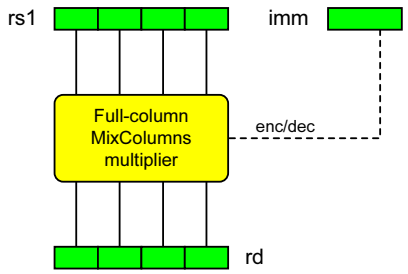

Figure 7.4: Functionality of the `mixcol4` instruction.

alone, `sbox4` is useful for reducing the number of memory accesses, while giving a moderate boost in performance. On the other side, `mixcol4` can be combined with the byte-oriented `sbox` instruction to yield a very good performance with minimal hardware overhead.

7.2.3 Analysis of ShiftRows as Bottleneck

The major drawback of the `sbox4` and `mixcol4` instructions is that they cannot be combined in a manner to allow an efficient AES implementation. The problem is the ShiftRows transformation, which becomes the performance bottleneck.

In a column-oriented implementation, SubBytes and MixColumns would be done with the respective custom instruction, while ShiftRows must be done separately. This problem is illustrated in Figure 7.5 where these transformations are shown during a normal round.

The grouping of four State bytes illustrates how the State is held in 32-bit words in each phase of the round. In this case, the State columns are packed into 32-bit words, which are held in general-purpose registers. The colors mark the required arrangement of State columns after the ShiftRows transformation. As MixColumns is the subsequent operation of ShiftRows, each `mixcol4` instruction requires as input a column of uniform color. However, as the `sbox4` instructions cannot change the layout of the State accordingly, ShiftRows must be performed explicitly in software. In a practical implementation, it turns out that this explicit ShiftRows requires a number of shift and logical operations, which require a total of about 44 instructions. Compared to the eight instructions required for SubBytes and MixColumns, this presents a dramatic overhead.

Another option for combining `sbox4` and `mixcol4` is to change the way the State is held in 32-bit words during the AES round. This solution is illustrated in Figure 7.6, where the change of representation for a single State column and row is highlighted for clarity.

SubBytes and ShiftRows can be performed simultaneously with the `sbox4` instructions, provided that the State rows are kept in the 32-bit registers. For MixColumns with `mixcol4` it is then necessary to hold the State columns in registers. Therefore, the State representation must be changed from rows to columns after the `sbox4` instructions, and back from columns to rows after the

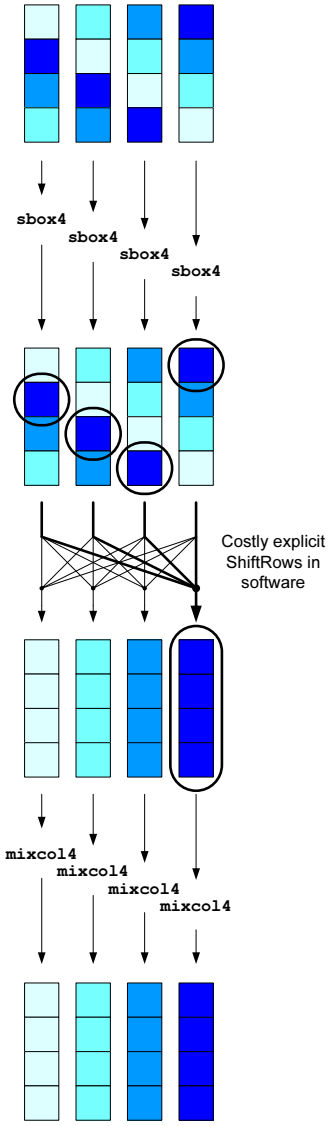

Figure 7.5: Combining `sbox4` and `mixcol4` through explicit ShiftRows.

`mixcol4` instructions. However, each such mapping would require similar effort as performing ShiftRows explicitly. Moreover, with two mappings required per round, this approach would be even more inefficient than the first one.

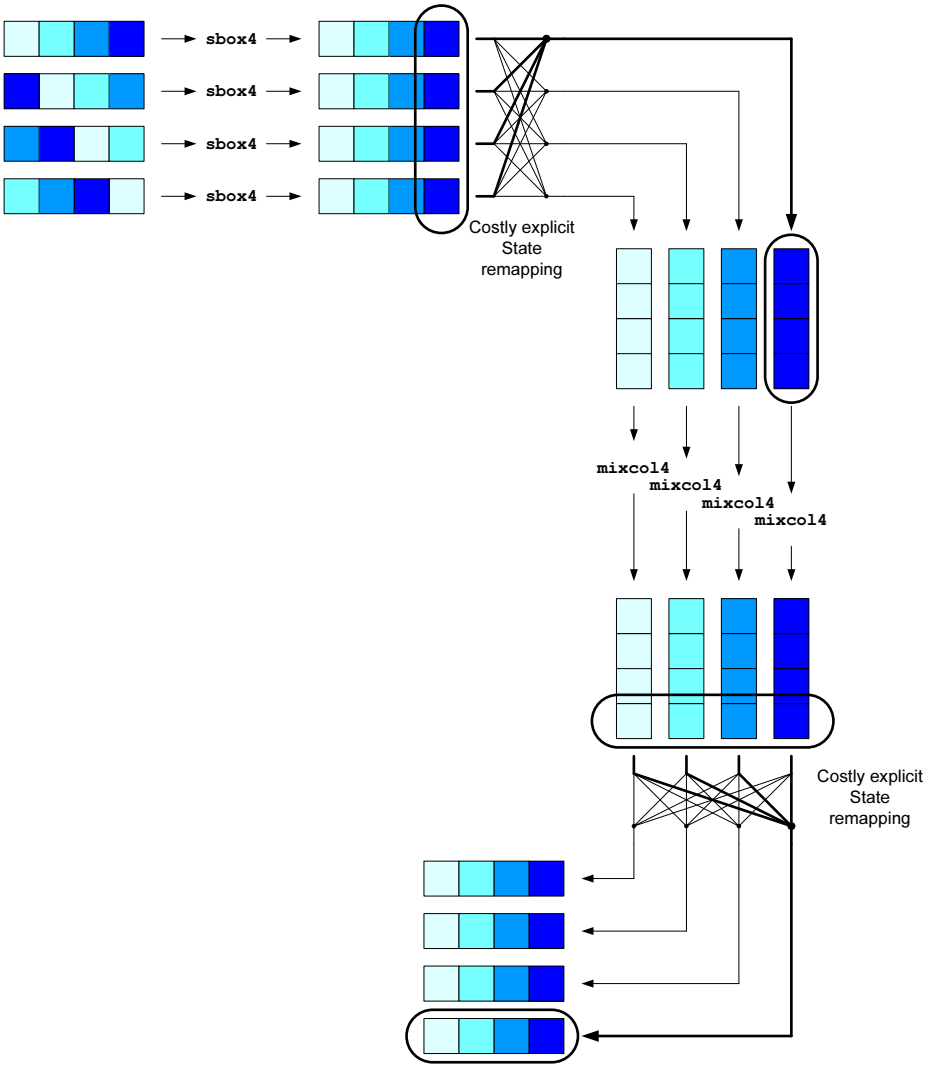

Figure 7.6: Combining `sbox4` and `mixcol4` through explicit State remapping.

7.2.4 Advanced Word-Oriented AES Extensions with Implicit ShiftRows

Fortunately, there is an elegant and efficient solution to the problem with ShiftRows. Assuming a column-oriented implementation, ShiftRows can be done implicitly with slightly modified `sbox4` and `mixcol4` instructions. In order to achieve this, the modified versions need to have two source register operands. From each source register, two bytes are extracted and assembled to a new intermediate State column. The respective AES transformation is performed on

this intermediate column and the result is stored in the destination register. By selecting the registers with the appropriate State columns as first and second source operands it is possible to perform the ShiftRows transformation implicitly. This selection is illustrated in Figure 7.7.

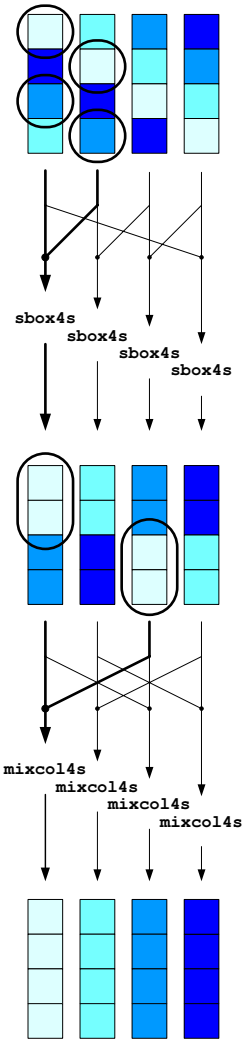

Figure 7.7: Performing ShiftRows implicitly with the `sbox4s` and `mixcol4s` instructions.

For the two new instructions, the selection of bytes from the two State columns is fixed, while the columns themselves can be chosen in software by supplying the appropriate registers as arguments. The fixed byte selection can

be used for both ShiftRows in encryption and InvShiftRows in decryption[1]. In the case of decryption, it is required to use the equivalent inverse cipher structure [185], where InvSubBytes, InvShiftRows, and InvMixColumns are subsequent transformations. The equivalent inverse cipher structure demands a more complicated key schedule, where most round keys have to be transformed with InvMixColumns. AES decryption with on-the-fly key expansion hence requires more time than encryption, but due to the fast implementation of InvMixColumns with the extensions, this overhead remains small.

As the second operand of the new instructions must now also be a register, there is an immediate value available to configure the specific operation of the instruction. Therefore, separate instructions must be used for S-box and inverse S-box transformation as well as for MixColumns and InvMixColumns. The modified instructions are denoted with an "`s`" appended to the original mnemonic (`sbox4s`, `mixcol4s`). To indicate the mnemonic for the respective inverse operation, an "`i`" is prepended (`isbox4s`, `imixcol4s`).

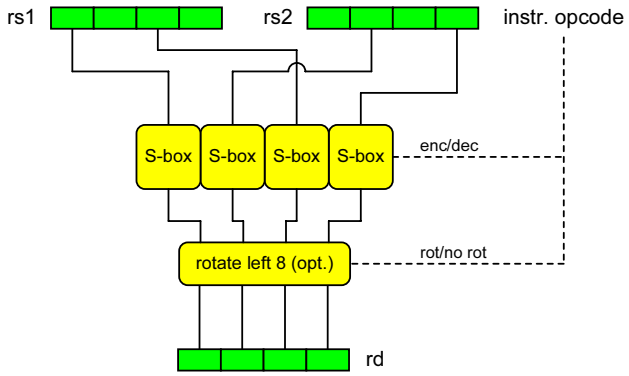

Figure 7.8: Functionality of the `sbox4s`, `isbox4s` and `sbox4r` instructions.

Figure 7.8 shows the functionality of the `sbox4s` and `isbox4s` instructions. These instructions have the format:

```
sbox4s rs1, rs2, rd
isbox4s rs1, rs2, rd
```

The first (i.e., most significant) and third byte of the first source register and the second and fourth (i.e., least significant) byte from the second source register are substituted using the AES S-box or inverse S-box. The optional rotation to the left by one byte is not used for these two instructions. The four S-boxes are used to realize a third instruction `sbox4r`, which performs S-box substitution followed by rotation to the left by 8 bits. This instruction implements the SubWord-RotWord operation of the AES key expansion. The format of `sbox4r`

[1]This scheme also supports most of the block lengths specified for Rijndael, except for 224-bit blocks

is different from the other instructions, as it has only a single source register. Its is as follows:

```
sbox4r rs1, rd
```

In reference to Figure 7.8, the `sbox4r` instruction is implemented by using the same register for both inputs. The byte rotation by a selectable distance of the `sbox4` instruction is not implemented as this functionality is not useful for column-oriented AES implementations.

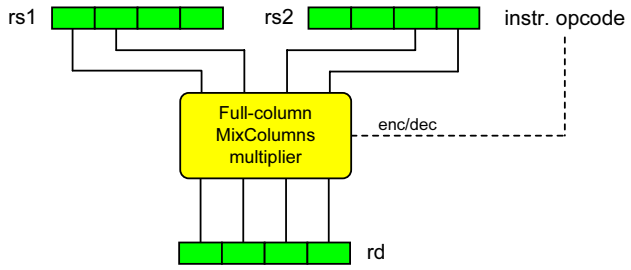

Figure 7.9: Functionality of the `mixcol4s` and `imixcol4s` instructions.

The two instructions `mixcol4s` and `imixcol4s` perform MixColumns and InvMixColumns, respectively. The instruction format is:

```
mixcol4s rs1, rs2, rd
imixcol4s rs1, rs2, rd
```

The functionality of these instructions is depicted in Figure 7.9. The input column to the MixColumns multiplier is assembled from the two most significant bytes of the first source register and the two least significant bytes of the second source registers. Note that an AES State column contained in a single register can still be transformed with a single instruction. This can be achieved by setting this source register as both first and second source operand.

7.3 Hardware Cost

We have integrated the proposed instructions into the LEON2 processor. In order to estimate the cost for the additional hardware, we synthesized the new functional units and the complete LEON2 integer unit (IU), i.e., the 5-stage processor pipeline, with the AES extensions using a UMC 0.13 µm standard-cell library. We used all viable combinations of custom instructions and have evaluated their performance in Section 7.4.

For the S-box extensions we have synthesized a single hardware S-box using two different approaches: The design of Canright, which calculates the S-box in hardware [41] and a hardware lookup table synthesized as an array of logic. The MixColumns multiplier follows the approach by Wolkerstorfer [258] and produces

a single byte of the resulting column. For synthesis of the integer unit we have chosen a target delay for the critical path of 4 ns, which conforms to a maximal clock frequency of 250 MHz. These synthesis results include the complete area overhead of the extensions, e.g., new functional units and decoding logic for the additional opcodes. The results are given in Table 7.1. Note that `sbox4s` indicates the three instructions `sbox4s`, `isbox4s` and `sbox4r` and that `mixcol4s` stands for the instructions `mixcol4s` and `imixcol4s`.

Table 7.1: Area and delay of the functional units for the proposed extensions and of the extended integer unit.

	Area			**Delay**
Functional unit/Component	μm^2	GE	norm.	ns
S-box (Canright) [41]	3,362.69	650	0.05	2.21
S-box (HW LUT)	15,709.25	3,033	0.23	0.64
MixColumns multiplier [258]	2,248.13	435	0.03	0.51
IU without extensions	69,144.19	13,349	1.00	3.93
IU with `sbox`	73,417.54	14,174	1.06	4.00
IU with `sbox4`	77,849.86	15,029	1.13	4.00
IU with `mixcol`	71,865.79	13,874	1.04	3.90
IU with `mixcol4`	72,372.10	13,972	1.05	3.98
IU with `sbox` & `mixcol`	71,753.47	13,853	1.04	4.00
IU with `sbox` & `mixcol4`	75,536.06	14,583	1.09	4.00
IU with `sbox4s` & `mixcol4s`	84,794.69	16,370	1.23	4.00

The S-box of Canright is about one fifth the size of the synthesized lookup table, but is also considerably slower. The MixColumns multiplier requires little area and has a shorter critical path than the S-boxes. The results in Table 7.1 for the integer unit use the approach of Canright [41] for the S-box extensions. Area overhead is calculated in relation to an integer unit without extensions and ranges between a factor of 1.04 and 1.23.

We used the minimal configuration (no hardware multiplier and divider, no FPU, no Ethernet MAC, no PCI controller, no SDRAM controller, no Debug Support Unit), where the IU accounts for less than half of the area of the LEON2 processor (excluding register file and cache memories). The size of the register file and caches is configurable and depends heavily on the particular RAM implementation. For the largest extensions (`sbox4s` & `mixcol4s`), the area overhead will therefore be at most half of the IU overhead (which is a factor of about 1.12), without taking register file or cache memory into consideration. In practice, these units will require a significant portion of the total area, so that the overall overhead factor for the area will be much lower.

7.4 Performance and Code Size

We have implemented AES using different combinations of the proposed custom instructions on the modified LEON2. In total, we examined seven different sets

of AES extensions. For comparison, the performance of AES implementations using T-lookup has also been determined on the same platform. Bitsliced implementations of AES are not expected to be faster than T-lookup [167] and have therefore not been considered in our evaluation. Both AES encryption and decryption with precomputed key schedule as well as with on-the-fly key expansion have been examined. A pure-software AES implementation has been used as baseline implementation. It uses no extensions and calculates all AES round transformations except SubBytes. For all implementations the number of clock cycles per block encryption/decryption and code size are given. Moreover, the speedup as well as relative change of code size in comparison to the baseline implementation are cited. For AES implementations with precomputed key schedule, the performance of the key expansion is also given.

The LEON2 has been implemented on a GR-PCI-XC2V FPGA board with a cache size of 16 KB for both instruction and data cache. The number of cycles has been obtained with the built-in cycle counter of the modified LEON2. For timing measurements we have used the code from Gladman's AES implementation [82][2]. The code size encompasses all functions and memory constants required to perform the respective AES operation. This includes the encryption/decryption function, the key expansion function (if required), and necessary lookup tables. The used custom instructions are indicated in the first column of each table. As before, `sbox4s` stands for `sbox4s`, `isbox4s` and `sbox4r`; `mixcol4s` stands for `mixcol4s` and `imixcol4s`.

When a set of extensions is useable for both column-oriented and row-oriented AES implementations, both of these options have been examined and the faster option has been included in the tables. Most AES implementations are written in C and use inline assembly to make use of the custom instructions. Implementations marked with *ASM* are completely written in assembly. For the implementation which uses the `sbox4s` and `mixcol4s` instructions, an assembly-optimized version with unrolled loops has also been tested (marked with *unrolled*). For each T-lookup implementation, the size of the tables is indicated. The first number indicates the table size for the round lookup, the second number (if present) is the table size for the last round. For AES decryption, the third number (if present) indicates the size of the table used for the key expansion function.

Table 7.2 summarizes the performance and code size for AES-128 encryption with a precomputed key schedule and Table 7.3 gives the respective figures for decryption. For the proposed extensions, speedups of up to 8.35 for encryption and 9.97 for decryption are achieved. With the fastest extensions, AES-128 encryption and decryption of a single block can be done in 196 clock cycles. The code size of these implementations is always reduced, whereby the savings are more significant for the MixColumns extensions than for the S-box extensions. The T-lookup implementations from Brian Gladman have been used for comparison [82]. There the speedup is up to 1.5 for encryption and 1.78 for decryption

[2]Gladman's code times the execution of 9 subsequent operations and of a single AES operation. The time for one operation is determined as the difference of these measurements divided by 8.

Table 7.2: AES-128 encryption, precomputed key schedule: Performance and code size.

	Key exp.	**Encr. perf.**		**Code size**	
Implementation	cycles	cycles	speedup	bytes	rel. change
No extensions (pure SW)	739	1,637	1.00	2,168	0.0%
`sbox`	647	1,140	1.44	1,464	- 32.5%
`sbox4` (C)	739	1,020	1.60	1,656	- 23.6%
`sbox4` (ASM)	739	718	2.28	1,520	- 29.9%
`mixcol`	498	1,047	1.56	1,262	- 41.8%
`mixcol4`	498	939	1.74	1,224	- 43.5%
`sbox` & `mixcol`	346	566	2.89	612	- 71.8%
`sbox` & `mixcol4` (C)	346	458	3.57	564	- 74.0%
`sbox` & `mixcol4` (ASM)	346	337	4.86	480	- 77.9%
`sbox4s` & `mixcol4s` (C)	316	458	3.57	568	- 73.8%
`sbox4s` & `mixcol4s` (ASM)	316	219	7.47	412	- 81.0%
`sbox4s` & `mixcol4s`, unrolled	316	196	8.35	896	- 58.7%
T-lookup (Gladman), 1 KB	436	1,585	1.03	9,956	+ 359.2%
T-lookup (Gladman), 4 KB	436	1,097	1.49	10,900	+ 402.8%
T-lookup (Gladman), 4/1 KB	532	1,091	1.50	12,272	+ 466.1%
T-lookup (Gladman), 4/4 KB	467	1,105	1.48	15,104	+ 596.7%

Table 7.3: AES-128 decryption, precomputed key schedule: Performance and code size.

	Key exp.	**Decr. perf.**		**Code size**	
Implementation	cycles	cycles	speedup	bytes	rel. change
No extensions (pure SW)	739	1,955	1.00	2,520	0.0%
`sbox`	647	1,555	1.26	1,592	- 36.8%
`sbox4` (C)	739	1,435	1.36	1,784	- 29.1%
`sbox4` (ASM)	739	1,061	1.84	1,676	- 33.5%
`mixcol`	498	1,078	1.81	1,548	- 38.6%
`mixcol4`	498	970	2.02	1,244	- 50.6%
`sbox` & `mixcol`	346	566	3.45	608	- 75.9%
`sbox` & `mixcol4` (C)	346	458	4.27	560	- 77.8%
`sbox` & `mixcol4` (ASM)	346	330	5.92	484	- 80.8%
`sbox4s` & `mixcol4s` (C)	316	459	4.26	564	- 77.6%
`sbox4s` & `mixcol4s` (ASM)	393	218	8.97	456	- 81.9%
`sbox4s` & `mixcol4s`, unrolled	393	196	9.97	944	- 62.5%
T-lookup (Gladman), 1 KB	1,517	1,292	1.51	12,816	+ 408.6%
T-lookup (Gladman), 4 KB	1,828	1,262	1.55	14,640	+ 481.0%
T-lookup (Gladman), 4/1 KB	1,828	1,260	1.55	15,408	+ 511.4%
T-lookup (Gladman), 4/4 KB	1,826	1,272	1.54	18,512	+ 634.6%
T-lookup (Gladman), 4/4/1 KB	1,085	1,099	1.78	18,512	+ 634.6%
T-lookup (Gladman), 4/4/4 KB	885	1,122	1.74	20,500	+ 713.5%

at the cost of quite significant increases in code size.

The results for AES-128 encryption with on-the-fly key expansion are given in

Table 7.4: AES-128 encryption, on-the-fly key expansion: Performance and code size.

	Encr. perf.		Code size	
Implementation	cycles	speedup	bytes	rel. change
No extensions (pure SW)	2,239	1.00	1,636	0.0%
`sbox`	1,595	1.40	952	- 41.8%
`sbox4`	1,618	1.38	1,696	- 3.7%
`mixcol`	1,294	1.73	1,260	- 23.0%
`mixcol4`	1,186	1.89	1,212	- 25.9%
`sbox` & `mixcol` (C)	747	3.00	580	- 64.6%
`sbox` & `mixcol` (ASM)	505	4.43	396	- 75.8%
`sbox` & `mixcol4` (C)	639	3.50	532	- 67.5%
`sbox` & `mixcol4` (ASM)	397	5.64	348	- 78.7%
`sbox4s` & `mixcol4s` (C)	616	3.63	528	- 67.7%
`sbox4s` & `mixcol4s` (ASM)	255	8.78	260	- 84.1%
`sbox4s` & `mixcol4s`, unrolled	226	9.91	852	- 47.9%
T-lookup, 1 KB	2,066	1.08	2,572	+ 57.2%
T-lookup, 4 KB	1,497	1.50	5,420	+ 231.3%
T-lookup, 4/1 KB	1,713	1.31	6,648	+ 306.4%
T-lookup, 4/4 KB	1,621	1.38	9,600	+ 486.8%

Table 7.5: AES-128 decryption, on-the-fly key expansion: Performance and code size.

	Decr. perf.		Code size	
Implementation	cycles	speedup	bytes	rel. change
No extensions (pure SW)	2,434	1.00	2,504	0.0%
`sbox`	1,867	1.30	1,564	- 37.5%
`sbox4`	1,715	1.42	1,748	- 30.2%
`mixcol`	1,605	1.52	1,648	- 34.2%
`mixcol4`	1,497	1.63	1,600	- 36.1%
`sbox` & `mixcol` (C)	698	3.49	580	- 76.8%
`sbox` & `mixcol` (ASM)	523	4.65	404	- 83.9%
`sbox` & `mixcol4` (C)	590	4.13	532	- 78.8%
`sbox` & `mixcol4` (ASM)	415	5.87	356	- 85.8%
`sbox4s` & `mixcol4s` (C)	557	4.37	520	- 79.2%
`sbox4s` & `mixcol4s` (ASM)	300	8.11	284	- 88.7%
`sbox4s` & `mixcol4s`, unrolled	262	9.29	996	- 60.2%
T-lookup, 1 KB	6,528	0.37	4,504	+ 79.9%
T-lookup, 4 KB	5,939	0.41	7,352	+ 193.6%
T-lookup, 4/1 KB	6,122	0.40	8,280	+ 230.7%
T-lookup, 4/4 KB	6,122	0.40	11,288	+ 350.8%
T-lookup, 4/4/1 KB	3,257	0.75	11,272	+ 350.2%
T-lookup, 4/4/4 KB	4,113	0.59	14,492	+ 478.8%

Table 7.4 and those for decryption in Table 7.5. All decryption implementations are supplied with the last round key. For encryption, speedups of up to 9.91 are achieved while the highest decryption speedup is 9.29. The fastest extensions allow for encryption in 226 cycles and decryption in 262 cycles. Note that

decryption is slightly slower as it uses the inverse equivalent cipher structure, which requires a more complex key expansion with additional InvMixColumns transformations. Some extensions allow quite significant reductions of code size. Implementations which make use of S-box extensions require no data memory accesses except for the loading of the input block and key and the storing of the output block. T-lookup implementations for encryption can achieve speedups of up to 1.5. Decryption functions with T-lookup are highly inefficient due to the more complex key expansion.

In order to get an idea of the worst-case execution time (WCET), we have also measured a single AES-128 encryption (rolled loops) with flushed data and instruction caches. Under these unfavorable conditions, encryption requires 565 cycles for a precomputed key schedule and 420 cycles for on-the-fly key expansion. Any subsequent encryption requires only little more than the number of cycles given in Tables 7.2 and 7.4. For unrolled loops, the first encryption naturally gets more costly with 761 cycles (precomputed) and 595 cycles (on-the-fly).

7.5 Exploiting Algorithmic Parallelism with Superscalar Architectures

The design of the high-performance instructions set extensions presented in Section 7.2.4 also allows to exploit the full parallelism of the AES algorithm in an efficient manner. This suitability for parallelization is also one of the main differences of our approach and the one of Nadehara et al. [180] based on T-table support. The use of T tables generates much larger intermediate values which in turn require additional effort for processing. In order to illustrate this difference, we look at the way how the two approaches calculate a single column after SubBytes, ShiftRows, and MixColumns from the according AES State at the beginning of a round. We investigate how the operations would be performed on a machine which could schedule all instructions in parallel and where the only restrictions would be imposed by the data dependencies of the instructions. The T-table approach is shown in Figure 7.10.

The four T-table lookups could be executed in parallel in the first clock cycle, yielding four 32-bit intermediate results. These four words then need to be XORed together with three instructions, taking two additional clock cycles. In contrast, our approach is much more economic in terms of intermediate results. This can be seen in Figure 7.11.

In the first clock cycle, two parallel `sbox4s` instructions generate the result after SubBytes, which is distributed amongst the upper half and lower half of two registers. Note that at the same time, the SubBytes transformation of a second column is generated and put in the remaining parts of the registers (left white in the figure). The two register halves are then combined in the second cycle with the help of a single `mixcol4s` instruction which yields the transformed column.

In order to transform the complete AES State under exploitation of the avail-

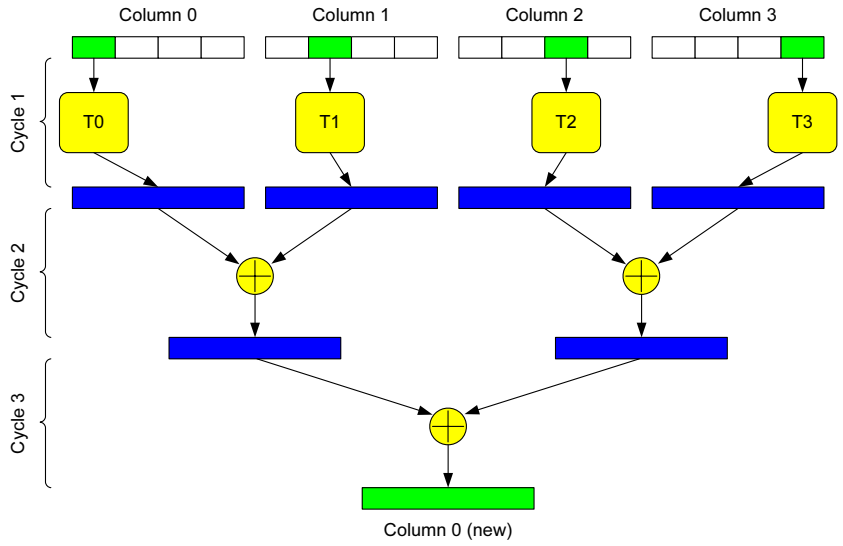

Figure 7.10: Dataflow for calculation a single State column after MixColumns with T-table extensions.

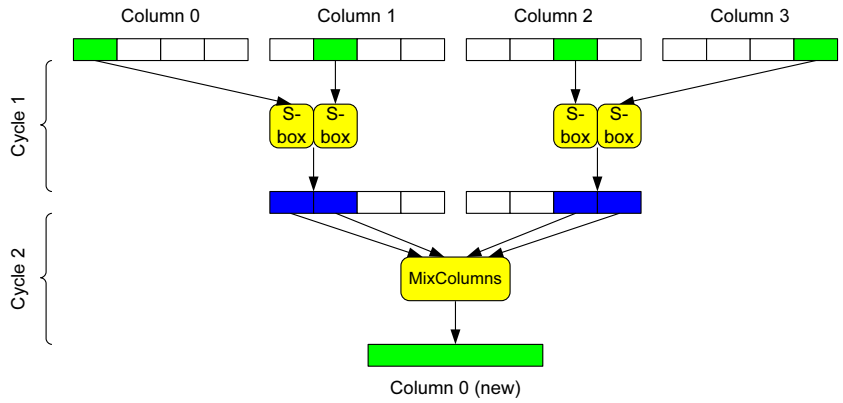

Figure 7.11: Dataflow for calculation a single State column after MixColumns with our proposed extensions.

able parallelism, the T-table approach would require to execute 16 instructions (i.e., 16 T-table lookups) in parallel. On the other hand, with our extension the State can be transformed with only four parallel instructions. But even if 16 (or more) parallel instructions could be executed in parallel, the T-table approach still requires one cycle more than our approach.

For practice, this means that a superscalar processor which is able to execute several of our proposed instructions in parallel can increase the performance significantly. With sufficient parallelism, the cycle count per block can be brought

down to a few dozen. At the same time the demands for memory performance can be kept relatively moderate.

For an implementation with precomputed key schedule, the complete parallelism of the AES algorithm can be exploited on a 6-way superscalar processor with four sets of the extensions and two independent data memory read ports. This means, that four of the execution pipelines can be kept busy with the AES transformations (performing `sbox4s`, `mixcol4s`, and `xor` instructions), while the other two pipelines continually load the subsequent round key. A complete AES round can then be executed in just three clock cycles. The instruction streams for the six parallel pipelines during a single round n are depicted in Figure 7.12, where each row corresponds to a specific clock cycle and each column shows the instructions executed in a specific pipeline (P1 to P6).

```
!  P1          P2          P3          P4          P5          P6
!-------------------------------------------------------------------
ld K(n,0)   ld K(n,1)   sbox4s     sbox4s     sbox4s     sbox4s
ld K(n,2)   ld K(n,3)   mixcol4s   mixcol4s   mixcol4s   mixcol4s
                        xor        xor        xor        xor
```

Figure 7.12: An AES round on a six-way processor with extensions.

It can be seen from Figure 7.12 that SubBytes and MixColumns are performed in the first and second clock cycle respectively (recall that ShiftRows is done implicitly by the custom instructions). At the same time, the two remaining pipelines load in two words of the round key per cycle. They are then XORed to the State in the third clock cycle in order to perform AddRoundKey.

A performance estimation for the hypothetical 6-way superscalar processor under optimal conditions results in 37 clock cycles for a complete AES-128 encryption (including loading of plaintext and storing of ciphertext). There are a number of additional conditions (e.g., loaded values are already available in the subsequent clock cycle) which would need to be satisfied to reach peak performance on such a hypothetical processor. However, even if some of these conditions are not met by an actual architecture, the resulting performance could still be very close to the maximum.

Figure 7.12 also shows that two of the execution pipelines are idle in the third clock cycle, which is a rather undesirable property. Considering the high implementation cost of a complete processor pipeline, it is normally much more efficient to limit the parallelism to such a degree, where all pipelines are kept busy all of the time. For our case of AES, a 4-way superscalar processor would deliver a much better balance of performance and cost. Three of the execution pipelines need to be equipped with our proposed extensions whereas a single data memory read port is sufficient for our implementation. Figure 7.13 illustrates how a single AES round n can be executed in just four cycles on such a processor. The four cycles for round n are shown grouped together in the middle of the

figure. As the rounds are executed in an interleaved fashion, the first line shows the last cycle of the previous round $n-1$, while the last line contains the first cycle of the subsequent round $n+1$.

```
!  P1            P2         P3         P4       ! Round
!-------------------------------------------------!------
ld K(n,3)     mixcol4s   xor        xor         ! n-1
                                                 !
ld K(n+1,0)   xor        xor        sbox4s      ! n
ld K(n+1,1)   sbox4s     sbox4s     sbox4s      ! n
ld K(n+1,2)   mixcol4s   mixcol4s   mixcol4s    ! n
ld K(n+1,3)   mixcol4s   xor        xor         ! n
                                                 !
ld K(n+2,0)   xor        xor        sbox4s      ! n+1
```

Figure 7.13: An AES round on a four-way processor with extensions.

In the four cycles of round n, SubBytes, ShiftRows, MixColumns and half of AddRoundKey (XOR) is performed. The second half of AddRoundKey of round n is performed in the first cycle of the next round. Note that the first pipeline is continually loading the round key for the subsequent round. Consequently, at the start of round n, the appropriate round key has already been loaded into registers. In this way, a single data memory read port is sufficient to achieve peak performance.

These examples for superscalar processors allow us to better compare our extensions to existing architectures which feature superscalar processing and/or a datapath width above 32. Note that we have not actually implemented such superscalar processors and that our performance figures are estimations based on pseudocode. Our code includes loading of input block and cipher key from memory, as well as storing of the output block back to memory. For our estimations we have assumed cache hits with a single-cycle latency for all loaded values.

7.6 Comparison with Related Work

Table 7.6 cites performance figures for most of the related work listed in Section 7.1. Note that it is difficult to compare the different approaches in a concise manner as some architectures have quite unique features. We categorized the different platforms by the width of their datapath (DPW), the number of instructions which can be executed per cycle (issue width, IW), and the number of data memory read ports ($DMRP$). Most architectures include dedicated lookup tables which allow parallel lookup. We have stated the number of lookup tables ($LUTs$), i.e., the number of possible parallel lookups, as well as the size of one table in bytes. The last two columns of Table 7.6 give the number of cycles

Table 7.6: AES-128 performance comparison with related work.

Platform & Reference	**DPW** bits	**IW/DMRP** #/#	**LUTs/Size** #/bytes	**Encr.** cycles	**Decr.** cycles
RISC-like [75]	128	1/1	16/1,024	32	32
PLX-128 [133]	128	1/1	0/0	609	n/a
Alpha (8W+) [38]	64	8/4	4/1,024	99	n/a
Alpha (4W+) [38]	64	4/2	4/1,024	164	n/a
Cryptonite [189]	64	2/1	16/256	71	83
RISC-like [75]	64	1/1	8/1,024	126	126
LEON2, ISE	**32**	**6/2**	**0/0**	**37**	**37**
CryptoManiac [262]	32	4/1	4/1,024	90	n/a
LEON2, ISE	**32**	**4/1**	**0/0**	**51**	**51**
RISC-like [180]	32	2/1	0/0	200	200
RISC-like [75]	32	1/1	4/1,024	315	315
Xtensa + ISE [202]	32	1/1	0/0	1,400	1,400
StrongARM [26]	32	1/1	0/0	311	n/a
LEON2, ISE	**32**	**1/1**	**0/0**	**196**	**196**
LEON2, COP (CPI) [119]	32	1/1	0/0	704	n/a
LEON2, COP (MM) [119]	32	1/1	0/0	1,228	n/a
LEON2, COP (MM) [213] [a]	32	1/1	0/0	1,494	n/a
Core 2 [169] [b]	128	4/1	0/0	178	n/a
Athlon 64 [167]	64	3/2	0/0	170	n/a
Pentium 4 [168]	32	3/1	0/0	251	n/a

[a] Performance calculated from time for encryption at 50 MHz.

[b] Bitslice implementation using the 128-bit XMM instructions. Performance includes transformations of standard data representation to and from bitslice format.

required for encryption and decryption of a 128-bit block with AES-128.

The fastest implementation with our proposed extensions is contained in the table with an indicated issue width of 1. However, as we have discussed in Section 7.5, our proposed extensions are also beneficial for processors with larger issue width. The estimated performance figures for the 4-way and 6-way superscalar processors described in Section 7.5 are also included in Table 7.6. Note that all our performance figures already include the loading of input block and cipher key from memory, as well as the storing of the output block back to memory.

Except for [202], [26] and our work, all architectures have either a datapath width greater than 32, an issue width greater than one and/or include dedicated parallel lookup tables. Our single-issue approach is nearly an order of magnitude faster than [202] and it has about the same performance of the approach in [180], which uses a superscalar processor with issue width 2. Despite the worse cited performance figures, the approach of [26] should be faster than our approach, but at the cost of a severe increase of the critical path and the need for non-standard parallel access to four processor registers. The CryptoManiac [262] with an issue width of 4 and four dedicated lookup tables of 1 KB each has

only half of the cycle count of our single-issue approach, and is slower than our superscalar approaches. The fastest architectures are those of Fiskiran et al. [75] and our 6-way superscalar processor. The solutions differ greatly, as Fiskiran et al. employ a 128-bit datapath with a large amount of dedicated lookup tables (16 tables of 1 KB each), which are combined with a dedicated XOR-tree.

Table 7.6 also includes the results of a LEON2 with an attached AES coprocessor (COP) [119, 213]. Both works have investigated a memory-mapped (MM) solution and Hodjat et al. have also examined an approach with a dedicated coprocessor interface (CPI) [119]. These works demonstrate impressively that the mere speed of an accelerator is not the important point to consider from a system's perspective. Hodjat et al. state in [119] that "the AES encryption itself takes only 11 cycles, but the complete program with loading the data and key, AES encryption, and returning the result back to the software routine takes a total of 704 cycles". Our worst-case execution times with flushed caches for precomputed key schedule (565 cycles with rolled loops, 761 cycles with unrolled loops) and on-the-fly key expansion (420 cycles with rolled loops, 595 cycles with unrolled loops)[3] compare very well to the coprocessor performance from [119, 213].

For comparison we have also specified the performance of an optimized AES implementations for the Pentium 4 (Northwood core) [168], for the Athlon 64 processor [167], and for a bitslice implementation on the Core 2 Duo (Conroe core) [169]. A single-issue LEON2 processor with our extensions has an area of about 50 kGates altogether and requires less cycles than the Pentium 4 (about 13.5 million gates). Moreover it can nearly reach the cycle count of the Athlon 64 (about 17 million gates) and Core 2 (about 72.5 million gates).

7.7 Summary and Conclusions

In this chapter we have presented instruction set extensions for 32-bit processors for the Advanced Encryption Standard. We have proposed byte-oriented and word-oriented custom instructions which can be combined in a number of different ways and which provide support for the most time-consuming transformations of AES. Our extensions are very flexible and can be used for encryption and decryption as well as with precomputed key schedule and on-the-fly key expansion. With hardware costs of about 3 kGates, AES-128 encryption and decryption is possible in 196 clock cycles. In relation to an AES implementation using only SPARC V8 instructions, speedups of up to 9.91 for encryption and 9.97 for decryption are achieved, while code size is reduced significantly. Furthermore, we have shown that our extensions can be implemented in a superscalar processor where they can compete very successfully with dedicated cryptographic processors and previously proposed instructions set extensions.

[3] The unrolling of loops results in a larger code size. Under worst-case conditions, there are more instruction cache misses, resulting in a worse performance.

8

Unified Support for Secret-Key and Public-Key Cryptography

In Chapter 5, we have demonstrated that the flexibility of instruction set extensions for public-key cryptography can be sufficiently large so that also secret-key algorithms can draw benefits. These synergies stem from similarities of the low-level support GF(2^m) arithmetic for EC-based public-key algorithms and the GF(2^m) operations used in some secret-key algorithms as AES or Twofish. An important difference between the public-key and secret-key domains is the size of the degree m. For public-key algorithms, m is typically in a range between 160 and 500, which exceeds the usual wordsize found in modern embedded processors. On the other hand, secret-key algorithms normally use much smaller degrees m which do not exceed the processor wordsize. Unlike in public-key algorithms, the operands can therefore reside in single registers and custom instructions can be better tuned to accommodate the necessary arithmetic operations.

In this chapter, we analyze the synergies of ECC over GF(2^m) and AES and we propose some modifications to the public-key extension from Chapter 5 in order to better accommodate secrete-key support. We present the design of a functional unit (FU) which can accelerate both types of cryptographic algorithms. The FU is basically a multiply-accumulate (MAC) unit which is able to perform multiplications and MAC operations on integers and binary polynomials. Polynomial arithmetic is a performance-critical building block of numerous cryptosystems using binary extension fields GF(2^m), including public-key primitives based on elliptic curves (e.g., ECDSA), secret-key ciphers (e.g., AES or Twofish), and hash functions (e.g., Whirlpool). We integrated the FU into the LEON2 SPARC V8 core and prototyped the extended processor in an FPGA.

All operations provided by the FU are accessible to the programmer through custom instructions.

Our results show that the FU allows to accelerate the execution of 128-bit AES by a factor of up to 1.78 compared to a conventional software implementation using only native SPARC V8 instructions. Moreover, the custom instructions reduce the code size by up to 87.4%. The FU increases the silicon area of the LEON2 core by just 8,352 gates and has almost no impact on its cycle time. Besides SPARC V8, the instruction set extensions described in this paper could be integrated into other 32-bit general-purpose RISC architectures, e.g., ARM or MIPS32.

8.1 Functional Units for Instruction Set Extensions

The custom instructions added by instruction set extensions can be executed in an application-specific FU or a conventional FU, such as the arithmetic/logic unit (ALU) or the multiplier, augmented with application-specific functionality. A typical example for the latter category is an integer multiplier able to execute not only the standard multiply instructions, but also custom instructions for long integer arithmetic [105]. Functional units are tightly coupled to the processor core and directly controlled by the instruction stream. The operands processed in FUs are read from the general-purpose registers and the result is written back to the register file. Hardware acceleration through custom instructions is cost-effective because tightly coupled FUs can utilize all resources already available in a processor, e.g., the registers and control logic. On the other hand, loosely-coupled hardware accelerators like coprocessors have separate registers, datapaths, and state machines. In addition, the interface between processor and coprocessor costs silicon area and may also introduce a severe performance bottleneck due to communication and synchronization overhead [119].

Application-specific FUs require less silicon area than coprocessors, but allow to achieve significantly better performance than "conventional" software implementations [152]. As we have also shown in Chapter 7 it might even be possible that instruction set extensions outperform a cryptographic coprocessor while demanding only a fraction of the silicon area.

In this chapter we present the design and implementation of a functional unit to accelerate the execution of both public-key and secret-key cryptography on embedded processors. The FU is basically a multiply-accumulate unit consisting of a (32×16)-bit multiplier and a 72-bit accumulator. It is capable of processing signed and unsigned integers as well as binary polynomials, i.e., the FU contains a so-called *unified multiplier*[1] [210]. Besides integer and polynomial multiplication and multiply-accumulate operations, the FU can also perform the reduction of binary polynomials modulo an irreducible polynomial of degree $m = 8$, such

[1] The term unified means that the multiplier uses the same datapath for both integers and binary polynomials.

as needed for AES encryption and decryption [57, 82]. The rich functionality provided by the FU facilitates efficient software implementation of a broad range of cryptosystems, including the "traditional" public-key schemes involving long integer arithmetic (e.g., RSA, DSA), Elliptic Curve Cryptography (ECC) over both prime fields $\mathbb{F}_p$ and binary extension fields $\mathbb{F}_{2^m}$, as well as the Advanced Encryption Standard (AES) [185].

A number of unified multiplier architectures for public-key cryptography, in particular ECC, have been published in the past [97, 210]. However, the FU presented in this chapter extends previous work in two important aspects. First, our FU supports not only ECC but also the AES, in particular the MixColumns and InvMixColumns operations. Second, we integrated the FU into the SPARC V8-compliant LEON2 softcore [80] and prototyped the extended processor in an FPGA. This allowed us, on the one hand, to evaluate the hardware cost and critical-path delay of the extended processor and, on the other hand, to analyze the impact of the FU on performance and code size of AES software. All reported execution times were measured on "working silicon" in form of an FPGA prototype.

The main component of our FU is a (32×16)-bit unified multiplier for signed/unsigned integers and binary polynomials. We used the unified multiplier architecture for ECC described in [97] as starting point for our implementation. The main contribution of this chapter is the integration of support for the AES MixColumns and InvMixColumns operations, which require besides polynomial multiplication also the reduction modulo an irreducible polynomial of degree $m = 8$. Hence, we focus in the remainder of this chapter on the implementation of the polynomial modular reduction and refer to [97] for details concerning the original multiplier for ECC. To the best of our knowledge, our functional unit is the first approach for integrating AES support into a unified multiplier for integers and binary polynomials.

Although the focus of this chapter is directed towards the AES, we point out that the presented concepts can also be applied to other block ciphers requiring polynomial arithmetic, e.g., Twofish, or to hash functions like Whirlpool, which has a similar structure as AES.

8.2 Arithmetic in Binary Extension Fields

The finite field $\mathbb{F}_q$ of order $q = p^m$ with p prime can be represented in a number of ways, whereby all these representations are isomorphic. The elements of fields of order 2^m are commonly represented as polynomials of degree up to $m-1$ with coefficients in the set $\{0, 1\}$. These fields are called *binary extension fields* and a concrete instance of $\mathbb{F}_{2^m}$ is generated by choosing an irreducible polynomial of degree m over $\mathbb{F}_2$ as reduction polynomial. The coefficient set of $\{0, 1\}$ allows for very efficient processing in digital computing systems since each coefficient can be represented by a bit. The arithmetic operations in $\mathbb{F}_{2^m}$ are defined as polynomial operations with a reduction modulo the irreducible polynomial. Binary extension fields have the advantage that addition has no carry propagation. This

feature allows efficient implementation of arithmetic in these fields in hardware. Addition can be done with a bitwise XOR. Multiplication may be realized with the simple shift-and-XOR method followed by reduction modulo the irreducible polynomial.

Binary extension fields play an important role in cryptography as they constitute a basic building block of both public-key and secret-key algorithms. For example, the NIST recommends to use binary fields as underlying algebraic structure for the implementation of elliptic curve cryptography (ECC) [112]. The degree m of the fields used in ECC is rather large, typically in the range between 160 and 500. The multiplication of elements of such large fields is very costly on 32-bit processors, even if a custom instruction for multiplying binary polynomials is available. On the other hand, the reduction of the product of two field elements modulo an irreducible polynomial $f(x)$ is fairly fast (in relation to multiplication) and can be accomplished with a few shift and XOR operations if $f(x)$ has few non-zero coefficients, e.g., if $f(x)$ is a trinomial [112].

Contrary to ECC schemes, the binary fields used in secret-key systems like block ciphers are typically very small. For example, AES and Twofish rely on the field $\mathbb{F}_{2^8}$. A multiplication of two binary polynomials of degree smaller or equal to 7 can be easily performed in one clock cycle with the help of a custom instruction like `gf2mul` [236]. However, the reduction of the product modulo an irreducible polynomial $f(x)$ of degree 8 is relatively slow when done in software, i.e., it requires much longer than one cycle. Therefore, it is desirable to provide hardware support for the reduction operation modulo irreducible polynomials of small degree.

8.3 Use of GF(2^m) Arithmetic in AES

The binary extension field GF(2^8) plays a central role in the AES algorithm [57]. Multiplication in GF(2^8) is part of the MixColumns operation and inversion in GF(2^8) is carried out in the SubBytes operation. The MixColumns/InvMixColumns operation is, in general, one of the most time-consuming parts of the AES [82]. Software implementations on 32-bit platforms try to speed up this operation either by using an alternate data representation [25] or by employing T-lookup tables [57]. However, as shown in Chapter 6, the use of the T-tables is disadvantageous for embedded systems since they occupy scarce memory resources, increase cache pollution, and may open up potential vulnerabilities to cache-based side channel attacks [23, 28, 147, 194, 248].

The MixColumns transformation of AES can be defined as multiplication in an extension field of degree 4 over $\mathbb{F}_{2^8}$ [57]. Elements of this field are polynomials of degree smaller or equal to 3 with coefficients in $\mathbb{F}_{2^8}$. The coefficient field $\mathbb{F}_{2^8}$ is generated by the irreducible polynomial $f(x) = x^8 + x^4 + x^3 + x + 1$ (0x11B in hexadecimal notation). For the extension field $\mathbb{F}_{2^8}[t]/(g(t))$ the irreducible polynomial $g(t)$ is $t^4 + 1$. The multiplier operand for MixColumns and InvMixColumns is fixed and its coefficients in $\mathbb{F}_{2^8}$ have a degree smaller or equal to 3. A multiplication in $\mathbb{F}_{2^8}[t]/(g(t))$ can be performed in three successive steps in

the following way:

1. Multiplication of binary polynomials.
2. Reduction of coefficients modulo $f(x)$.
3. Reduction of a polynomial over $\mathbb{F}_{2^8}$ modulo $g(t)$.

8.3.1 Functional Units for Instruction Set Extensions

Previous work on instruction set extensions for AES was aimed at both increasing performance as well as minimizing memory requirements. Nadehara et al. [180] designed custom instructions that calculate the result of the SubBytes and MixColumns operations in a dedicated functional unit. Bertoni et al. [26] proposed custom instructions to speed up AES software following the approach of [25]. Lim and Benaissa [159] implemented a subword-parallel ALU for binary polynomials that supports AES and ECC over GF(2^m). Their approach includes some very interesting concepts, like the support of arbitrary reduction polynomials and of GF(2^m) division which can be used to implement the AES S-box. But their solution has also some drawbacks, e.g., ECC is supported only up to certain length.

In the previous Chapters 6 and 7 we have presented approaches to reduce the memory requirements and to optimize performance of AES with dedicated functional units for custom instructions. In Chapter 5 we demonstrated the potential synergies of a unified support for asymmetric and symmetric cryptographic algorithms as we used three custom instructions originally designed for ECC (`gf2mul`, `gf2mac`, and `shacr`) to accelerate AES by a factor of up to 1.34. We used a multiplier with support for asymmetric cryptography to speed up AES without changing the underlying hardware. In this chapter we extended this multiplier to increase AES performance even further.

8.4 Design of a Unified Multiplier with AES Support

Our base architecture is the unified multiply-accumulate (MAC) unit presented in [97]. It is capable of performing unsigned and signed integer multiplication as well as multiplication of binary polynomials. Our original implementation of the MAC unit has been optimized for the LEON2 processor and consists of two stages. The first stage contains a unified (32×16)-bit multiplier that requires two cycles to produce the result of a (32×32)-bit multiplication. The second stage features a 72-bit unified carry-propagation adder, which adds the product to the accumulator.

Of the three steps described in Section 8.3, binary polynomial multiplication is already provided by the original multiplier from [97]. The special structure of the reduction polynomial $g(t)$ for step 3 allows a very simple reduction: The higher word (i.e., 32 bits) of the multiplication result after step 2 (with reduced

coefficients) is added to the lower word. This operation can be implemented in the second stage (i.e., the accumulator) of the unified MAC unit without much overhead. The only remaining operation is the reduction modulo $f(x)$ of the coefficients (step 2). In the following we introduce the basic ideas for integrating this operation into the unified multiplier presented in [97].

8.4.1 Basic Unified Multiplier Architecture

Figure 8.1 shows the structure of our baseline multiplier. The multiplier from [97] employs unified radix-4 partial product generators (PPGs) for unsigned and signed integers as well as binary polynomials. In integer mode, the partial products are generated according to the modified Booth recoding technique, i.e., three bits of the multiplier B are examined at a time. On the other hand, the output of each PPG in polynomial mode depends on exactly two bits of B. A total of $\lfloor n/2 \rfloor + 1$ partial products are generated for an n-bit multiplier B if performing an unsigned multiplication, but only $\lfloor n/2 \rfloor$ partial products in the case of signed multiplication or when binary polynomials are multiplied.

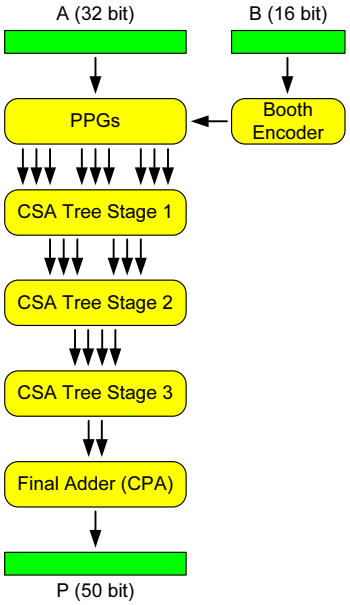

Figure 8.1: Basic unified (32×16)-bit multiplier.

The unified MAC unit described in [97] uses *dual-field adders (DFAs)*. The DFAs are arranged in an array structure to sum up the partial products. However, we decided to implement the multiplier in form of a Wallace tree [252] in order to minimize the critical path delay. Another difference between our unified MAC unit for the LEON2 core and the design from [97] is that our unit adds the multiplication result to the accumulator in a separate stage. Therefore, our

unified (32×16)-bit multiplier has to sum up only the 9 partial products generated by the modified Booth recoder. This is done in a Wallace-tree structure with four summation stages using dual-field adders. The first three stages use unified carry-save adders (CSAs) with either (3:2) or (4:2) compressors. The result of each adder is in a redundant form, split up into a carry vector and a sum vector. This redundant representation allows for addition without carry propagation and minimizes the contribution of these summation stages to the overall circuit delay. The fourth and last stage consists of a unified carry-propagate adder (CPA), which produces the final result in non-redundant representation.

8.4.2 Concepts for Support of AES MixColumns Multiplication

Two observations are important to integrate AES MixColumns support into the basic unified multiplier:

1. For AES MixColumns and InvMixColumns, the coefficients of the constant multiplier B have a degree smaller or equal to three (i.e., only the lowest four bits can be set). At least half of the PPGs will therefore produce a partial product of 0 in polynomial mode. We denote those PPGs as "idle".

2. As binary polynomials have no carry propagation in addition, the carry vectors of the carry-save summation stages will always be 0 in polynomial mode.

When two polynomials over $\mathbb{F}_{2^8}$ are multiplied with the unified multiplier in polynomial mode, the result will be incorrect. The reason for this is that the coefficients of the polynomial over $\mathbb{F}_{2^8}$ will exceed the maximum degree of 7, i.e., they will be in non-reduced form. The coefficient bits of degree greater than 7 are added to the bits of the next-higher coefficient in the partial product generators and in the subsequent summation stage. But in order to perform a reduction of the coefficients to non-redundant form (degree smaller or equal to 7), it is necessary to have access to the excessive bits of each coefficient. In the following we will denote these excessive bits as *reduction bits*. The reduction bits indicate whether the irreducible polynomial $f(x)$ must be added to the respective coefficient with a specific offset in order to reduce the degree of the coefficient.

The reduction bits can be isolated in separate partial products. A modification of the PPGs can be prevented by making use of the "idle" PPGs to process the highest three bits of every coefficient of the multiplicand A. This is achieved with the following modifications:

- The "not-idle" PPGs are supplied with multiplicand A where only the lowest 5 bits of each coefficient are present (i.e., A AND 0x1F1F1F1F).

- Multiplicand A for the "idle" PPGs contains only the highest 3 bits of every coefficient (i.e., A AND 0xE0E0E0E0) and is shifted to the right by 4 bits.

- The multiplier B has the lower nibble (4 bits) of each byte replicated in the respective higher nibble (e.g., 0x0C0D → 0xCCDD).

These modifications entail a different generation of partial products but still result in the same multiplication result after the summation tree. This is because processing of the multiplicand A is spread across all PPGs (which is done by the masking of A). The "idle" PPGs are activated through replication of the nibbles of the multiplier B. Moreover, the "idle" PPGs produce partial products with a higher weight than intended, which is compensated by the right-shift of the input multiplicand A for these PPGs. Figure 8.2 and 8.3 illustrate the partial product generation for a multiplication of a polynomial over $\mathbb{F}_{2^8}$ of degree 1 (16-bit multiplicand A) with a polynomial of degree 0 (8-bit multiplier B). Note that partial product 1 in Figure 8.2 is split into the partial products 1 and 3 in Figure 8.3. The same occurs for partial product 2, which is split into the partial products 2 and 4. The PPG scheme in Figure 8.3 yields partial products which directly contain the reduction bits.

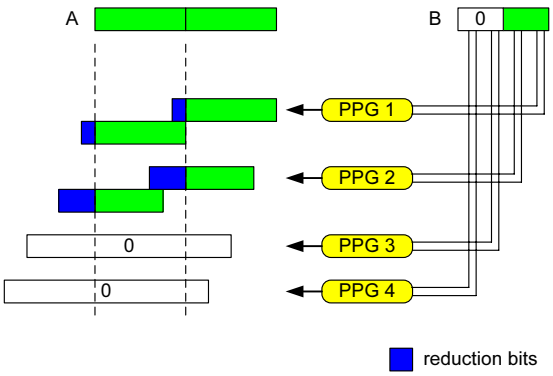

Figure 8.2: Multiplication of polynomials over $\mathbb{F}_{2^8}$ with a radix-4 multiplier for binary polynomials.

To determine whether the reduction polynomial needs to be added to a coefficient of the multiplication result with a specific offset, it is necessary to combine (add) reduction bits with the same weight from different partial products. In order to minimize delay, these few additional XOR gates are placed in parallel to the summation tree stages. The resulting reduction bits determine the value of the so-called *reduction vectors*, which can hold several reduction polynomials with different offsets. The reduction vectors reduce the coefficients to non-redundant form and are injected via the carry vectors of the summation tree. More specifically, if a reduction bit is set, then a portion of a carry-vector (with the correct offset) is forced to the value of the reduction polynomial $f(x)$ (0x11B), otherwise it is left 0. Reduction vectors for different coefficients can be injected in the same carry vector, as long as they do not overlap and the carry-vector is long enough. Thus, by making use of the "idle" PPGs and the carry vectors of the summation tree, the multiplier can be extended to support

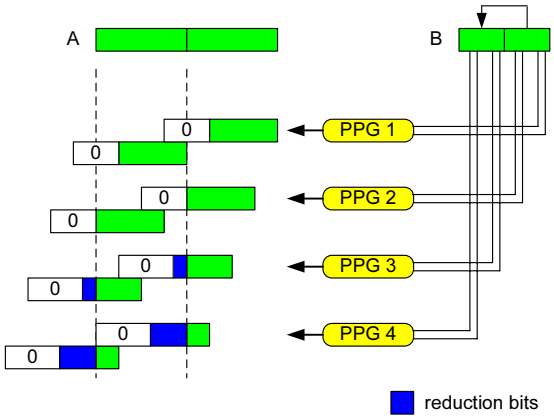

Figure 8.3: Multiplication of polynomials over $\mathbb{F}_{2^8}$ with the modified PPG-scheme for AES support.

AES MixColumns multiplication.

8.5 Implementation Details

The general concepts for integrating AES MixColumns support into the unified multiplier of [97] are described in Section 8.4.2. Figure 8.4 shows our modified multiplier with all additional components. *PPG Input Masking* and *Nibble Replication* make sure that the partial products are generated in a redundant fashion where the reduction bits are subsequently accessible. *Reduction Bit Addition* adds up reduction bits of coefficients of partial products with the same weight. *Reduction Vector Insertion* conditionally injects reduction polynomials for the coefficients with different offsets, depending on the reduction bits. The result P will be a polynomial over $\mathbb{F}_{2^8}$ of degree 4 with fully reduced coefficients. In the following we briefly describe the implementation of the additional components.

8.5.1 PPG Input Masking

The AES MixColumns mode is controlled with the signal *ff_mix*. This signal selects the input multiplier A for the PPGs either as unmodified or masked (and shifted) as described in Section 8.4.2.

8.5.2 Multiplier Nibble Replication

In our implementation the multiplier B is set by the processor in dependance on the required operation (MixColumns or InvMixColumns). Nibble replication is therefore performed outside of our multiplier. It could also be done within the multiplier, where it would just require an additional multiplexer for the multiplier B controlled by *ff_mix*.

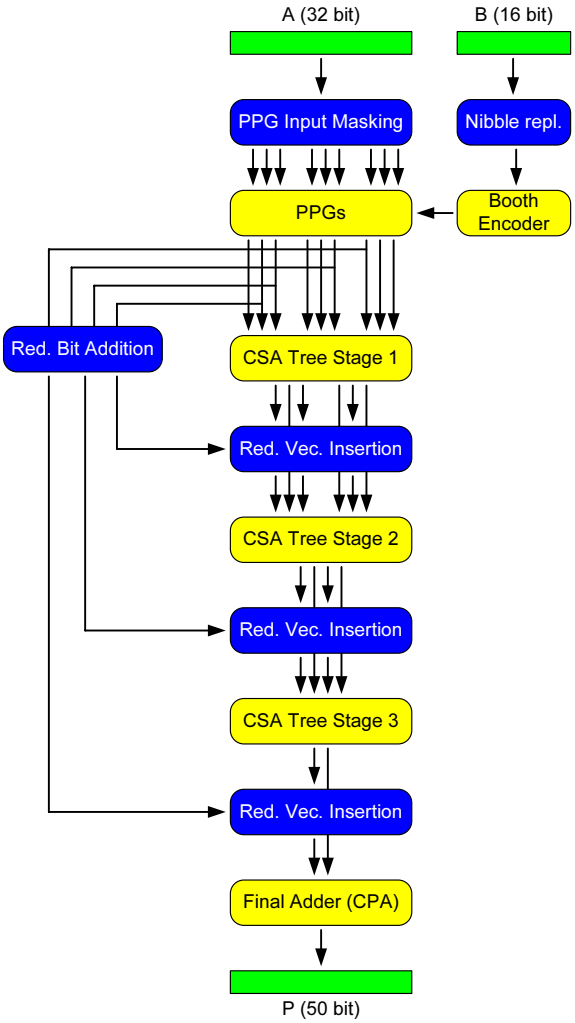

Figure 8.4: Proposed unified (32×16)-bit multiplier with AES support.

8.5.3 Reduction Bit Addition

Reduction bits of the same weight are XORed in parallel to the summation tree stages. For the (32×16)-bit case, the resulting reduction bits have contributions from one, two, or four partial products.

8.5.4 Reduction Vector Insertion

For each reduction vector, the *ff_mix* signal and the corresponding reduction bit are combined with a logical AND. The result is used to conditionally inject the reduction polynomial over a logical OR with the required bit lines of a carry

vector. This is illustrated in Figure 8.5. Reduction bits which have contributions from more partial products are used in later stages of the summation tree than reduction bits which depend on less partial products.

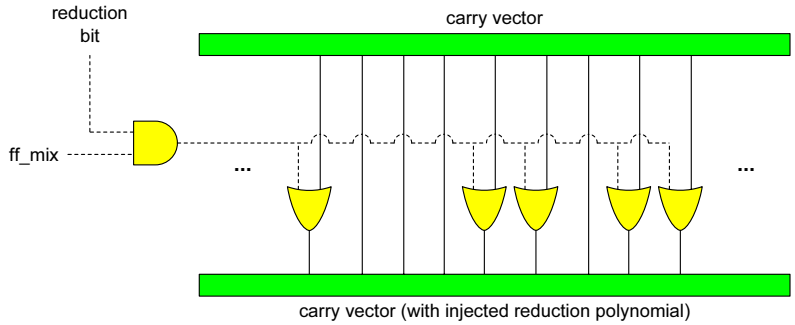

Figure 8.5: Conditional injection of the reduction polynomial into a section of a carry vector.

8.6 Experimental Results

We integrated our functional unit into the SPARC V8-compatible LEON2 core [80] and prototyped the extended processor in an FPGA. For performing AES MixColumns and InvMixColumns, four custom instructions were defined: Two of these instructions (`mcmuls`, `imcmuls`) can be used for the MixColumns and InvMixColumns transformation only, while the other two (`mcmacs`, `imcmacs`) include an addition of the transformation result to the accumulator. The latter two instructions write their result only to the accumulator registers and not to the general-purpose register file. They require two clock cycles to produce the result[2]. If the subsequent instruction does not need the multiplication result or access to the multiply-accumulate unit, then it can be processed in parallel to the multiply instruction, resulting in one cycle per instruction. In addition, our new custom instructions assemble the 32-bit multiplicand for AES multiplication from the two source register operands of the instruction (the 16 higher bits of the first register and the 16 lower bits of the second register), in order to facilitate the AES ShiftRows/InvShiftRows transformation.

8.6.1 Silicon Area and Critical Path

The impact of our modifications on the critical path of the multiplier is very small. One additional multiplexer delay is required to select the input for the PPGs. The reduction bits are added in parallel to the summation tree, which

[2]Although the multiply-accumulate unit takes three cycles for the calculation, subsequent instructions can access the result after two cycles without a pipeline stall due to the implementation characteristics of the accumulator registers.

should not extend the critical path. For injection of the reduction vectors, there is one additional OR-delay for the 2nd, 3rd and 4th summation tree stage, i.e., in the worst case three OR-delays altogether.

We synthesized the original unified multiplier from [97] (unimul32x16) and our proposed unified multiplier with AES support (unimul_mix32x16) using a 0.13 µm standard-cell library in order to estimate the overhead in silicon area and the impact on the critical path delay. These results were compared with the conventional (32×16)-bit integer multiplier that is part of the LEON2 soft-core (intmul32x16). We also made comparisons including the enclosing unified multiply accumulate units (unimac32x16, unimac_mix32x16) and the five-stage processor pipeline, denoted as integer unit (IU). The results are summarized in Table 8.1.

Table 8.1: Area and delay of the functional units and the extended LEON2 core.

	Minimal Delay		**Typical Delay**	
FU/Component	Area (GE)	Delay (ns)	Area (GE)	Delay (ns)
intmul32x16	7,308	2.05	5,402	2.50
unimul32x16	9,660	2.15	7,413	2.50
unimul_mix32x16	9,988	2.21	8,418	2.50
unimac32x16	14,728	2.53	12,037	3.00
unimac_mix32x16	16,145	2.56	12,914	3.00
LEON2 IU (intmul32x16)	27,250	2.59	17,867	4.97
LEON2 IU (unimac32x16)	38,705	2.77	24,927	5.00
LEON2 IU (unimac_mix32x16)	39,306	2.85	26,219	4.99

All results in Table 8.1 are given for the minimal and for a typical critical path delay. The former give an estimate of the maximum frequency with which the processor can be clocked, while the latter allow to assess the increase in silicon area due to our proposed modifications. Taking a LEON2 processor with a unified MAC unit for ECC (unimac32x16) as reference, our modifications for AES support increase the critical path by about 5% and the silicon area by less than 1.3 kGates. The overall size of the FU with support for ECC and AES is approximately 12.9 kGates when synthesized for a delay of 3 ns. However, it must be considered that the "original" (32×16)-bit integer multiplier of the LEON2 core has an area of about 5.4 kGates. Therefore, the extensions for ECC and AES increase the size of the multiplier by just 7.5 kGates and the overall size of the LEON2 core by approximately 8.35 kGates.

8.6.2 AES Performance

In order to estimate the achievable speedup with our proposed FU, we prototyped the extended LEON2 on an FPGA board. We evaluated AES encryption and decryption functions with 128-bit keys (AES-128) both with precomputed key schedule and on-the-fly key expansion. The number of cycles was determined with an integrated cycle counter using the timing code of the well-known AES

software implementation of Brian Gladman [82]. Note that the AES decryption function with on-the-fly key expansion is supplied with the last round key. The code size for each implementation is also listed, which encompasses all required functions as well as any necessary constants (e.g., S-box lookup table).

Table 8.2: AES-128 encryption and decryption: Performance and code size.

	Key exp.	**Performance**		**Code size**	
Implementation	cycles	cycles	speedup	bytes	rel. change
Encryption, precomputed key schedule					
No extensions (pure SW)	739	1,637	1.00	2,168	0.0%
`mcmuls` (C)	498	1,011	1.62	1,240	−42.8%
`sbox4s` & `mcmuls` (ASM)	316	260	6.30	460	−78.8%
Decryption, precomputed key schedule					
No extensions (pure SW)	739	1,955	1.00	2,520	0.0%
`mcmuls` (C)	316	1,299	1.51	1,572	−37.6%
`sbox4s` & `mcmuls` (ASM)	465	259	7.55	520	−79.4%
Encryption, on-the-fly key expansion					
No extensions (pure SW)	–	2,239	1.00	1,636	0.0%
`mcmuls` (C)	–	1,258	1.78	1,228	−21.3%
`sbox4s` & `mcmuls` (ASM)	–	296	7.56	308	−81.2%
Decryption, on-the-fly key expansion					
No extensions (pure SW)	–	2,434	1.00	2,504	0.0%
`mcmuls` (C)	–	1,596	1.53	1,616	−35.5%
`sbox4s` & `mcmuls` (ASM)	–	305	7.98	316	−87.4%

Table 8.2 specifies the number of clock cycles per encryption/decryption and the code size for implementations using precomputed key schedule as well as on-the-fly key expansion. Our baseline implementation is written in C using only native SPARC V8 instructions. The "`mcmuls` implementation" refers to an implementation written in C where MixColumns or InvMixColumns is realized using our proposed functional unit. The "`sbox4s` & `mcmuls` implementation" is written in assembly and uses our multiplier as well as an additional custom instruction for performing the S-box substitution. The `sbox4s` instruction has already been introduced in Chapter 7.

The C implementations can be sped up with the proposed custom instructions by a factor of up to 1.78. However, our extensions are designed to deliver maximal performance in combination with the custom instruction for S-box substitution described in Chapter 7. By combining these extensions, a 128-bit AES encryption can be done in less than 300 clock cycles, which corresponds to a speedup factor of between 6.3 (pre-computed key schedule) and 7.98 (on-the-fly key expansion) compared to the baseline implementation. Moreover, the custom instructions for AES reduce the code size by up to 87.4%.

For the AES extension presented in Chapter 7, we have used a custom functional unit to implement the instructions for MixColumns and InvMixColumns. The impact on both silicon area and AES performance is similar, and the custom functional unit has some slight advantages. However, the multiply-

accumulate unit with AES still offers some potential for optimization: The AES performance can be further improved by reducing the latency of the multiply-accumulate unit. With a (32×32)-bit multiplier and integration of the accumulation into the summation tree (as proposed in [97]), an instruction for MixColumns/InvMixColumns could be executed in a single cycle and could also include the subsequent AddRoundKey transformation. With such an instruction, a complete AES round could principally be executed in only 12 clock cycles, and a complete AES-128 encryption or decryption in about 160 cycles (including all loads and stores of the data and key)[3].

8.6.3 Comparison with Designs Using an AES Coprocessor

Hodjat et al. [119] and Schaumont et al. [213] attached an AES coprocessor to the LEON2 core and analyzed the effects on performance and hardware cost. The implementation reported by Hodjat et al. used a dedicated coprocessor interface to connect the AES hardware with the LEON2 core. Schaumont et al. transferred data to and from the coprocessor via memory-mapped I/O. Both systems were prototyped on a Xilinx Virtex-II FPGA on which the "pure" LEON2 core consumes approximately 4,856 LUTs, leaving some 5,400 LUTs for the implementation of the AES coprocessor. Table 8.3 summarizes the execution time of a 128-bit encryption and the additional hardware cost due to the AES coprocessor. For comparison, the corresponding performance and area figures of the extensions proposed in this paper are also specified.

Table 8.3: Performance and cost of AES coprocessor vs. instruction set extensions.

		Performance	**HW cost**
Reference	**Implementation**	cycles	LUTs
Hodjat [119]	Coprocessor (COP interface)	704	4,900
Schaumont [213]	Coprocessor (mem. mapped)	1,494	3,474
This work	ISE for MixColumns	1,011/1,299	3,194
This work	ISE for MixColumns + S-box	260	3,695

Hodjat et al.'s AES coprocessor uses about 4,900 LUTs (i.e., requires more resources than the LEON2 core) and is able to encrypt a 128-bit block of data in 11 clock cycles. However, loading the data and key into the coprocessor, performing the AES encryption itself, and returning the result back to the software routine takes 704 cycles altogether [119, page 492]. Schaumont et al.'s coprocessor with the memory-mapped interface requires less hardware and is slower than the implementation of Hodjat et al. The performance of our AES extensions lies between the two coprocessor systems. As mentioned in Section 8.6.2, the custom instruction for S-box substitution would allow to reduce the execution time of 128-bit AES encryption to 260 cycles, which is significantly faster than

[3]Note that this would also require a flexible access to the accumulator registers, e.g., so that round keys could be loaded directly.

the coprocessor systems. The additional hardware cost of the FU is comparable to that of the two coprocessors. However, contrary to AES coprocessors, the FU presented in this paper supports not only the AES, but also ECC over both prime fields and binary extension fields.

8.7 Summary and Conclusions

In this chapter we introduced a functional unit for increasing the performance of embedded processors when executing cryptographic algorithms. The main component of the functional unit is a unified multiply-accumulate (MAC) unit capable of performing integer and polynomial multiplication as well as reduction modulo an irreducible polynomial of degree 8. Due to its rich functionality and high degree of flexibility, the functional unit facilitates efficient implementation of a wide range of cryptosystems, most notably ECC and AES. When integrated into the LEON2 SPARC V8 processor, the functional unit allows to execute a 128-bit AES encryption with precomputed key schedule in about 1,000 clock cycles. Hardware support for the S-box operation further reduces the execution time to 260 cycles, which is more than six times faster than a conventional software implementation on the LEON2 processor. The hardware cost of the AES extensions is roughly 1.3 kGates and the additional area for the support of ECC and AES amounts to just 8.35 kGates altogether. These results confirm that the functional unit presented in this paper can be a flexible and cost-effective alternative to a cryptographic coprocessor.

9

Instruction Set Extensions for AES on 8-Bit Architectures

The favorable properties of cryptography instruction set extensions on 32-bit processors lead to the natural question, whether some of these benefits can be rolled over to architectures with smaller word sizes. This question is especially interesting, as small microcontrollers will be present in many new appliances. In this chapter we demonstrate these benefits at the example of the 8-bit Advanced Virtual RISC (AVR) architecture. We propose extensions which can increase the performance of AES by a factor of up to 3.6. On the other hand, code size can be reduced by up to 75%. The hardware cost of our approach amounts to about 800 gates.

9.1 Previous Work

So far, almost all architectural extensions for cryptography have been proposed for processors of a word size of 32 bits or more. An exception is the work of Eberle et al. which describes support for ECC over binary extension fields GF(2^m) for the AVR architecture [65]. A custom 8-bit microcontroller for AES has been presented by Chia et al. in [48], with a focus on minimizing code size rather than performance.

To the best of our knowledge, our work is the first to propose concrete architectural enhancements for a secret-key algorithm targeting a common 8-bit microcontroller architecture.

9.2 Description of the AVR Architecture

Advanced Virtual RISC (AVR) by Atmel is an architecture for 8-bit microcontrollers [16]. AVR features a separate access to data and program memory, which is commonly denoted as Harvard architecture. The program memory is implemented as an in-system programmable FLASH memory whose size can vary from 1 KB to 256 KB depending on the concrete microcontroller model. The available RAM and internal in-system programmable EEPROM also depends on the model. RAM size can vary from 32 bytes to 8 KB. Non-volatile memory in the form of EEPROM is commonly used to store parameters and is included in most devices. The maximum size of the EEPROM is currently 4 KB.

The AVR instruction set consists of about 110 instructions. Most of them have a 16-bit encoding and can operate on the 32 general-purpose registers specified by the architecture. Of these 32 registers, six have additional functionality. They can be used as three independent memory pointers which can hold 16-bit addresses. Most of the instructions require only a single clock cycle to execute. Only a few instructions take between two and four clock cycles to finish. The instructions are directly executed from the FLASH memory, which also limits the maximum clock frequency. Some of the AVR microcontrollers not only support in-system programming but also self programming. This functionality allows the controllers to reload source code during runtime and increases the flexibility of implemented applications.

In order to address the requirements of low-power designs, the supply voltage for the AVR family microcontrollers ranges from 1.8 V to 5.5 V. Moreover, they are equipped with a sleep controller which supports various modes and the operation frequency can be controlled by software to support power save modes. The AVR family is built to support clock frequencies of up to 20 MHz.

The AVR microcontrollers are explicitly designed to be programmed in C. There are various free software development kits available, e.g., avr-gcc for C compilation and AVR Studio for assembly and simulation. The availability of free development tools supports the widespread use of the AVR controllers in various embedded applications like sensor nodes (e.g., MICAz [55]).

In 2008, Atmel announced the introduction of a new line of AVR microcontrollers named XMEGA [17]. The new devices will have FLASH memory sizes ranging from 16 KB to 384 KB, SRAM sizes between 2 KB and 32 KB, and EEPROM sizes between 1 KB and 4 KB. Operating voltages range between 1.6 V and 3.6 V and the maximum clock frequency is 32 MHz. The XMEGA devices contain cryptographic support in the form of instruction set extensions for DES and a coprocessor for AES. The DES extensions work on 16 registers simultaneously and can perform a complete DES round with a single instruction [18]. The AES coprocessor supports a key size of 128 bits and requires 375 clock cycles[1] to encrypt or decrypt a singe 128-bit block [19].

[1] This figure excludes the time for loading the key and block into the coprocessor, selection of the mode of operation, and read-back of the result.

9.3 Our Proposed AES Extensions

All AES extensions proposed in the literature so far try to make full use of the 32-bit datapath of the underlying processor [26, 180]. Therefore, none of these solutions can be scaled down to an 8-bit architecture in a straightforward way. As we will show in this section, it is however possible to reuse some of the important concepts of these 32-bit approaches to arrive at a worthwhile solution for small microcontrollers.

9.3.1 Support for AES Encryption

We propose three instructions to speed up AES encryption, whereby two instructions are intended to speed up the AES round transformations, while the third instruction is conceived for use in the final round and also in the key expansion. The instruction formats fully adhere to the AVR architecture and therefore allow for easy integration. All instructions use similar hardware components and a small and flexible functional unit can be easily designed to reach the maximal speed of current state-of-the-art AVR cores (which ranges at the time of writing at around 20 MHz).

Our basic concept is to use the capability of typical AVR microcontrollers to retrieve two register values per clock cycle [15]. With appropriate selection of the register operands, all four AES round transformations can be executed for two State bytes with only a few instructions. In the best case, a complete round for an AES State column (contained in registers) can be processed and stored back to the original registers in only 15 clock cycles.

The functionality of the two instruction variants `aesenc(1)` and `aesenc(2)` is depicted in Figure 9.1. Note that the symbols $\bigoplus$ and $\bigotimes$ denote addition (conforming to bitwise XOR) and multiplication in the Galois field GF(2^8), respectively. These instructions have the same format as the integer multiplication instruction `mul` of the basic AVR architecture:

```
aesenc(1) Rd, Rr
aesenc(2) Rd, Rr
```

First, the values from the two specified registers `Rd` and `Rr` are substituted according to the AES S-box. Depending on the instruction variant, the substituted bytes are multiplied with specific constants from the field GF(2^8). Two of the multiplication results are then combined with the values from the registers `R0` and `R1` by means of an XOR operation. The resulting values are stored to the registers `R0` and `R1`.

The intended use of the `aesenc(x)` instructions (where $x \in \{1, 2\}$) is to perform all transformations of a single AES round on two bytes of a State column with merely two invocations. The GF(2^8) constants have been chosen carefully from the AES MixColumns matrix. Each invocation of `aesenc(x)` conforms to the processing of a quadrant of that matrix. Due to the symmetry of the MixColumns matrix, there are only two distinct quadrants, cf. Figure 9.2 where

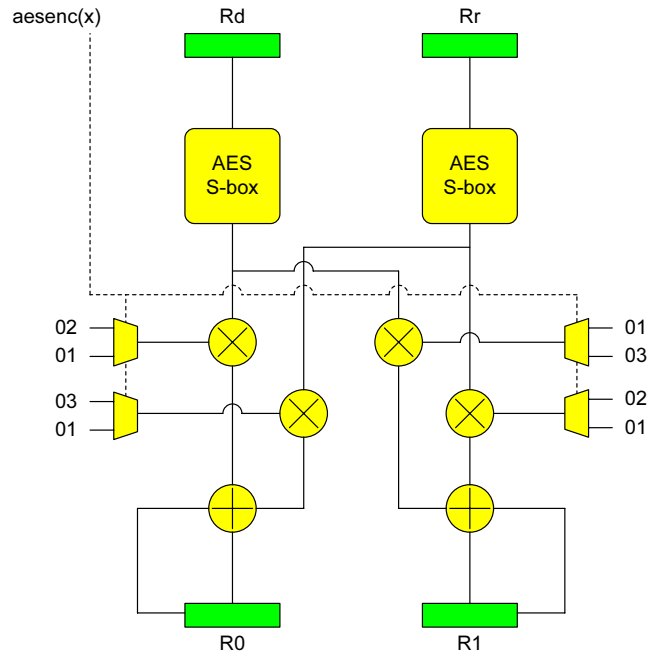

Figure 9.1: AES extensions for a normal encryption round.

quadrants 1 and 4 and quadrants 2 and 3 are equal, respectively. Therefore, the two variants of the `aesenc(x)` instruction are sufficient to transform the complete AES State.

$$\begin{matrix} & & & \\ Q1 & 02 & 03 & 01 & 01 \\ & 01 & 02 & 03 & 01 \\ \hline Q3 & 01 & 01 & 02 & 03 \\ & 03 & 01 & 01 & 02 \end{matrix} \begin{matrix} \\ Q2 \\ \\ Q4 \end{matrix}$$

Figure 9.2: Quadrants of the MixColumns matrix.

The `aesenc(x)` instructions can be used to produce two State bytes at the end of a round from the according four State bytes at the start of the round and the corresponding two bytes of the round key. In order to do this, the two bytes of the round key are loaded into `R1` and `R0` and then `aesenc(1)` and `aesenc(2)` are invoked with the according State bytes to produce a half of the resulting State column. The feedback from `R1` and `R0` into the final XOR stage (cf. Figure 9.3) has a dual functionality: On the first invocation of `aesenc(x)`, the round key bytes are added to the intermediate result. On the second invocation, this intermediate result is combined with the contribution from the other State bytes.

The `aesenc(1)` instruction handles quadrants 1 and 4 of the MixColumns matrix. Its functionality can be described more formally as follows (where $S(a)$

stands for the S-box lookup of a):

$$\begin{aligned} \texttt{R0} &= 02 \otimes S(\texttt{Rd}) \oplus 03 \otimes S(\texttt{Rr}) \oplus \texttt{R0} \\ \texttt{R1} &= 01 \otimes S(\texttt{Rd}) \oplus 02 \otimes S(\texttt{Rr}) \oplus \texttt{R1} \end{aligned}$$

Accordingly, the `aesenc(2)` instruction deals with quadrants 2 and 3 of the matrix:

$$\begin{aligned} \texttt{R0} &= 01 \otimes S(\texttt{Rd}) \oplus 01 \otimes S(\texttt{Rr}) \oplus \texttt{R0} \\ \texttt{R1} &= 03 \otimes S(\texttt{Rd}) \oplus 01 \otimes S(\texttt{Rr}) \oplus \texttt{R1} \end{aligned}$$

Our approach is similar to the ones of [26] and [180] in that it tries to pack as many operations as possible into a single instruction. We have already shown in Chapter 7 that slight modifications can lead to a considerable increase in implementation flexibility. Therefore, we also propose a lightweight variant of the `aesenc(x)` instruction, which can be used in the final round of AES encryption as well as in the key expansion. The functionality of this instruction, which we denote by `aessbox`, is shown in Figure 9.3.

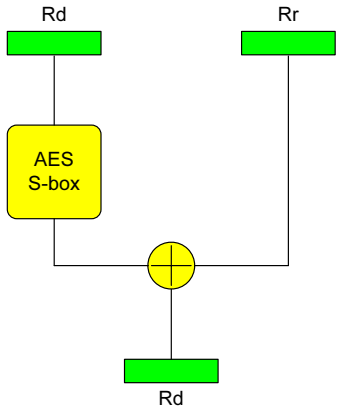

Figure 9.3: AES extension for the final encryption round and the key expansion.

The `aessbox` instruction adheres to the *two-input, one-output format*, which is common to most of the arithmetic and logic instructions of the AVR architecture, e.g., integer addition `add` and bitwise exclusive-or `eor`:

```
aessbox Rd, Rr
```

One of the two input registers (namely `Rd`) is also the destination register of the instruction, while the second input register (`Rr`) can be chosen freely.

For our proposed `aessbox` instruction, the value from register `Rd` is substituted according to the AES S-box and XORed to the value from register `Rr`:

$$\texttt{Rd} \quad = \quad S(\texttt{Rd}) \oplus \texttt{Rr}$$

9.3.2 Support for AES Decryption

Most common modes of operations of block ciphers are defined with the sole use of the according encryption function, e.g., the CTR mode for confidentiality and the CBC-MAC variants for authentication. However, in some situations the decryption function of the block cipher might be of use, e.g., when CBC encryption mode is preferred over CTR mode. For this case we also propose instruction set support for AES decryption, additionally motivated by the following reasons:

- Decryption support can be seamlessly integrated with encryption support with little extra hardware cost.
- With these extensions, decryption speed can be made equal to that of encryption, opening up additional options for more flexible protocol implementations.

Similarly to encryption, decryption support consists of the two instruction variants `aesdec(1)` and `aesdec(2)` with the following format:

```
aesdec(1) Rd, Rr
aesdec(2) Rd, Rr
```

Each instruction variant conforms to one of the two distinct quadrants of the InvMixColumns constant matrix (cf. Figure 9.4). Another necessary change is the use of the inverse S-box.

$$\begin{array}{c} \\ Q1 \\ \\ Q3 \end{array} \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{array}{c} \\ Q2 \\ \\ Q4 \end{array}$$

Figure 9.4: Quadrants of the InvMixColumns matrix.

Decryption support incurs a slight complication of the implementation in regard to the AddRoundKey transformation. For the `aesenc(x)` instructions, the final XOR stage (cf. Figure 9.3) performs both the AddRoundKey transformation as well as a combination of intermediate values in order to yield the State bytes at the end of the round. In contrast, the `aesdec(x)` instructions require the AddRoundKey transformation at a different stage (namely after the

inverse S-boxes), due to the slight change of order of the inverse round transformations in the AES decryption [185]. One possible solution is the introduction of a conditional XOR stage after the inverse S-boxes for AddRoundKey and another such stage at the end for the combination of intermediate results. The `aesdec(1)` instruction can then make use of the first stage and bypass the second stage, whereas `aesdec(2)` can do the opposite. By sticking to a fixed order of `aesdec(1)` and `aesdec(2)` instructions, decryption can be implemented correctly. The functionality of the `aesdec(x)` instruction variants is depicted in Figure 9.5.

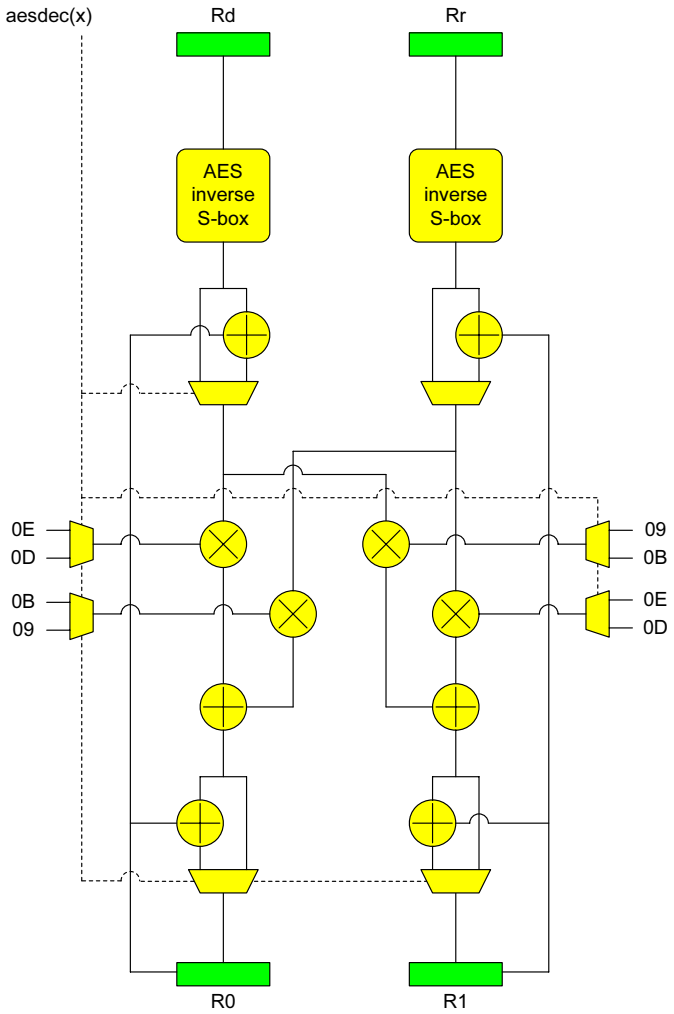

Figure 9.5: AES extensions for a normal decryption round.

The instruction `aesdec(1)` incorporates the AddRoundKey functionality directly after the S-box. In this way, the instruction performs the following oper-

ations (where S_{inv} stands for the inverse S-box):

$$\begin{aligned} \texttt{R0} &= 0\text{E} \otimes (S_{inv}(\texttt{Rd}) \oplus \texttt{R0}) \oplus 0\text{B} \otimes (S_{inv}(\texttt{Rr}) \oplus \texttt{R1}) \\ \texttt{R1} &= 09 \otimes (S_{inv}(\texttt{Rd}) \oplus \texttt{R0}) \oplus 0\text{E} \otimes (S_{inv}(\texttt{Rr}) \oplus \texttt{R1}) \end{aligned}$$

The `aesdec(2)` instruction has again a similar structure as the `aesenc(x)` instructions, just with different coefficients:

$$\begin{aligned} \texttt{R0} &= 0\text{D} \otimes S_{inv}(\texttt{Rd}) \oplus 09 \otimes S_{inv}(\texttt{Rr}) \oplus \texttt{R0} \\ \texttt{R1} &= 0\text{B} \otimes S_{inv}(\texttt{Rd}) \oplus 0\text{D} \otimes S_{inv}(\texttt{Rr}) \oplus \texttt{R1} \end{aligned}$$

For the last round, we propose an instruction `aesinvsbox` with this format:

```
aesinvsbox Rd, Rr
```

This instruction is similar to `aessbox` for encryption. Its functionality is shown in Figure 9.6. The only difference is the use of the inverse S-box in the case of decryption:

$$\texttt{Rd} = S_{inv}(\texttt{Rd}) \oplus \texttt{Rr}$$

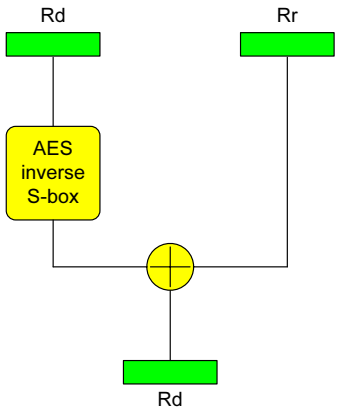

Figure 9.6: AES extension for the final decryption round.

9.3.3 Performance Enhancement and Implementation Flexibility

Our proposed extensions are designed to improve performance using three main strategies. Firstly, the instructions support AES transformations which are not very well catered for by the microcontroller's native instruction set (especially MixColumns and InvMixColumns). Secondly, two State bytes are transformed simultaneously, which effectively "widens" the 8-bit datapath. And finally, several transformations can be executed by a single instruction invocation.

Compared to typical AES coprocessors, our instruction set extensions allow a more flexible application. The custom instructions support all three key sizes of 128, 192, and 256 bits. All modes of operations can be realized seamlessly, as the AES State can be retained in the register file. In contrast, a coprocessor might require to transfer blocks to and from the processor whenever the chosen mode requires operations which are not supported by the coprocessor. The resulting overhead can be detrimental to the overall performance. Another advantage of our extensions is that they support fast implementations of all variants of Rijndael [57], which is a superset of AES and which specifies independent block sizes and key sizes between 128 and 256 bit in 32-bit increments. A potential application of Rijndael is as building block for a cryptographic hash function: By setting Rijndael's block and key size equal, it can be applied in a hashing mode of operation to build a hash function with a hash size equal to the block size.

9.4 Implementation Issues

We now give details on possible hardware implementation options for our proposed extensions and different ways to optimize AES software implementations through utilization of those extensions.

9.4.1 Hardware Implementation of the Proposed Extensions

In this section we outline important implementation issues for the functional units as well as integration issues for the AVR architecture. We will thereby refer to a unified implementation, which is able to provide support for both AES encryption as well as AES decryption as described in Sections 9.3.1 and 9.3.2, respectively.

One important aspect is the support for both the AES S-box and its inverse. In the literature, there have been several proposals for S-box hardware implementations targeting low area, high speed or low power consumption. A comparison of the state-of-the-art regarding their implementation characteristics in standard-cell technology will be given in Chapter 10. An implementation offering a mix of small size and relatively good speed is the design of Canright [41].

The functional part for MixColumns and InvMixColumns demands multiplication with constants in GF(2^8) under a fixed reduction polynomial [185]. These multiplications are rather easy to implement, as the characteristic two of the finite field allows for addition without carry. This is a very desirable property which makes GF(2^m) multipliers generally much faster than their integer counterparts.

Several implementation options are available to realize the GF(2^8) constant multipliers required by our proposed extensions. The smallest solution would be to integrate fixed multipliers similar to those used by Wolkerstorfer in [258]. Wolkerstorfer's approach reuses the results for MixColumns to perform InvMixColumns, thus keeping the overall size of the multipliers small.

In another approach, Elbirt proposed to realize the multipliers in a flexible fashion by providing the possibility to configure the multiplication constants [67]. Elbirt's approach supports AES as well as other implementations in need of fast GF(2^m) multiplication with constants. Naturally, this flexibility has to be bought with an increased demand in hardware. Moreover, the multipliers of Elbirt's solution have to be configured for the specific constants and the reduction polynomial at hand, before they can be used.

The highest degree of flexibility is offered by fully-fledged GF(2^8) multipliers which can vary both multiplier and multiplicand at runtime without configuration overhead. Eberle et al. have proposed to integrate an (8×8)-bit multiplier and a multiply-accumulate unit for binary polynomials in an AVR microcontroller to accelerate Elliptic Curve Cryptography (ECC) over binary extension fields [65]. We have already demonstrated similar synergies for 32-bit architectures in Chapters 5 and 8. Although this variant would be the most costly option in terms of hardware, the increased flexibility and potential support of both symmetric and asymmetric cryptography could make the integration of such multipliers a worthwhile solution for 8-bit architectures.

9.4.2 AES Software Implementation Using the Proposed Extensions

In order to check the benefits of the proposed extensions and to have a base for performance estimations, we have implemented AES-128 encryption and decryption in AVR assembly. We have tried to make the best use of the vast amount of 32 general-purpose registers offered by the architecture in order to keep costly memory accesses at an absolute minimum. In our implementation, the 16-byte AES State is kept in 16 registers at all times and an on-the-fly key expansion is used to preserve key agility. Three of the four 32-bit words of the current round key are also kept in 12 additional registers and only a single round key word has to be held in memory. From the remaining four registers, two (namely `R0` and `R1`) are used to receive the result of `aesenc(x)` or `aesdec(x)` instructions and the other two registers are necessary to hold temporary values during round transformation.

A round function is called to perform the four round transformations on the

State and to generate the subsequent round key. The transformations are performed in-place on the 16 registers holding the State, i.e., all State columns are written back to the same four registers from which they were originally loaded. The ShiftRows function is not performed explicitly on this "register State", but it is only taken into account by an appropriate selection of registers in the round function. As a consequence, a specific State column is contained in a different set of registers after each invocation of the round function. Consequently, we require several different round functions which load the State bytes from the correct registers in conformance to the current layout of the State. Fortunately, the layout of the State reverts back to its original form after four invocations of ShiftRows. This property is illustrated in Figure 9.7, where the four State columns are marked in different shades. Hence, it is sufficient to have four variants of the round function.

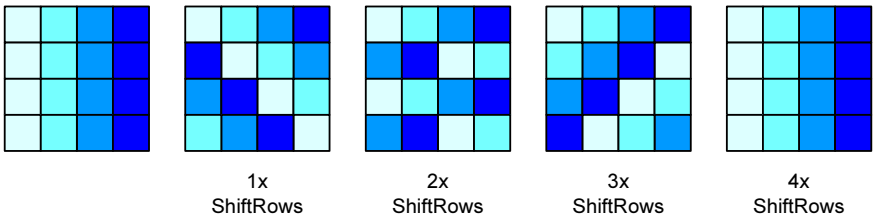

Figure 9.7: Change of AES State layout through ShiftRows for in-place storage.

The assembly code performing all four round transformations on a single State column is shown in Figure 9.8. The update of the first round key word is shown in Figure 9.9.

The main function is responsible for saving the 32 registers onto the stack at entry. Moreover, the function has to load the AES State and cipher key into the corresponding registers. After nine calls to the appropriate round functions, the final round is performed directly by the main function. At the end, the ciphertext is stored to memory and the registers are restored from stack prior to return.

9.5 Performance Analysis

This section gives figures on implementation cost of the proposed instruction set extensions, the performance of our optimized AES implementation and its demands in terms of program memory and working memory.

9.5.1 Hardware Cost

In order to determine the hardware cost for the proposed extensions, we have implemented a functional unit capable of supporting all six custom instructions for AES encryption and decryption. For the AES S-boxes we used the approach of Canright [41]. We included a pipeline stage in the functional unit to adapt

```
; State column in R6, R11, R16, R5
; Round key word in R22-R25
; New State column is written over old column

; Calculate upper half of new column
movw R0, R22          ; Move two round key bytes into R0-R1
aesenc(1) R6, R11     ; ShiftR., SubB., MixC. & AddRK
aesenc(2) R16, R5     ; ShiftR., SubB., MixC.
movw R30, R0          ; Half column into temporary regs R30-R31

; Calculate lower half of new column
movw R0, R24          ; Move rest of round key into R0-R1
aesenc(2) R6, R11     ; ShiftR., SubB., MixC. & AddRK
aesenc(1) R16, R5     ; ShiftR., SubB., MixC.

; Store new column over old column
mov R6, R30
mov R11, R31
mov R16, R0
mov R5, R1
```

Figure 9.8: Round transformations for a single State column.

```
; First word of old round key in R18-R21
; Last word of old round key in R26-R29
; Rcon located in R30
; New first round key word written over old word

eor R18, R30          ; Add Rcon
aessbox R18, R27      ; RotWord, SubWord, Add to old byte
aessbox R19, R28      ; RotWord, SubWord, Add to old byte
aessbox R20, R29      ; RotWord, SubWord, Add to old byte
aessbox R21, R26      ; RotWord, SubWord, Add to old byte
```

Figure 9.9: Update of the first round key word.

it to the read-write capabilities of the register file of existing AVR microcontrollers [15].

Our functional unit is depicted in Figure 9.10. The different sections conforming to different AES transformations are highlighted. The dashed line represents configuration information which determines the functionality in dependence on the actual instruction. The S-boxes are used in forward direction for the instructions for encryption (`aesenc(x)` and `aessbox`) and in inverse direction for the instructions for decryption (`aesdec(x)` and `aesinvsbox`). The multiplex-

ers in the AddRoundKey sections select the left input for the `aesenc(x)` and `aesdec(2)` instructions and the right input for the `aesdec(1)` instruction. The four multiplexers in the (Inv)MixColumns section are driven by the same control signal, i.e., they always select the same input position at a given time. `aesenc(1)` selects the first (top) inputs of the multiplexers, `aesenc(2)` the second ones, `aesdec(1)` the third ones, and `aesdec(2)` the last (bottom) ones. The result of the `aesenc(x)` and `aesdec(x)` instructions is delivered into `R0` and `R1`, while the result of the `aessbox` and `aesinvsbox` instructions appears at the output for `Rd`.

The functional unit from Figure 9.10 can be improved when it can be supplied with the information on the current execution cycle of an `aesenc(x)` or `aesdec(x)` instruction, i.e., whether it is the first or the second execution cycle. In this case, a single AES S-box is sufficient to perform the required operations. This alternative implementation is shown in Figure 9.11 with the changes shaded. In comparison with the implementation from Figure 9.10, the pipeline register at (1) has been moved above the S-box to (2). Moreover, an additional multiplexer (3) is used to select the input to the S-box. In the first clock cycle of any `aesenc(x)` or `aesdec(x)` instruction, `Rd` is selected. In the second clock cycle, `Rr` will be used as input to the S-box. Also note that the implementation of the `aessbox` and `aesinvsbox` instructions does not change.

The GF(2^8) multipliers of the functional units have been hardwired for the constants used in MixColumns and InvMixColumns. Thereby, the two multipliers for a byte have been implemented jointly. A byte b is multiplied with the powers of two, yielding four intermediate results (b, $\{2\}b$, $\{4\}b$, and $\{8\}b$). Depending on the instruction, these intermediate results are added to yield the required multiplication results. Figure 9.12 shows the implementation for the first byte (i.e., the upper two multipliers transforming the byte from `Rd` in Figure 9.10), while Figure 9.13 contains the respective implementation for the second byte.

We have implemented both variants of our functional unit (with two and one S-boxes, respectively) using a 0.35 µm CMOS standard cell library from austriamicrosystems. For the two-S-box variant, the synthesized circuit has a size of 1,126 gates with a critical path of 36.7 ns. The variant with a single S-box required only 791 gates and has a critical path of 46.4 ns. Note that we have optimized the synthesis result towards minimal area, just setting a maximal critical path of 50 ns to match the 20 MHz maximal clock frequency of state-of-the-art AVR microcontrollers. The speed of the circuit could easily be increased by trading off area efficiency.

The smallest AES coprocessor reported in literature so far is by Feldhofer et al. with a size of about 3,400 gates [70]. Our proposed extensions have less than a fourth of this size.

9.5.2 Performance

Based on our optimized assembly implementation, we have estimated the number of clock cycles for a single AES-128 encryption and decryption (including the

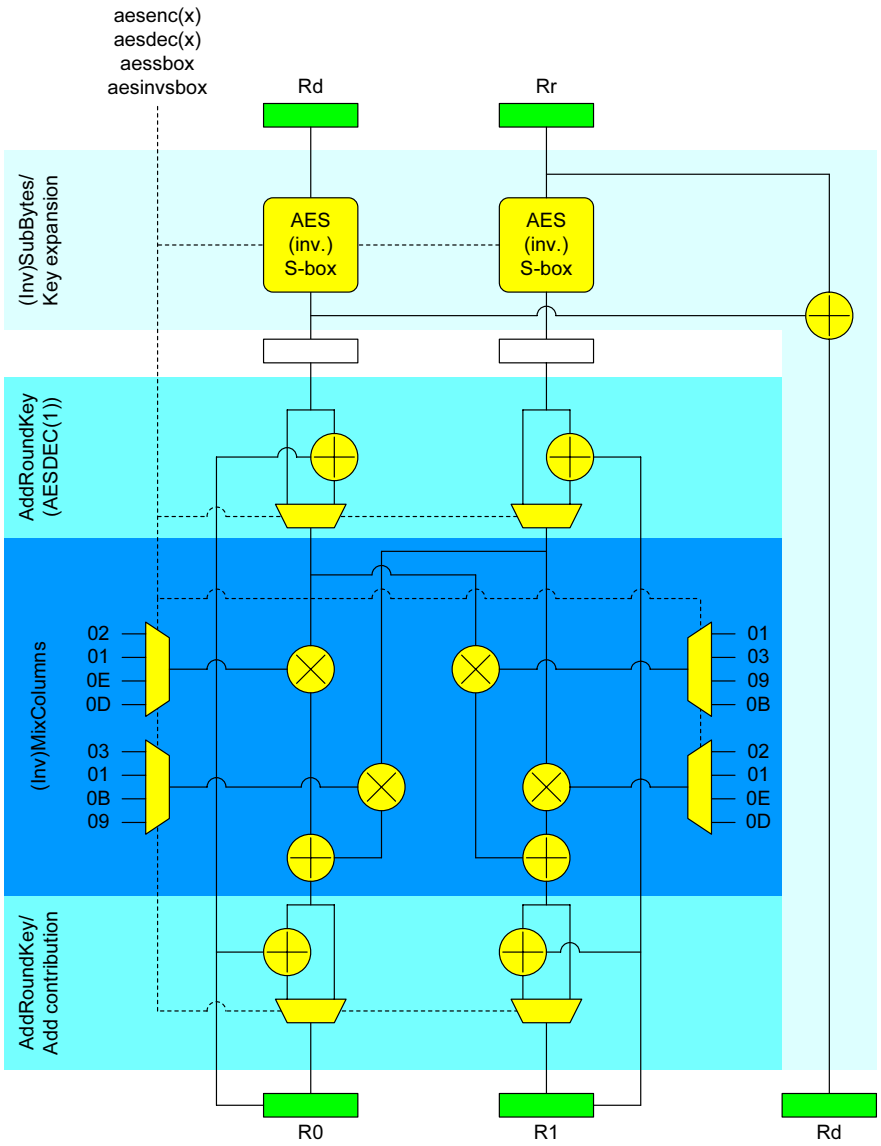

Figure 9.10: Implementation of the functional unit for supporting the AES extensions.

complete on-the-fly key expansion). Thanks to the simple and deterministic structure of AVR microcontrollers, this estimation can be done with a high level of accuracy. For all our custom instructions we have assumed a cycle count of 2, which we deem to be realistic for implementation. Executing a single round function (either for encryption or decryption) requires 106 clock cycles. With

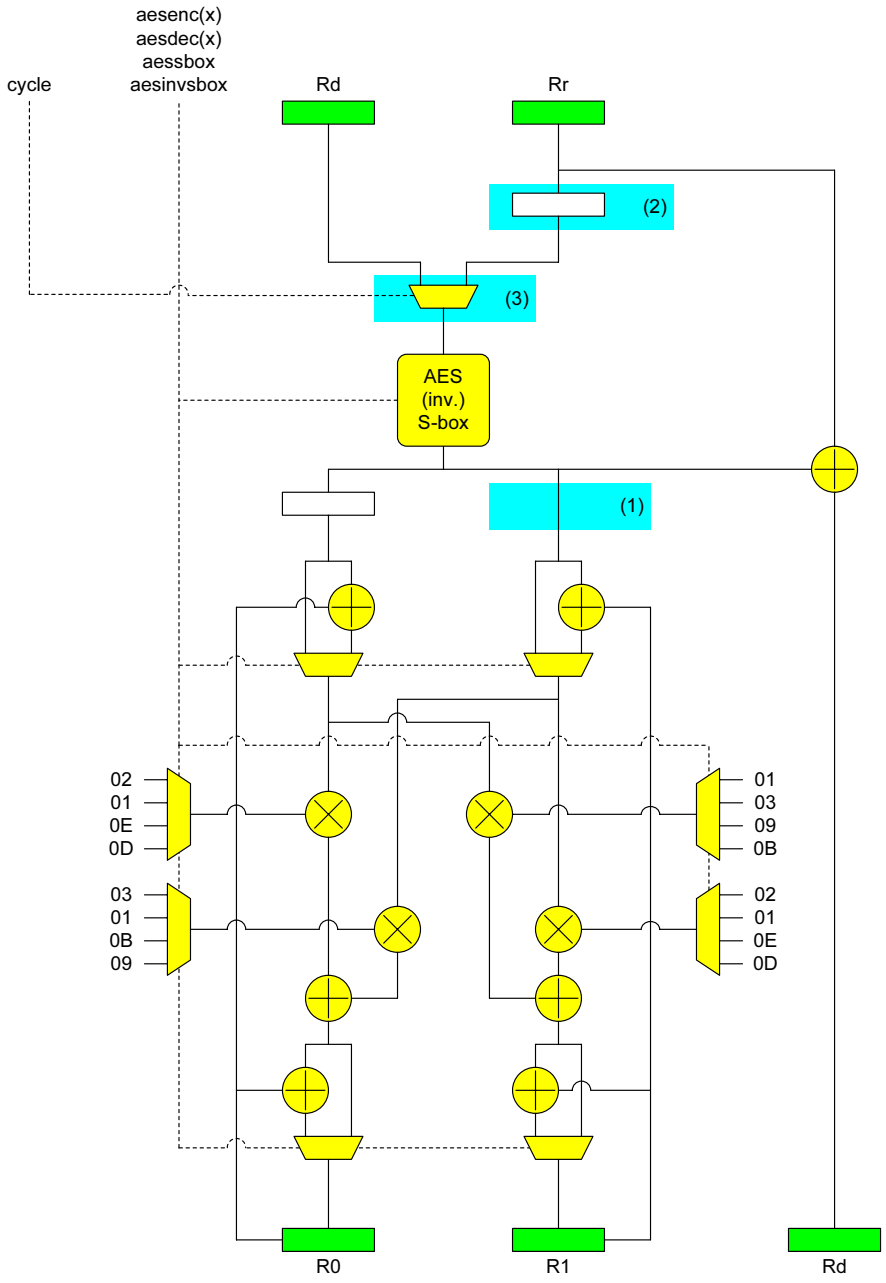

Figure 9.11: Implementation variant of the functional unit with a single AES S-box.

the overhead from the main function, the cycle count for encryption of a 128-bit block amounts to 1,262 (including the loading of the plaintext from memory and

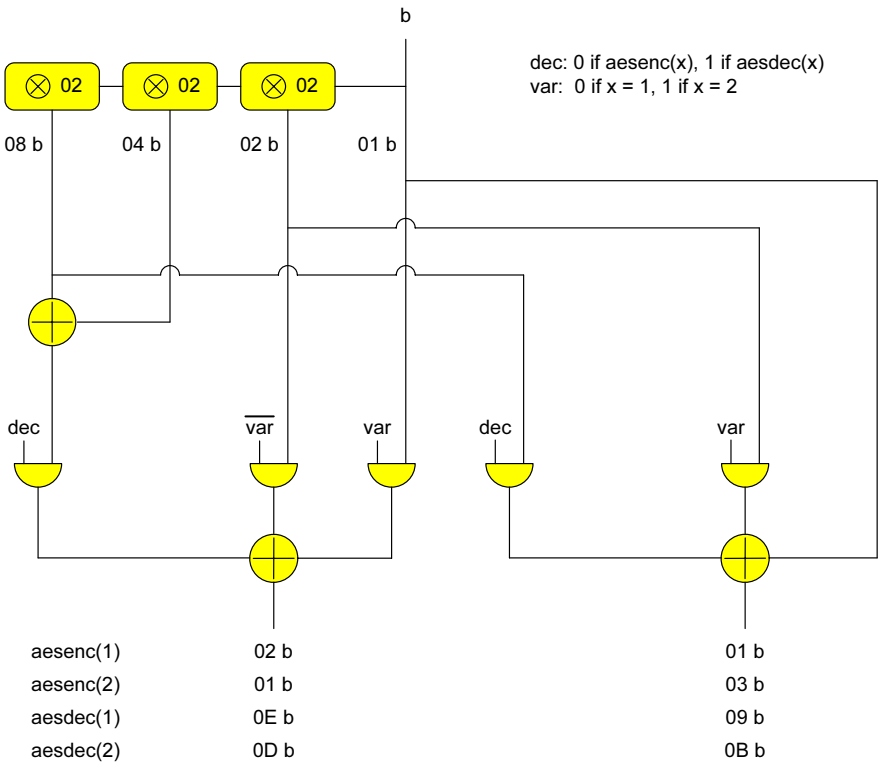

Figure 9.12: Implementation of the finite field constant multipliers for the first byte.

the storing of the ciphertext back to memory). Thanks to the symmetry of the extensions, AES decryption can be done equally fast in 1,263 cycles.

We compare our performance to that of an assembly-optimized software implementation of AES for the AVR architecture reported in [203]. It requires 3,766 cycles for encryption and 4,558 cycles for decryption, where the overhead for decryption mainly stems from the more complicated InvMixColumns transformation. The speedup factors for our implementation are therefore about 3 and 3.6, respectively.

The coprocessor of Feldhofer et al. has a performance roughly equivalent to our extensions with a cycle count of 1,032 for encryption and 1,165 for decryption of a single block [70].

9.5.3 Code Size and RAM Requirements

Our assembly implementation of encryption and decryption requires 1,708 bytes of code memory. This size can be further reduced with an explicit ShiftRows at the end of each round function (20 additional `mov` instructions requiring 20 cycles). In this case, a single round function for encryption and decryption

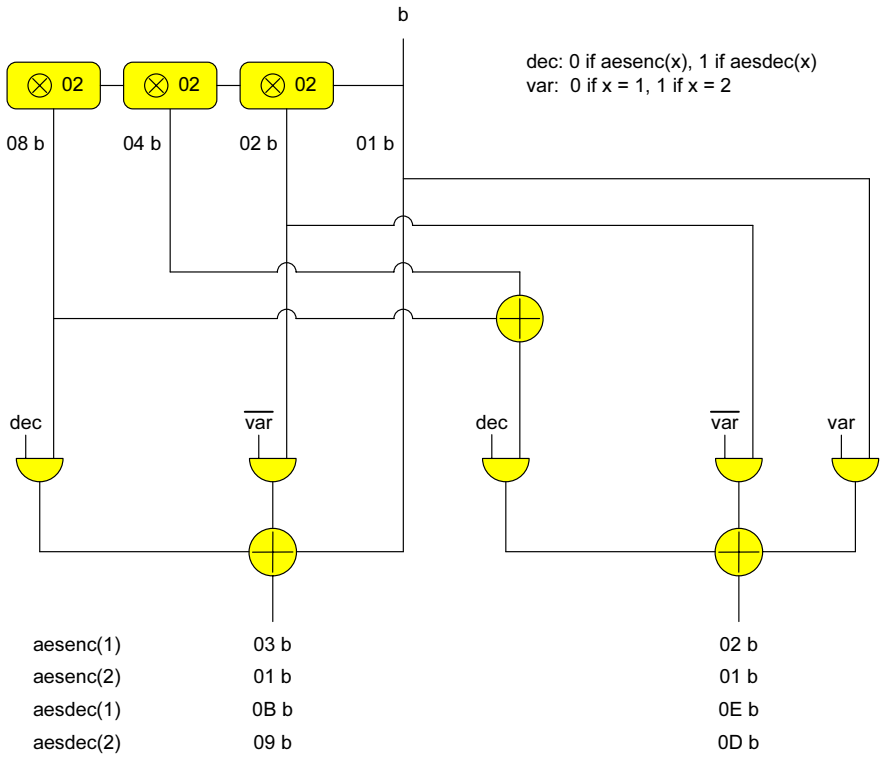

Figure 9.13: Implementation of the finite field constant multipliers for the second byte.

would suffice, which brings the overall code size down to 840 bytes. However, the number of cycles per encryption and decryption would increase by 180.

In terms of RAM, our implementation requires only four bytes of extra memory in addition to the use of the general-purpose registers. Note that we are not considering the memory from which we load the plaintext at the start of encryption and where we store the ciphertext to at the end.

9.5.4 Summary of Comparison

Table 9.1 summarizes our performance figures with those of the optimized software implementation from [203], the custom AES microcontroller from [48], Feldhofer et al.'s tiny AES coprocessor [70], and the coprocessor in the new XMEGA devices [19]. We have included both of our implementation variants for maximal speed (fast) and minimal code size (compact), cf. Section 9.5.3. The cycle count refers to AES-128 encryption and decryption of a single 16-byte block. The code size refers to an implementation which can support both encryption and decryption.

Our proposed solution is considerably faster and requires less code size than

Table 9.1: AES performance characteristics in comparison to related work.

Implementation	Encryption cycles	Decryption cycles	Code size bytes	HW cost GE
AVR software [203]	3,766	4,558	3,410	none
AES coprocessor [70]	1,032	1,165	n/a	3,400
AES coprocessor [19]	375 [a]	375 [a]	n/a	n/a
AES microcontroller [48]	2,695 [b]	2,944 [b]	918 [c]	n/a
This work (fast)	**1,259**	**1,259** [d]	**1,708**	**791**
This work (compact)	**1,442**	**1,443** [d]	**840**	**791**

[a] Excluding cost for transferring key and data.
[b] Excluding cost for precomputed key schedule (2,167 cycles).
[c] Total size for encryption, decryption and key expansion.
[d] Last round key supplied to decryption function.

the pure-software approach. Nevertheless, the flexibility of the software solution is fully retained. Compared to the coprocessor approach, our solution offers similar performance at much smaller hardware overhead. The AES microcontroller has a similar code size as our compact implementation, but is significantly slower. The AES coprocessor in the new XMEGA devices [19] is considerable faster than our solution. Unfortunately, there are no figures on its hardware size available. But it can be assumed that this coprocessor is significantly bigger than the one of Feldhofer et al. [70]. In any case, both coprocessor solutions are limited to a key size of 128-bit, while our approach supports all possible key sizes and is also more flexible in regard to different modes of operation.

9.6 Summary and Conclusions

In this chapter we have presented a set of small and simple AES instruction set extensions for the 8-bit AVR architecture. We have demonstrated the benefits of these extensions with an optimized AES encryption implementation, which is about three times faster than an optimized assembly implementation using native AVR instructions. The speedup for decryption is even higher, amounting to a factor of about 3.6. As an additional benefit, code size is small and RAM requirements are very low. The hardware cost of our extensions ranges around 800 gates. Compared to the smallest AES coprocessor reported so far, our extensions deliver similar performance at less than a fourth of the hardware cost. All in all, our extensions provide a very good tradeoff between hardware overhead, performance gain and implementation flexibility and position themselves at a favorable section of the design space.

10

Design Options for the AES S-box in Standard-Cell Technology

Cryptographic substitution boxes (S-boxes) are an integral part of modern block ciphers like the Advanced Encryption Standard (AES). There exists a rich literature devoted to the efficient implementation of cryptographic S-boxes, whereby hardware designs for FPGAs and standard cells received particular attention. In this chapter we present a comprehensive study of different standard-cell implementations of the AES S-box with respect to timing (i.e., critical path), silicon area, power consumption, and combinations of these cost metrics. We examined implementations which exploit the mathematical properties of the AES S-box, constructions based on hardware look-up tables, and dedicated low-power solutions. Our results show that the timing, area, and power-consumption properties of the different S-box realizations can vary by up to almost an order of magnitude. In terms of area and area-delay product, the best choice are implementations which calculate the S-box output. On the other hand, the hardware look-up solutions are characterized by the shortest critical path. The dedicated low-power implementations do not only reduce power consumption by a large degree, but they also show good timing properties and offer the best power-delay and power-area product, respectively.

The work described in this chapter provides valuable insights for designers of a large range of different AES implementations which feature hardware support for the S-box lookup. This is particularly interesting for high-speed hardware designs, which typically feature a large number of hardware S-boxes. These S-boxes make up a considerable fraction of the total silicon area and contribute a significant portion to the total power consumption. Any improvements of the S-boxes can have dramatic improvements to the overall efficiency of the design.

Similarly, our results help in designing and implementing optimized functional units which are capable of supporting the custom instructions described in Chapters 6, 7, and 9.

10.1 Hardware Implementation Aspects of AES

The AES is a flexible algorithm well suited for implementation in hardware. A multitude of hardware architectures are possible, which allows for optimization towards different requirements, ranging from high performance to low power consumption and small silicon area. There exists a considerable literature devoted to efficient hardware implementation of the AES [49, 70, 117, 200, 209, 265]. One possibility to categorize AES architectures is the size of the data path; the most common values are 8, 32, and 128 bits. A second criterion is whether the AES rounds are unrolled or not. Thirdly, AES hardware implementations differ in the number of pipeline stages.

The width of the datapath determines the main characteristics (i.e., performance, area, power consumption) of an AES implementation. Since the AES is byte-oriented, an 8-bit architecture with a single S-box is the natural choice for applications where small area and low power dissipation are crucial, e.g., smart cards or RFID tags. At the other end of the spectrum are 128-bit architectures containing 16 S-boxes to compute the SubBytes function of a 128-bit data block in one pass. Due to this massive parallelism, 128-bit architectures can reach high throughput rates at the expense of large silicon area. 32-bit architectures with four S-boxes constitute a good compromise between the two aforementioned extremes; they allow for much higher performance than 8-bit architectures but demand only a fraction of the area of 128-bit implementations.

10.2 Implementation Strategies for the AES S-box

All AES architectures sketched in Section 10.1 have in common that the SubBytes transformation occupies a significant portion of the overall silicon area. The size of SubBytes is, in turn, determined by the number of S-boxes and their concrete implementation. Various implementation options for the AES S-box have been investigated in the recent past, which has led to an abundant literature [27, 41, 163, 172, 178, 209, 259].

The SubBytes transformation substitutes all 16 bytes of the State independently using the S-box. Furthermore, the S-box is also used in the AES key expansion. In software, the S-box is typically realized in form of a look-up table since the inversion in the finite field $\mathbb{F}_{2^8}$ cannot be calculated efficiently on general-purpose processors. In hardware, on the other hand, the implementation of the S-box is directed by the desired trade-off between area, delay, and power consumption. The most obvious implementation approach for the S-box takes the form of hardware look-up tables. However, since encryption and decryption

require different tables, and each table contains 2048 bits, the overall hardware cost of this approach is relatively high.

An implementation option related to standard cells is the usage of ROM compilers to produce hardware macros. For the technology that we used, a sufficiently large ROM would require a considerable amount of silicon area. The critical path delay would be similar to a hardware look-up approach, but the power consumption of generated ROMs is about two to three orders of magnitude higher (exact performance figures for ROMs were unfortunately not accessible for our used technology). Therefore, we did not consider the implementation of the S-box as ROM.

More advanced approaches calculate the S-box function in hardware using its arithmetic properties. The focus of such implementations is the efficient realization of the inversion in $\mathbb{F}_{2^8}$, which can be achieved by decomposing the finite field into the sub-fields $\mathbb{F}_{2^4}$ and $\mathbb{F}_{2^2}$. An inversion in a finite field of characteristic 2 can be carried out in different ways, depending on the basis which is used to represent the field elements [158]. The two most common types of bases for $\mathbb{F}_{2^m}$ are the *polynomial basis* and the *normal basis*. A polynomial basis is a basis of the form $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ where α is a root of an irreducible polynomial $p(x)$ of degree m with coefficients from $\mathbb{F}_2$. On the other hand, a normal basis can be found by selecting a field element $\beta \in \mathbb{F}_{2^m}$ such that the elements of the set $\{\beta, \beta^2, \beta^4, \ldots, \beta^{2^{m-1}}\}$ are linearly independent.

Another approach for implementing the AES S-box was proposed by Bertoni et al. in [27]. By using an intermediate one-hot encoding of the input, arbitrary logic functions (including cryptographic S-boxes) can be realized with minimal power consumption. The main drawback of this approach is that it demands a relatively large silicon area.

10.3 Examined Implementations

All AES S-box implementations analyzed in this chapter can perform forward and inverse byte substitution for encryption and decryption, respectively. We implemented the S-boxes either from scratch or obtained them from the authors of the respective publications[1]. The examined implementations consist solely of combinatorial logic, i.e., no pipeline stages have been inserted. In the following we describe a total of eight different implementation variants of the AES S-box which can be grouped into three basic categories: look-up implementations, calculating implementations, and low-power implementations. We use figures to illustrate the most important design concepts. In these figures, S_{in} and S_{out} denote the S-box input and output, respectively, while $enc/\overline{dec}$ is the control signal to switch between the forward substitution for encryption and the inverse substitution for decryption.

The simplest design in our comparison is a straight-forward implementation of a hardware look-up table. The synthesizer transforms the behavioral

[1]We would like to thank Johannes Wolkerstorfer and David Canright for providing their HDL source code for several AES S-box implementation options.

description of the look-up table into a mass of unstructured standard cells (cf. Figure 10.1). This approach will be denoted as **hw-lut**.

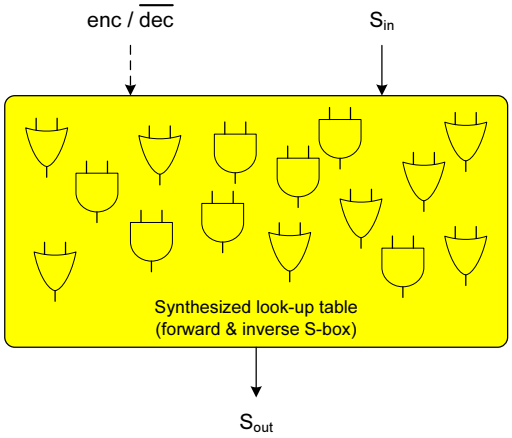

Figure 10.1: S-box as hardware look-up table.

Implementations which calculate the S-box transformation in hardware have been first proposed by Wolkerstorfer et al. [259] and Satoh et al. [209]. In order to give a general idea, the former approach is depicted in Figure 10.2. Wolkerstorfer et al.'s solution decomposes the elements of $\mathbb{F}_{2^8}$ into polynomials over the sub-field $\mathbb{F}_{2^4}$ and performs inversion there. Our implementation of this solution is denoted as **wolkerstorfer**. Satoh et al.'s solution decomposes the field elements further into polynomials over the sub-field $\mathbb{F}_{2^2}$, where inversion is a trivial swap of the lower and higher bit of the representation. This implementation is called **satoh** in the following. Both of these approaches represent the field elements by using a polynomial basis. Canright improved the calculation of the S-box by switching the representation to a normal basis [41]. Like in Satoh's solution, the finite field element's representation is mapped to a polynomial over the sub-field $\mathbb{F}_{2^2}$. This approach will be denoted as **canright**.

A compromise between hardware look-up and calculation has also been examined. In this implementation (denoted as **hybrid-lut**) only the inversion in $\mathbb{F}_{2^8}$ is implemented as look-up table. As this inversion is used for both encryption and decryption, the size of the look-up table is halved in relation to the **hw-lut** approach. The affine and inverse affine transformations are done in logic just as in the calculating implementations of **wolkerstorfer**, **satoh**, and **canright**. The **hybrid-lut** solution is shown in Figure 10.3.

The hardware look-up (**hw-lut**) approach can be modified towards lower power consumption by the use of sub-tables (cf. Figure 10.4). This reduces switching activity in the look-up tables in order to reduce power consumption. We examined solutions with sub-tables of size 16, 32, 64, 128, and 256 bytes, but in this chapter we only cite the results for sub-tables which are 16 bytes in size, which will be denoted as **sub16-lut**).

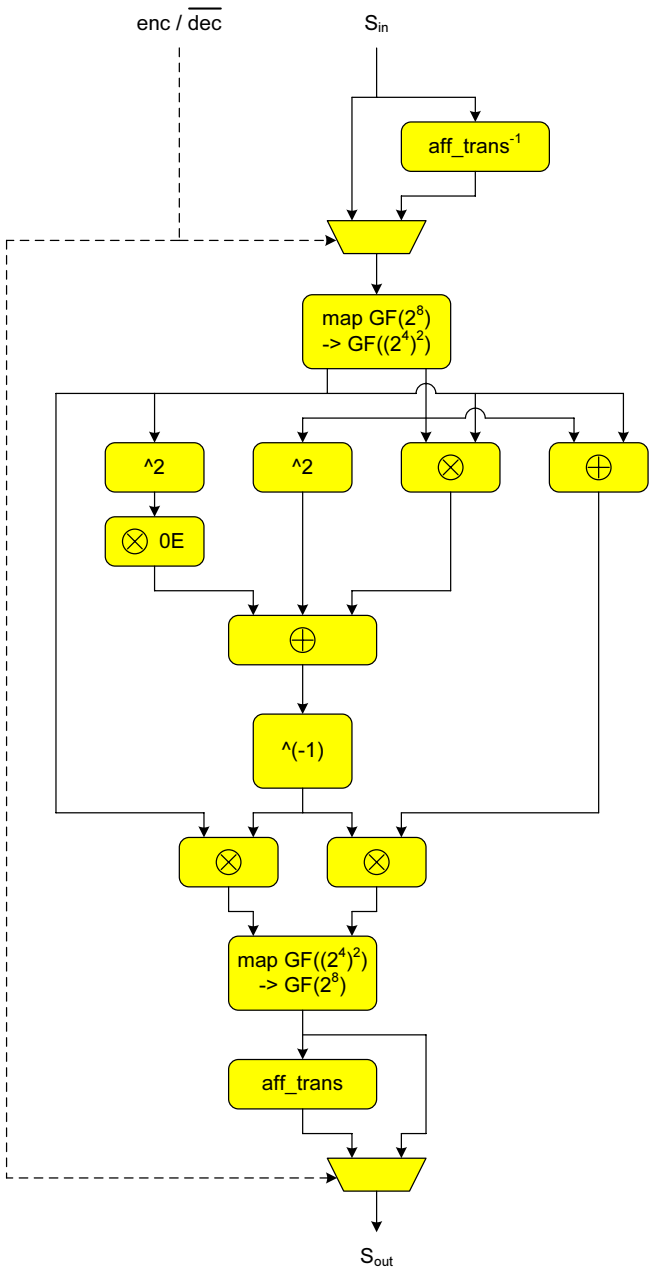

Figure 10.2: S-box calculated according to Wolkerstorfer et al.'s approach [259].

The low-power approach of Bertoni et al. [27] is shown in Figure 10.5. It uses a decode stage to represent the eight bits of the input byte and the control

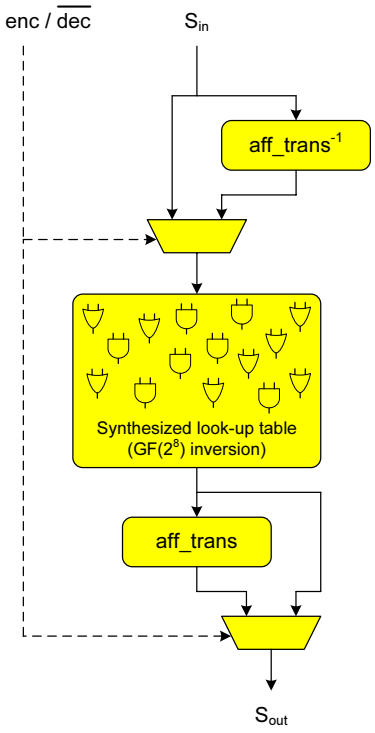

Figure 10.3: S-box as compromise between calculation and hardware look-up (**hybrid-lut**).

bit which selects encryption or decryption into a one-hot encoding on 512 lines. The substitution itself is just a simple rewiring of these lines. As two of the lines always map to the same 8-bit result (one for encryption and one for decryption), these line pairs are combined with a logical OR to result in a one-hot encoding of the result on 256 lines. A subsequent encoder stage transforms this result back to an 8-bit binary value. Due to this decoder-permute-encoder structure, there is only very little signal activity within the circuit at a change of the input, resulting in low power consumption.

Note that the structure of Bertoni et al.'s approach makes it in principle easily possible to introduce pipeline stages. However, it may be necessary to add a large number of additional flip-flops when the pipeline stage is placed between the decoder and encoder, i.e., on the one-hot encoded signal lines. These flip-flops will increase power consumption considerably and can easily mitigate the low-power advantages of this solution. For design scenarios where both power consumption and silicon area are of minor importance, Bertoni et al.'s approach can offer the best opportunity for reaching very high clock frequencies.

We tested two implementations of Bertoni's approach: One implementation uses a decoder with four stages as proposed in the original publication in [27] for

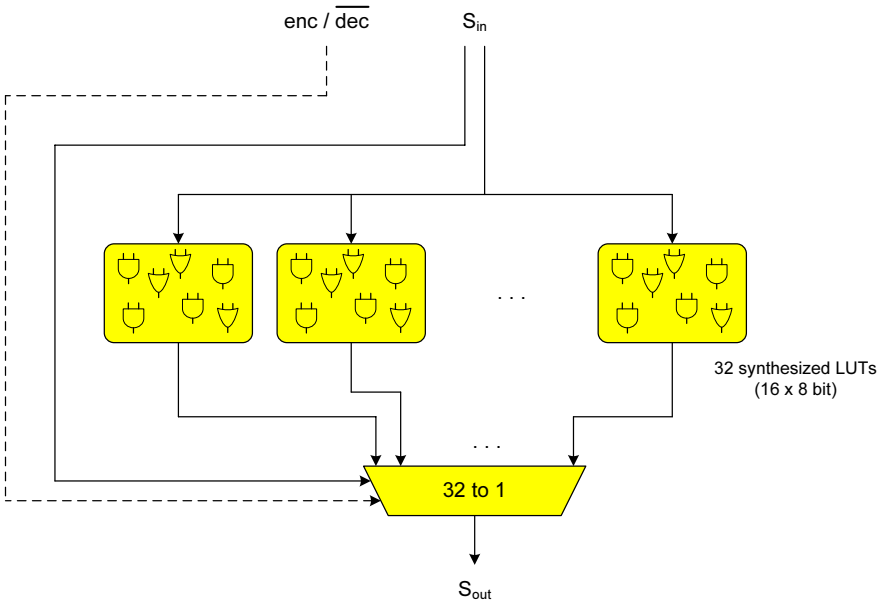

Figure 10.4: S-box implementation with small hardware look-up tables.

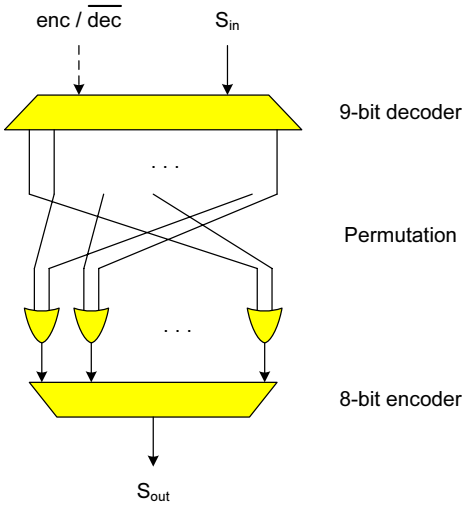

Figure 10.5: S-box following Bertoni et al.'s approach [27].

minimal power consumption (**bertoni**). The second implementation, denoted as **bertoni-2stg**, uses a different decoder structure with only two stages in order to reduce the critical path of the circuit.

In the remainder of this chapter we will refer to **wolkerstorfer**, **satoh**, and **canright** as *calculating implementations*. We will denote **hw-lut** and **hybrid-**

lut as *look-up implementations*, and **sub16-lut**, **bertoni**, and **bertoni-2stg** as *low-power implementations*.

10.4 Design Flow and Evaluation Methodology

In our earlier work in [232], we used a 0.35 µm standard cell library from austriamicrosystems. In contrast, all results in this chapter were obtained with the VST250 standard cells from Virtual Silicon. These standard cells are built upon the 0.25 µm process technology L250 of UMC, which provides one poly-silicon layer and five metal layers. The nominal supply voltage of the VST250 cell library is 2.5 V.

We implemented the eight S-box designs described in Section 10.3 in VHDL according to the specifications in the respective papers. In order to ensure a fair comparison and a common interface for all implementations, we provided the input and output of each S-box with 8-bit registers. The integration of the registers made it possible to optimize for area and delay during synthesis. The logic synthesis was done using the Physically Knowledgeable Synthesis (PKS) tool from Cadence. We varied the constraints for the delay time (i.e., maximum clock frequency) from the minimum value to a value where the constraints could just be met. The delays given in Table 10.1 are the actual delays of the synthesized circuit. Empty cells in the table indicate that the respective target delay could not be achieved by the synthesizer.

After synthesis, the placement and routing of the standard cells was performed with the Cadence tool First Encounter. We did not include I/O cells into the designs, i.e., we analyzed only the core of the S-boxes consisting of standard cells and the power supply rings. During placement we used an area utilization of 70%. All the figures in Table 10.1 are results from synthesis excluding the clock tree for the input and output registers. After the routing step we integrated the layouts of the standard cells into the design, which gave us the full layout in GDS2 format.

We extracted a Spectre netlist from the layout using Assura RCX, whereby we only considered resistances larger than 1 Ω and capacitances larger than 1 pF. In contrast to our previous work [232], we obtained the power consumption of the different S-box designs through simulation with Synopsys NanoSim. All simulations were performed with BSIM3v3 transistor models characterized for the UMC L250 technology and the built-in NanoSim models for resistors and capacitors. The results of the NanoSim simulations shown in Table 10.1 represent the mean current consumption of the S-boxes at a supply voltage of 2.5 V. We used a clock frequency of 50 MHz (i.e., new input values are applied to the circuit with a period of 20 ns) and simulated all 256 possible input patterns.

Table 10.1: Synthesis results of the eight S-box designs depending on the target delay.

		Target delay (ns)							
Design	**Result**	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00
canright	Act. delay (ns)	–	–	–	4.98	5.00	6.55	6.55	6.55
	Area (GE)	–	–	–	496	400	303	303	303
	Power (µA)	–	–	–	1.78	1.78	1.81	1.81	1.81
satoh	Act. delay (ns)	–	–	–	–	5.93	6.55	6.99	6.99
	Area (GE)	–	–	–	–	438	409	385	385
	Power (µA)	–	–	–	–	2.00	1.73	1.51	1.51
wolkers-torfer	Act. delay (ns)	–	–	–	4.93	5.94	6.48	7.51	7.51
	Area (GE)	–	–	–	625	412	415	392	392
	Power (µA)	–	–	–	1.87	1.97	1.75	1.53	1.53
hw-lut	Act. delay (ns)	1.95	2.91	3.90	4.98	5.88	6.61	6.61	6.61
	Area (GE)	1545	1415	1351	1352	1302	1301	1301	1301
	Power (µA)	1.18	0.97	1.00	0.97	0.93	1.00	1.00	1.00
sub16-lut	Act. delay (ns)	–	2.94	3.92	4.46	4.46	4.46	4.46	4.46
	Area (GE)	–	2040	1979	1957	1957	1957	1957	1957
	Power (µA)	–	0.56	0.53	0.55	0.58	0.58	0.58	0.58
hybrid-lut	Act. delay (ns)	–	2.93	3.92	4.86	5.83	6.49	6.49	6.49
	Area (GE)	–	1222	840	810	799	798	798	798
	Power (µA)	–	1.34	1.02	0.98	0.95	0.98	0.98	0.98
bertoni	Act. delay (ns)	1.86	2.90	3.31	3.31	3.31	3.31	3.31	3.31
	Area (GE)	2016	1433	1399	1399	1399	1399	1399	1399
	Power (µA)	0.42	0.30	0.27	0.27	0.27	0.27	0.27	0.27
bertoni-2stg	Act. delay (ns)	1.98	2.79	3.53	3.26	3.26	3.26	3.26	3.26
	Area (GE)	1941	1446	1436	1421	1421	1421	1421	1421
	Power (µA)	0.42	0.32	0.31	0.33	0.33	0.33	0.33	0.33

10.5 Experimental Results

We synthesized all eight S-box implementations described in Section 10.3 using the design flow outlined in Section 10.4. For each implementation several synthesis runs were performed, whereby we specified different target values for the maximal critical path delay, ranging from 2 ns to 9 ns. Table 10.1 summarizes the actual delay, the area of the synthesized design, and the mean power consumption. We omitted the results of all synthesis runs where the timing constraints were not met, i.e., when the actual delay was higher than the target delay.

In order to analyze the results we created plots of various combinations of metrics. In all the plots we use the same color and marker symbols for each specific S-box implementation. The ordering of names in the legend corresponds to the occurrence of the respective lines in the plot, e.g. the topmost name in the legend is also the topmost line in the plot.

In Figure 10.6 the area of synthesized designs with a specific critical path delay are shown. The area is given in gate equivalents (GE), calculated as total area divided by the size of a 2-input NAND with the lowest driving strength, which is the NAN2D1 standard cell in the library we used.

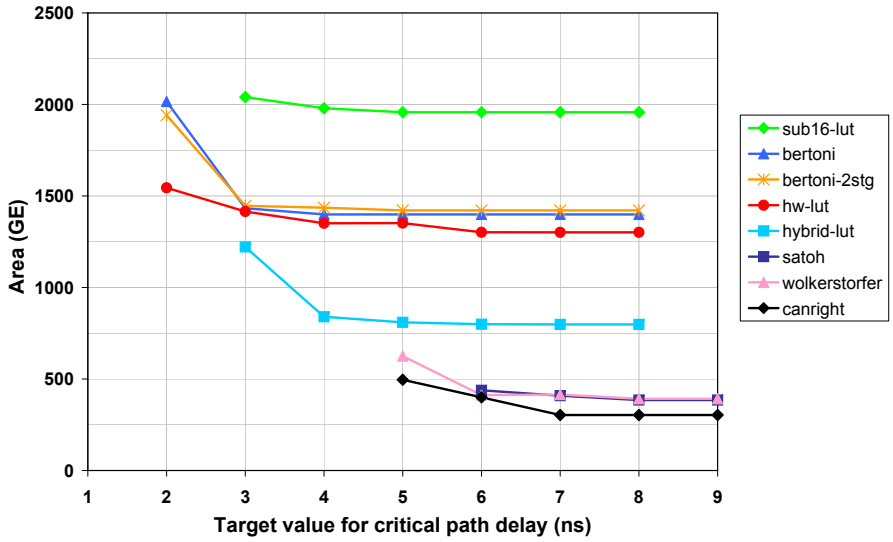

Figure 10.6: Area vs. critical path delay.

Amongst the three calculating implementations (bottom of Figure 10.6), **canright** is clearly the best. It has the smallest size, but suffers from a longer critical path than the hardware look-up implementations and the low-power implementations. The calculating implementations are smaller than the other two approaches because they make use of the algebraic structure of the S-box to implement the substitution. On the other hand, this structure has a relatively long critical path. The shortest critical path can be achieved with **bertoni** but its size is about three times that of **canright**. Look-up implementations neglect the algebraic structure of the S-box and just aim at a straightforward realization of the Boolean equations constituted by the input-output relation. Hence, the synthesizer has a much higher degree of freedom for optimizing the circuit, which allows for a much shorter critical path at the expense of silicon area.

The low-power implementations also neglect the arithmetic properties of the substitution and just implement the Boolean equations of the input-output relation. However, they use a specific structure (decode-permute-encode) to reduce signal activity. Although the critical path is similarly short as for look-up implementations, the one-hot encoding requires more silicon area than the look-up implementations. The **sub16-lut** approach also has a significant area overhead introduced by the address decoding of the sub-tables, which makes it the solution requiring the most silicon area. Moreover, the address decoding logic leads to a longer critical path. As expected, the compromise between hardware look-up and calculation (**hybrid-lut**) lies roughly between **hw-lut** and the calculating implementations in regard to both critical path delay and area.

Figure 10.7 shows the total power consumption in relation to the critical path delay. All power figures have been normalized to the power consumption of

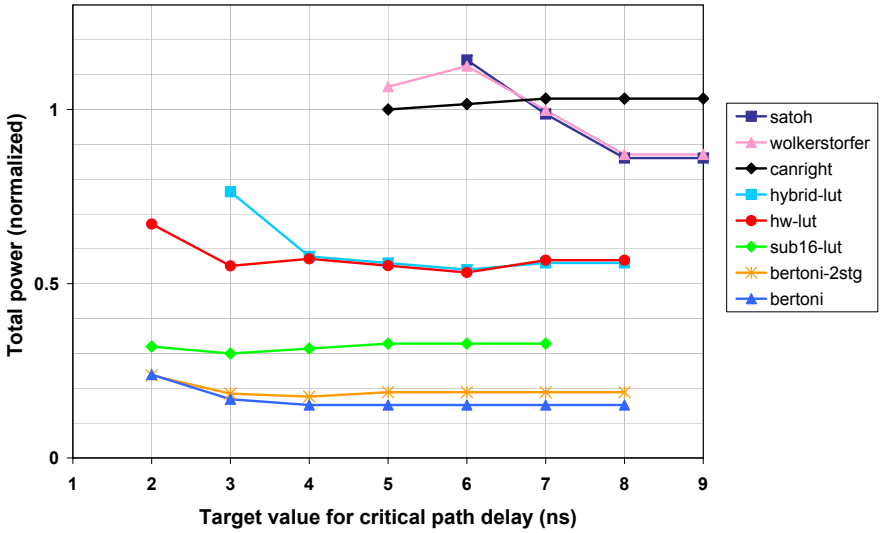

Figure 10.7: Total power consumption vs. critical path delay.

hw-lut at a 5 ns delay. The low-power implementations based on the approach by Bertoni et al. (**bertoni**, **bertoni-2stg**) show the lowest power consumption. The original implementation **bertoni** has the best characteristics while the modified version **bertoni-2stg** is slightly worse. Bertoni et al.'s approach is solely directed towards low power consumption with a minimal level of signal activity in the circuit. Therefore, it is better than the **sub16-lut** approach, which tries to improve a straightforward look-up table implementation (**hw-lut**) with low-power measures. The **sub16-lut** implementation requires almost twice as much power as **bertoni**, while **hw-lut** consumes about three times more power. The **hybrid-lut** approach requires about the same amount of power as **hw-lut**.

The power consumption of the calculating implementations is much higher than that of the low-power and look-up versions. The algebraic evaluation of the S-box function in calculating implementations requires re-computation of all intermediate values even if only a few number of input bits toggle. This behavior entails very high signal activity. In look-up implementations a change of a few input bits affects the calculation of all output bits separately. As some output bits will be left unchanged, the signal activity within this particular path is low and hence limits the power consumption. The most power-efficient variant among the calculating implementations is **canright**, which has less than two times the power consumption of **hw-lut**. The power consumption of **wolkerstorfer** and **satoh** is a little bit higher.

Figure 10.8 shows our results in terms of the power-area product. This metric is particularly relevant for applications which require both small silicon area and low power consumption, e.g., cryptographically enhanced RFID tags or sensor nodes.

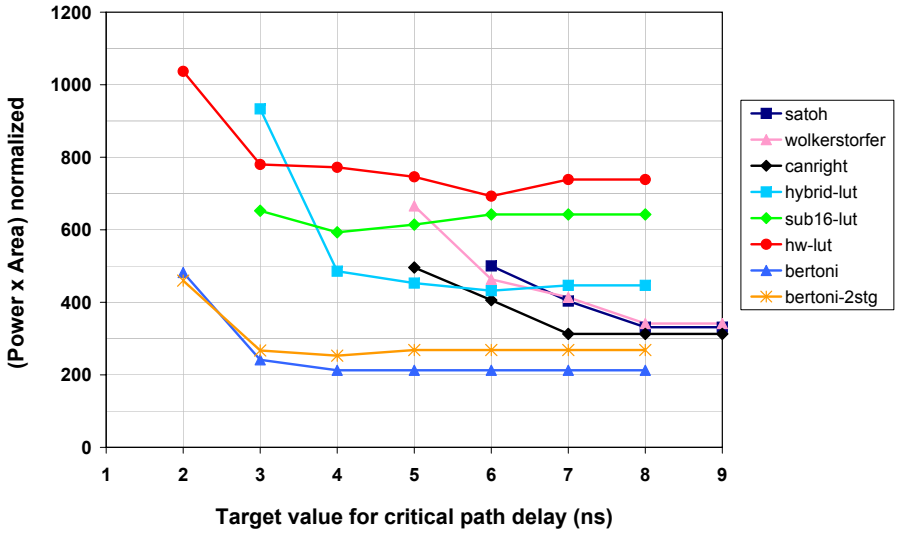

Figure 10.8: Power-area product vs. critical path delay.

The worst behavior is shown by **hw-lut** and **sub16-lut**. The calculating approaches have all similar characteristics for relaxed path conditions. Both **satoh** and **wolkerstorfer** also have similar properties for more stringent constraints on the critical path, whereas **canright** becomes more and more advantageous for faster designs. For the minimal delay of the calculating implementations, **hybrid-lut** is even slightly better than **canright**. However, **hybrid-lut** becomes very unattractive if the critical path needs to be shorter. The best tradeoff is delivered by the low-power approaches of **bertoni** and **bertoni-2stg**, where the former is slightly better than the latter.

Figure 10.9 displays total power consumption in relation to required silicon area. Generally, the points farther away from the point of origin belong to synthesis results for shorter critical path delays. The figure shows that calculating implementations tend to sacrifice power efficiency to achieve higher speed. On the other hand, low-power implementations trade silicon area for a shorter critical path. The **sub16-lut** implementation shows similar behavior. The look-up implementations **hw-lut** and **hybrid-lut** sacrifice area as well as power efficiency to approximately the same degree.

To minimize the critical path delay, the synthesizer uses optimization techniques like the utilization of standard cells with higher driving strengths and duplication of logic paths, which results in a considerable higher power consumption for signal switches. Calculating implementations have an inherently high number of signal switches and therefore incur an over-proportional increase in power consumption for reduced critical path delays. Low-power implementations, on the other hand, have much lower levels of signal activity which only leads to moderate increases in power consumption for shorter critical paths.

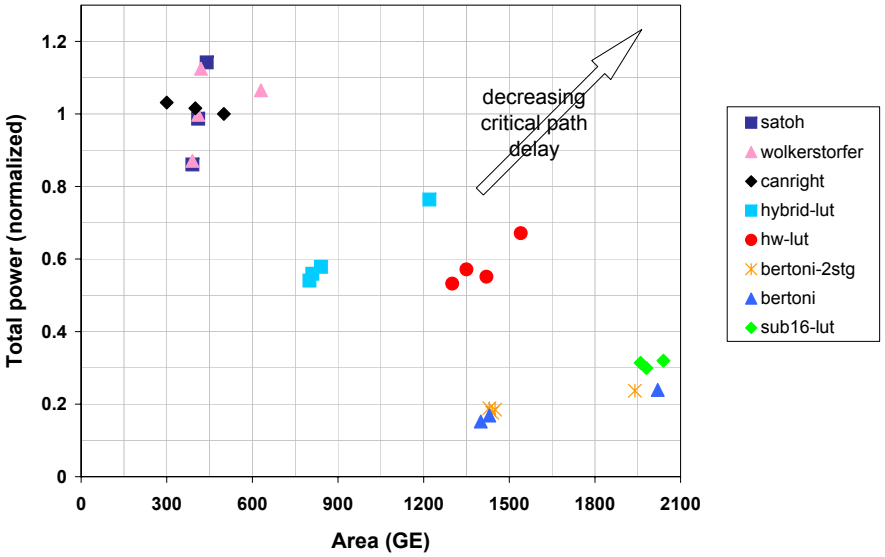

Figure 10.9: Total power consumption vs. area.

When compared to our practical results reported in [232] (where a 0.35 µm standard-cell technology has been used), the chip area and critical path delay figures correspond quite well to the current ones obtained with the UMC 0.25 µm technology. In regard to power consumption, we have noticed that the current figures indicate a less dramatic difference for the examined S-box implementations, as those given in [232]. We attribute this deviation to the different estimation methods used. While the earlier results were obtained via synthesizer estimation, our current figures result from a much more accurate power simulation of the placed and routed design. This of course has also lead to differences in all other metrics which include the power consumption results.

10.6 Summary and Conclusions

In this chapter we examined eight AES S-box implementations which follow three different design strategies. We compared various cost metrics like critical path delay, silicon area, and power consumption of these implementations based on synthesis runs with a 0.25 µm CMOS standard-cell library. According to our results, Canright's S-box design is the best choice for applications where small silicon area is of utmost importance (e.g., RFID tags). Bertoni et al.'s S-box is well suited for applications with a demand for low power or energy consumption, e.g., sensor nodes. In addition, Bertoni et al.'s S-box also has the shortest critical path, followed by the look-up implementations. While the results for the calculating implementations only apply to the AES S-box, the insights from the other two implementation strategies apply to any cryptographic S-box.

11

Effectiveness of Software Countermeasures Against Side-Channel Attacks on 32-bit Processors

In this chapter we investigate the impact of instruction set extensions for AES on the implementation security. We have compared several AES implementation options which incorporate state-of-the-art software countermeasures against power analysis attacks—with and without the use of instruction set extensions. For both scenarios we provide a thorough analysis for different countermeasures with regard to security, performance, and memory. In our evaluations we take a conservative stance and tip all odds towards favoring the attacker. In this setting, we have found that a moderate level of protection requires a considerable overhead both in terms of speed and memory. The instruction set extensions, which have been solely designed to increase performance, help to reduce this overhead significantly.

11.1 Introduction to Side-Channel Attacks

Today, most of the commonly used cryptographic algorithms can be considered to be reasonable secure in the face of mathematical cryptanalytic attacks. However, an implementation of a cryptographic algorithm in a specific device is not necessarily secure. Each concrete implementation has physical characteristics which can convey information about used keys which in turn can violate basic security assumptions. These exploitable physical characteristics are called side channels and the most frequently exploited ones are timing behavior [147], power

consumption [148], and electromagnetic emanation [81, 201]. Any straightforward implementation on a device is likely to be vulnerable to side-channel attacks. Power analysis attacks, which exploit the power consumption, have been studied very thoroughly, and many proposals have been made on how to make them more effective as well as on how to defend against them.

In Figure 11.1, the principles of a side-channel attack are depicted. The cryptographic device contains a key which is not accessible from the outside via the device's output (e.g., ciphertext). However, the key has some influence on the physical information conveyed by the side channel. Note that the side-channel output also depends on the device's input (e.g., plaintext) and other influences (e.g., voltage, clock frequency). An attacker models the side-channel output under consideration of the device's input (and possibly some of the other influences) and by guessing a part of the unknown key. Statistical analysis is used to match the modeled side-channel output against the actual one. In this way, information about the correctness of key hypotheses can be gained and the key can be revealed piecewise.

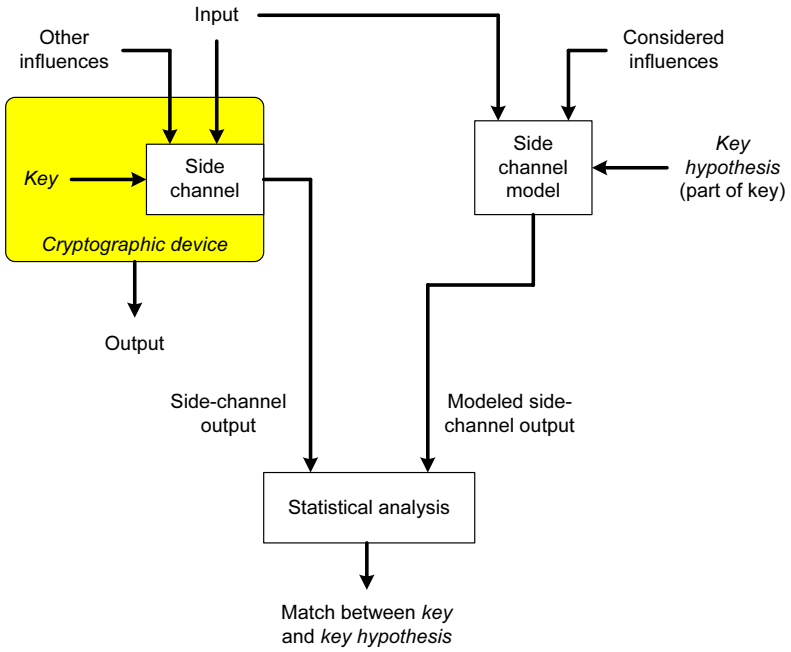

Figure 11.1: Overview of a typical side-channel attack.

While it is unlikely that side-channel attacks can be fully prevented, appropriate countermeasures can hamper an attack to the point where it becomes practically infeasible. Some hardware countermeasures have proven to be rather effective in doing this. On the other hand, counteracting power analysis in software is very hard, as the programmer normally has only a very limited influence on the power consumption of the processor. To make things even worse, in the

last years new attack variants have emerged, which are very effective against software implementations of cryptographic algorithms.

We have investigated the current situation regarding software countermeasures against state-of-the-art power analysis attacks. We have focused on 32-bit embedded processors and on the AES algorithm, but most of the discussed methods also work for processors of different word size and other cryptographic algorithms.

11.2 Power Analysis Countermeasures for Software Implementations

In power analysis, an attacker has to record the power consumption of a device while it performs cryptographic operations with an unknown key. A particularly powerful attack method is Differential Power Analysis (DPA) [148], which predicts intermediate values of the cryptographic algorithm and an according power consumption and matches it against the recorded power traces. In this fashion, the used key can be recovered even if the relevant information is deeply buried within noise.

In order to secure implementations of cryptographic algorithms against power analysis attacks, two principal approaches exist: Masking and hiding [166]. Both have the goal of reducing the correlation between the sensitive data and the observable side-channel information. Masking tries to break the link between the predicted intermediate values and the values processed by the device. This is usually done by concealing intermediate values during the calculation with a random mask. Hiding seeks to minimize the effect of the processed values on the power consumption.

Both hiding and masking can be realized in hardware or in software; the former generally offers more possibilities than the latter. Many concrete countermeasures have been proposed on different levels. Well-known examples are masking at the algorithmic level (e.g., [32]), randomization of operations (e.g., [166]), and the use of secure logic styles (e.g., [246]). For software implementations on a given platform, the options tend to be limited to masking schemes and to hiding through the randomization of executed operations in time. Software countermeasures trade performance for security, i.e., they entail a performance degradation. On the other hand, hardware countermeasures tend to increase silicon area and power consumption. In this section we give an overview of the two types of countermeasures.

11.2.1 Masking

Masking schemes split each intermediate value in a number of shares, which are then processed independently. Only by combining all the shares, the original value can be reconstructed. In its simplest form, a value a is split into two shares $a_m = a \circ m$ and m, where m is a random mask, so that $a = a_m \circ m = (a \circ m) \circ m$. The masks are generated by the device for each execution of the algorithm and

they should not be accessible by an attacker. A common choice for the operation $\circ$ is the logical XOR (Boolean masking).

Generally, we can distinguish between Boolean and arithmetic masking. In arithmetic masking, intermediate values and masks are combined with an arithmetic operation like addition or multiplication. For AES, Akkar et al. suggested a multiplicative masking scheme [3] where the intermediate values are concealed with a multiplicative mask $a_m = a \cdot m \pmod{n}$. Boolean masking uses the XOR operation to combine intermediate values and masks $a_m = a \oplus m$. Masking schemes for software implementations of AES based on Boolean masking have been proposed in [115] and [136].

The simple reasoning behind the masking countermeasure is that when each intermediate value is masked with a random value, the power consumption caused by this value cannot be predicted by an attacker. This holds under the condition, that each masked value a_m is independent of a. Usually, the masks are applied to the plaintext values at the beginning of the algorithm. During the execution of the algorithm it is necessary to keep track of the modification of the masks by the operations of the algorithm. For the AES operations ShiftRows and AddRoundKey, this can be done with virtually no effort, because they do not change the applied masks[1]. The MixColumns operation combines different values of one column of the AES State. As MixColumns is a linear operation, the modified masks after this step can be calculated easily: $MC(a_m) = MC(a \oplus m) = MC(a) \oplus MC(m)$.

In order to monitor the change of the masks through the nonlinear SubBytes transformation, a more elaborated approach is needed. A very common way to implement the SubBytes transformation in software is to use lookup tables: $a_{out} = S(a_{in})$, where S denotes the AES S-box, which is used on every byte of the State for SubBytes. In order to mask the SubBytes transformation, we have to calculate a masked table S' such that $S'(a_m) = S'(a \oplus m) = S(a) \oplus m'$. When implementing such a masking scheme, care has to be taken that all intermediate values stay masked during the critical computations of the algorithm. At the end of the calculation of the algorithm all masks have to be removed.

Implementing masking countermeasures correctly is a non-trivial task. Of special concern is the problem of unintentional unmasking. This can for example happen in a device which leaks the Hamming distance of subsequently processed values, i.e., the Hamming weight of the XOR of these two values [166]. In such a device, two subsequently occurring values which carry the same Boolean mask could therefore lead to a leakage which is again dependent on the unmasked values.

Provably-secure masking schemes for AES have been published in [32] and [192]. These schemes focus on hardware implementations. In [193], a proposal for a software implementation of the scheme presented in [192] has been made. This scheme has higher performance rates than a conventional lookup scheme, as long as a set of masks is only used for a single encryption. In schemes where

[1] ShiftRows may just change the position of individual masks while AddRoundKey has no effect on a Boolean mask: $(a \oplus m) \oplus k = (a \oplus k) \oplus m$.

masks are used for more than one encryption, the lookup-table approach is still faster. This is one of the reasons why we have chosen a lookup-based scheme for our implementation.

Simple masked implementations (with a single mask) are still vulnerable to higher-order attacks and template attacks. Such attacks combine information of the power consumption of the different shares (higher-order DPA preprocessing) so that the resulting power consumption is again dependent on the unprotected value a and thus susceptible to a "normal" first-order DPA attack. Higher-order attacks are discussed in [140, 191, 226, 251]. A template based attack on a protected AES software implementation has been published by Oswald et al. in [190]. Due to the presence of these powerful attacks, simple masking is normally not sufficient to provide adequate protection.

11.2.2 Hiding

In general, hiding can take place in two domains, namely in the time domain and in the amplitude domain. Hiding in the time domain tries to randomize the time of occurrence of a specific operation, whereas hiding in the amplitude domain tries to reduce the effect of the performed operation on the overall power consumption.

For software implementations, hiding in the time domain is normally easier to achieve. The goal is to distribute the occurrence of critical operations and intermediate values over a given period during each execution of an algorithm. This leads to a reduced correlation of targeted values at specific points in time. Two appropriate methods to achieve this randomization are the insertion of dummy operations and the shuffling of operations. Both insertion and shuffling are controlled by random values generated by the device. Inserted dummy operations should not be distinguishable from normal operations. Otherwise an attacker could be able to remove their effect from the power trace. Shuffling of operations means that for each execution of the algorithm, the order of the occurring intermediate values is changed. How these two methods can be applied to a software implementation of AES is described in Section 11.3.3.

Hiding in the amplitude domain is rather hard for software implementations. One possibility is to choose only such instructions which leak a minimum amount of information. This technique highly depends on the used device and its leakage properties. The statistic effects of hiding have been investigated in [47], [52], and [165].

11.3 Effectiveness of Software Countermeasures

This section gives a thorough evaluation of software countermeasures that can be applied to secure an AES implementation on a 32-bit platform. In this context we have considered two classes of processing platforms. The first class consists of typical 32-bit embedded processors with a standard RISC architecture. The

second class includes processors which have explicit support for cryptographic operations in their instruction set.

For our evaluation, we have selected the high-performance AES extensions described in Section 7.2.4 of Chapter 7. We give a brief recapitulation of the functionality of these custom instructions in the following. The AES instructions work on 32-bit words performing either four parallel AES S-box lookups (`sbox4s`, `isbox4s`, `sbox4r`) or a MixColumns transformation for a single State column (`mixcol4s`, `imixcol4s`). Our notation uses *rs1* and *rs2* to denote the two 32-bit input operands and *rd* for the 32-bit result of the instruction. Brackets with indices are used to select a part of the respective 32-bit value by specifying a range of bit indices. The symbol | is used for concatenation of four 8-bit values or two 16-bit values to a 32-bit value. S substitutes an 8-bit value according to the AES S-box, while MC transforms a 32-bit value following the AES MixColumns operation.

The instruction `sbox4s` is defined as:

$$\begin{aligned} \text{rd}[31..0] \;=\; & \text{S}(\text{rs1}[31..24]) \mid \text{S}(\text{rs2}[23..16]) \mid \\ & \text{S}(\text{rs1}[15..8]) \mid \text{S}(\text{rs2}[7..0]) \end{aligned}$$

In the case of `mixcol4s`, the definition is:

$$\text{rd}[31..0] \;=\; \text{MC}(\text{rs1}[31..16] \mid \text{rs2}[15..0])$$

The definition of `isbox4s` and `imixcol4s` is similar, with the only difference that the inverse AES S-box and the InvMixColumns transformation are used, respectively. Finally, the `sbox4r` instruction has only one input operand (rs1), whose bytes are transformed with the AES S-box and where the result is rotated by one byte to the left:

$$\begin{aligned} \text{rd}[31..0] \;=\; & \text{S}(\text{rs1}[23..16]) \mid \text{S}(\text{rs1}[15..8]) \mid \\ & \text{S}(\text{rs1}[7..0]) \mid \text{S}(\text{rs1}[31..24]) \end{aligned}$$

The `sbox4r` instruction is designed for use in the AES key schedule, while the other instructions are intended to speed up the AES round transformations.

In the following sections we analyze different options for power analysis countermeasures. The most powerful attacks are listed and implementation-specific details for use of the instruction set extensions are given. The maximum correlation coefficient ρ is stated for each attack.

With the help of ρ, the effectiveness of different attacks and the impact of various countermeasures can be compared: An attack on the protected implementation requires at least $(\frac{\rho_{unprotected}}{\rho_{protected}})^2$ more power traces [166]. For our estimations we have set $\rho_{unprotected} = 1$ and can therefore state the security gain as $(\frac{1}{\rho})^2$, where ρ always denotes the correlation coefficient for an attack on the protected implementation. The correlation coefficient has been determined under the assumption that the Hamming weight of processed operand values leaks through the power consumption. More specifically, we have restricted the evaluation to single 32-bit operands. Note that our strategy estimates ρ for environments which are free of most noise components which would be encountered

in practical measurements. Adhering to the classification in [166], the following noise components are neglected:

- All electronic noise.
- All switching noise, except for switching noise generated by those parts of the attacked 32-bit operand which are unrelated to the attack at hand.

Hence, our evaluation does not allow to conclude for the absolute effectiveness of a given attack as this would require to account for the noise[2]. Nevertheless, our estimations of ρ are sufficient for a relative comparison of unprotected and protected implementations.

Formula 6.5 of [166] (replicated below in Equation 11.1) illustrates the connection between the correlation in a noisy and a noise-free setup:

$$\rho(H_i, P_{total}) = \frac{\rho(H_i, P_{exp})}{\sqrt{1 + \frac{1}{SNR}}} \tag{11.1}$$

In a practical attack, the achievable correlation between a hypothesis H_i and the noise measurements P_{total} depends on the correlation in a noise-free environment ($\rho(H_i, P_{exp})$) and the signal-to-noise ratio (SNR). The former is determined by our evaluations. The SNR will stay more or less the same for the different attacks and will just constitute a common scaling factor which can be neglected in relative comparisons.

Many devices leak the Hamming distance of subsequently processed values, but it is very hard to determine the correlation coefficient for such a setting without taking many details of the processor architecture and software implementation into account. We have therefore taken the Hamming-weight leakage model as a lower bound for devices that leak the Hamming distance. This assumption holds as long as the software implementation avoids potential vulnerabilities due to the Hamming-distance leakage, e.g., unintentional unmasking as explained in Section 11.2.1.

11.3.1 Unprotected Implementation

An unprotected 32-bit AES software implementation is vulnerable to a multitude of attacks. One of the most powerful attacks is a first-order DPA on an 8-bit intermediate result after the S-box lookup ($\rho = 1$). The key expansion can also be targeted directly with a template-like attack as described in [164]. This attack extracts the Hamming weights of 8-bit intermediate values of the key expansion and uses the dependency of these values to narrow down the number of potential keys. The use of the instruction set extensions allows to calculate the key schedule with 32-bit values only, which makes the attack from [164] attack infeasible.

[2]The correlation coefficients observed in a practical attack will be lower due to the additional noise components.

11.3.2 Masking

A masked implementation protects critical intermediate values with a random mask. An intermediate value of the AES operation can be considered critical when it depends on a small portion of the (round) key and on the plaintext or ciphertext. In this case the attacker can guess the part of the key and verify the guess through analysis of the measured power traces under consideration of the actually used plaintext or ciphertext. The choice of masks and the processing order of masked values must always be done carefully with regard to the leakage of the device to prevent problems like unintentional unmasking. If, for example, the device leaks the Hamming distance of two subsequently processed values, then subsequent values must never carry the same mask. Otherwise, the mask would be removed, and the device would leak information about the unmasked values.

If the masking countermeasure is implemented properly, it can prevent first-order DPA attacks. However, a masked implementation is still vulnerable to higher-order DPA attacks. In such an attack, several points of each power trace are combined to form a single value by means of a preprocessing function *pre*. The output of *pre* again depends on some predictable value. The preprocessed values can then be used in a conventional first-order DPA attack. A second-order DPA attack is normally sufficient to break a masked implementation, when only a single mask is used to protect critical intermediate values. The targeted values for preprocessing are either a masked intermediate value and the corresponding mask, or two intermediate values with the same mask.

The best vantage point to break a masked AES implementation is the masked S-box lookup, which is used for SubBytes. This lookup requires masked 8-bit input and output values, which are easier to target than the masked 32-bit values resulting from other transformations (e.g., MixColumns). The cost for precomputing a single masked S-box is very high, and it is therefore necessary to reuse masked S-box tables. This results in the processing of 8-bit values with the same mask, which can be targeted in a second-order attack with $\rho = 0.24$ as has been shown in [166].

But even if no 8-bit value carries the same mask, the preprocessing function could use the power consumption of the mask itself[3] as second value. In the worst case for the attacker, this 8-bit mask will only occur in form of a 32-bit word, where the other 24 bits are random[4]. Even in this case, the level of protection against a second-order DPA attack is rather low ($\rho \approx 0.1$).

A possibility to prevent the S-box lookup in software is to perform most of the AES round as table lookup (T-table lookup). However, the precomputation of masked T-tables would be much more costly than the precomputation of masked S-boxes. Moreover, a T-table lookup still requires an 8-bit masked input value, which can be targeted in an attack.

As the effort for higher-order DPA attacks is expected to grow exponentially

[3] Note that it can normally not be prevented that the mask value influences the power consumption at some point in time where the mask is generated, stored, loaded, applied, etc.

[4] Note however, that this can only be achieved with the help of the instruction set extensions.

with the order, it is assumed that a masking scheme with enough shares will make practical attacks infeasible. A higher-order masking scheme for AES based on this idea has been developed by Schramm et al. [218]. However, Coron et al. have demonstrated that this scheme is susceptible to third-order DPA attacks irrespective of the number of used shares [53]. Another problem is posed by the large computational overhead which is required for refreshing the masks. In [218], it has been shown that a single AES encryption with resistance against second-order DPA attacks requires over 40 times more clock cycles on an 8-bit platform (about 200,000 clock cycles in total).

11.3.3 Randomization

In the following, we will denote countermeasures of hiding in the time domain (cf. Section 11.2.2) as *randomization*. In a randomized AES implementation, the occurrence of a specific intermediate value at a specific point in time is reduced to a certain probability. This can be done by shuffling of operations and by random insertion of dummy operations. In this case, an attacker needs to capture more power traces in order to compensate for this uncertainty.

Simple solutions, like the random insertion of `nop` instructions, are likely to be detected and removed by an attacker. Therefore, if dummy operations are added, it is important that they cannot be distinguished from the genuine operations. This can be achieved by performing the same transformations as in the normal AES operations on some dummy data.

The best degree of randomization can be achieved by using both the shuffling of operations and the insertion of dummy operations. In AES, the smallest unit of data, whose processing can be randomized, is the 8-bit input and output value used in the S-box lookup. The 16 S-box lookups per AES round can therefore be shuffled, resulting in a probability of $p = \frac{1}{16}$ for a specific value at a specific point in time. Dummy operations can be inserted by processing a certain number of dummy values. Processing of complete dummy States (i.e., 4×4-byte matrices) seems to be a good granularity for that purpose. If N dummy States are processed in addition to the genuine State, then the probability for the occurrence of a specific value goes down to $p = \frac{1}{(N+1)\cdot 16}$.

It would be very inefficient to perform a selection for each of the $(N+1)\cdot 16$ byte values separately. Moreover, the AES algorithm does not allow to perform all critical round transformations with just a single byte. The smallest value which is sufficient for all those transformations is a single State column. For practical implementation it is sufficient to determine the processing order of the bytes in an orthogonal way: The States are processed one after the other, i.e., the processing of the genuine State is randomly embedded within the processing of the dummy States[5]. For each State, the columns are processed in a fixed order beginning with a randomly chosen column[6]. For each column, the bytes are

[5] Example for $N = 3$: Process a complete dummy State, followed by the genuine State, followed by two dummy States.

[6] Example: Process second column, followed by third, fourth and first column.

processed separately and also in a fixed order starting with a randomly chosen byte[7].

The randomization degree p determines the resistance against DPA attacks. The power traces obtained from an implementation with randomization are often referred to as misaligned power traces. A direct DPA attack on the misaligned traces would require $(\frac{1}{p})^2$ more traces to compensate for the randomization. However, Clavier et al. [52] have proposed to sum up all points in the power trace where the targeted value can occur. This approach is often referred to as *windowing*. With this approach, an attacker only requires $\frac{1}{p} = (N+1) \cdot 16$ more traces to defeat the randomization.

Therefore, we can assume that the number of power traces to attack a randomized AES implementation scales up with a factor of only $(N+1) \cdot 16$, as $\rho = \sqrt{p} = \frac{1}{\sqrt{(N+1) \cdot 16}}$. Most of the overhead of a randomized implementation comes from the preparation of the randomization and the byte-wise processing of the AES State. Doubling the security (which corresponds to doubling of $(N+1)$) roughly doubles the total running time. This results in a considerable overhead.

11.3.4 Masking and Randomization

For better protection, an AES implementation needs to combine masking and randomization countermeasures. However, there are still several possible attacks which could break such an implementation rather efficiently.

An attacker will try to defeat the masking with a second-order attack. At least one of the attacked intermediate values (i.e., a masked 8-bit value) is protected by randomization. As we have already outlined in Section 11.3.3, a very effective way to defeat randomization is to sum up the power consumption at all moments in time where the attacked value can occur (recall that for our considered randomization there are $(N+1) \cdot 16$ points in time, where N is the number of dummy States). The second attacked value can either be the mask of the first value, or another randomized intermediate value carrying the same mask as the first value.

There are two main strategies on how to use the second value in an attack. On the one hand, this value can be employed to introduce a bias in the occurring masks. On the other hand, the value can be combined with the first one to yield a result that depends on the unmasked value.

11.4 Attacks on Masked and Randomized AES Implementations

We have shown in the previous section, that a protection by masking or randomization alone cannot withstand power analysis attacks. In this section we analyze the possible attacks on software implementations which use a combination of both countermeasures. The attacks presented in this section have either

[7] Example: Process third byte, followed by fourth, first and second byte of the column.

been published and evaluated or are natural extensions or combinations of existing attacks. The method of windowing (cf. Section 11.3.3) published by Clavier et al. [52] is fundamental to all of the examined attacks, as it is a very good way to compensate the effects of the randomization countermeasure. The possibility of second-order DPA attacks has already been mentioned in the original publication of Kocher et al. [148]. Second-order attacks on software implementations of block ciphers have been analyzed in [191].

In [190], Oswald et al. have evaluated the effectiveness of template-based attacks against masked software implementations and have shown that such methods can be very effective. However, as long as the targeted operation used for template-building remains randomized in time, we assume that it is very hard to create well-matching templates, which lead to better results than techniques based on counteracting randomization, e.g., windowing.

11.4.1 Biasing Masks and Windowing

A very powerful attack is to introduce a bias into the distribution of masks used by the device. Such a bias can lead to a dramatic decrease of security. This idea has been introduced by Jaffe [139], and practically evaluated by Oswald et al. [190]. In practice, an attacker can use template methods to guess the mask value and discard all traces where the mask had some specific property. Figure 11.2 shows the timeline of a power trace, where the time of occurrence of targeted values is marked at the top. Below the timeline it is shown how the power consumption values at these times would be used in a biased-mask attack. Windowing is used to sum up the power consumption at all points in time in the selected traces where the attacked value can occur (due to randomization). A classical first-order DPA attack is performed on the resulting preprocessed power values.

Without instruction set extensions, the 8-bit masks of the S-box can be targeted directly during the generation of the masked S-box. With instruction set extensions, the masked S-box can be generated using only 32-bit masks (provided that four masked S-boxes are used). A bias of either the 8-bit or 32-bit mask has a devastating effect on the security. For example, biasing the 8-bit masks to a Hamming weight (HW) ≥ 6 yields $\rho = -0.1$ (for $N = 1$). For 32-bit masks, a bias of HW ≥ 20 results in $\rho = -0.05$ (again for $N = 1$).

Increasing the degree of randomization does not lower the correlation coefficient very effectively (cf. Table 11.2). For example, with 8-bit masks biased to HW ≥ 6, a doubling of the randomization effort (i.e., N goes up from 5 to 11) lowers the correlation coefficient only from $\rho = -0.06$ to $\rho = -0.04$[8].

Note that a possible defense against this attack could consist of randomizing the time of occurrence of each mask. However, the mask and values directly dependent on the mask occur at several points in the computation, e.g., generation of the mask, appliance of the mask to the S-box, calculation of the mask

[8]Remember that $N = x$ means that x dummy State are processed and that each State byte can be processed at one of $(x + 1) \cdot 16$ locations.

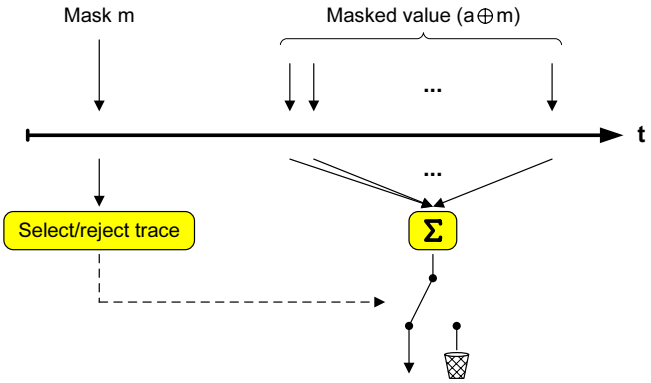

Figure 11.2: Information extraction from power traces for biasing masks and windowing.

after MixColumns, (re)masking of the key schedule. Proper randomization of all these operations would be quite challenging and also incur a considerable overhead in terms of performance.

11.4.2 Second-Order DPA Followed by Windowing

A second-order DPA can be combined with windowing in order to break the masking and randomization. This approach can be seen as performing multiple second-order DPA attacks in parallel. The attack can be done by combining the power consumption for the mask processing with each of the $(N+1) \cdot 16$ points in time where the targeted masked value can occur (due to randomization) using the second-order preprocessing function *pre*.

However, due to the randomization, the attacker does not know which of the resulting values corresponds to the targeted value. This is the same problem as in an implementation which has only randomization countermeasures. Consequently, an efficient solution is to sum up all $(N+1) \cdot 16$ preprocessed values and to perform a first-order DPA attack on the result. Figure 11.3 depicts this approach.

The effectiveness of this attack can be evaluated for both attack stages separately. In the first stage, the second-order DPA preprocessing function is applied to each pair of values (mask and masked value). For our randomization scheme we have an 8-bit masked value. As already stated in Section 11.3.2 we have $\rho = 0.24$ for 8-bit masks and $\rho \approx 0.1$ for 32-bit masks (using instruction set extensions). The summation of the second attack stage corresponds to windowing, which scales down the correlation coefficient with a factor of $\frac{1}{\sqrt{(N+1)\cdot 16}}$. The overall correlation coefficient stays therefore rather high: For the 8-bit masks we get $\rho = \frac{0.24}{\sqrt{(N+1)\cdot 16}}$, and for 32-bit masks we get $\rho \approx \frac{0.1}{\sqrt{(N+1)\cdot 16}}$. So in order to achieve $\rho = 0.01$, we would need at least $N = 5$.

Principally, it would be desirable to randomize the occurrence of the mask to

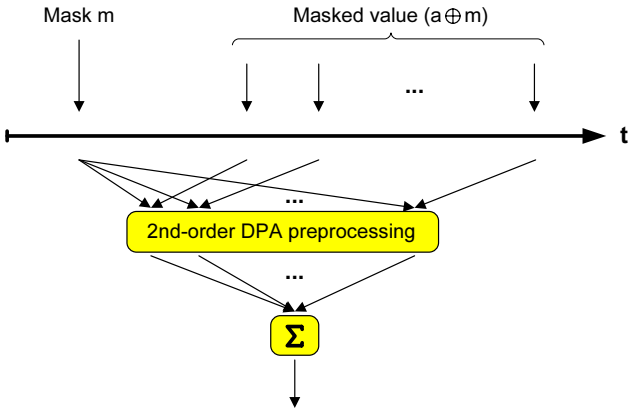

Figure 11.3: Information extraction from power traces for second-order DPA followed by windowing.

the same degree as the masked value. This measure would require to sum up all possible combinations where mask and masked value can appear. The number of combinations is $((N+1)\cdot 16)^2$, which would lead to a reduction of the correlation by a factor of $(N+1)\cdot 16$ after windowing. At $N = 1$, the correlation would already be about as low as $\rho = 0.003$ for 32-bit masks. However, as already mentioned in Section 11.4.1, randomization of the mask would be very costly in terms of performance.

11.4.3 Attack on Weak Randomization

Targeting two randomized intermediate values which carry the same mask is normally less efficient than to target one fixed (e.g., the mask) and one randomized value. However, a weak randomization can be broken more easily with this strategy.

In this context, a weak randomization is one where two intermediate values with the same mask always occur with a fixed distance in time. An example for this are the S-box inputs of the first and second AES round, when the used S-boxes have the same input masks and the two lookups are not randomized separately. The attacker can therefore apply the second-order DPA preprocessing function to each such pair of values, which is depicted in Figure 11.4. The rest of the attack is exactly the same as the one described in Section 11.4.2 (summation followed by first-order DPA).

Targeting two 8-bit intermediate values with the same mask is equivalent to targeting one intermediate value and the according 8-bit mask. The correlation coefficient is therefore $\rho = \frac{0.24}{\sqrt{(N+1)\cdot 16}}$. However, as one of the targeted values may occur deeper in the AES operation (e.g., the S-box lookup in the second round), a hypothesis can be harder to obtain. For example, it might be necessary to hold some parts of the plaintexts constant so that the intermediate values stay

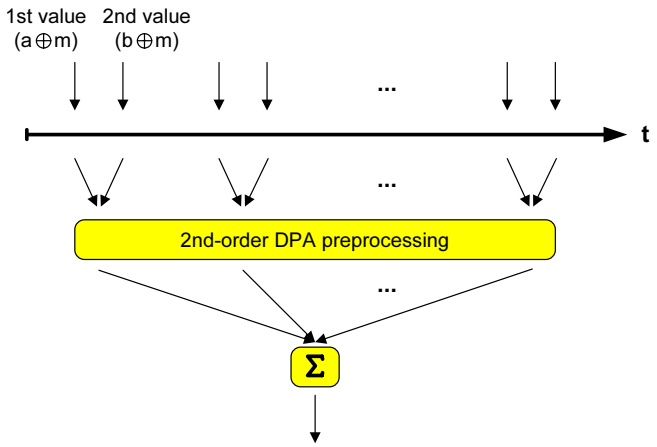

Figure 11.4: Information extraction from power traces when attacking a weak randomization.

predictable. Moreover, it could also become necessary to formulate hypotheses for more than a single key byte.

The effectiveness of this attack can again be evaluated for both attack stages independently. In the first stage, the second-order DPA preprocessing function is applied to each pair of values with the same mask. For our randomization scheme we have two masked 8-bit values, which yields $\rho = 0.24$ [166]. The summation of the second attack stage again corresponds to windowing, which reduces the correlation coefficient by a factor of $\frac{1}{\sqrt{(N+1)\cdot 16}}$. The resulting correlation coefficient remains rather high with $\frac{0.24}{\sqrt{(N+1)\cdot 16}}$ (e.g., $\rho = 0.01$ would require $N = 35$).

To counteract this attack it would be necessary to prevent a fixed distance in time between identically-masked values. In our example the S-box lookup in the first and second AES round would need to be randomized separately[9].

11.4.4 Windowing Followed by Second-Order DPA

Another way to combine second-order DPA and windowing is to perform windowing first to counteract the effects of randomization, and to do a "classical" second-order DPA attack on the result. Figure 11.5 depicts the processing steps performed on every power trace. The resulting value can then be subjected to a first-order DPA attack. A preprocessing function *pre*, which is generally very effective, is the absolute difference of the inputs: $pre(a, b) = |a - b|$ [166]. For this function, it is important that both a and b are of the same magnitude, e.g., if a is a single point from the trace and b is a sum of n points, then the preprocessing function should scale a up to b: $pre(a, b) = |n \cdot a - b|$. It should be noted that

[9] The same applies to the S-box lookups in the last two rounds.

the proposed preprocessing function is very sensitive to the amount of noise[10]. Therefore, this attack is presumed to be very ineffective in a practical setting.

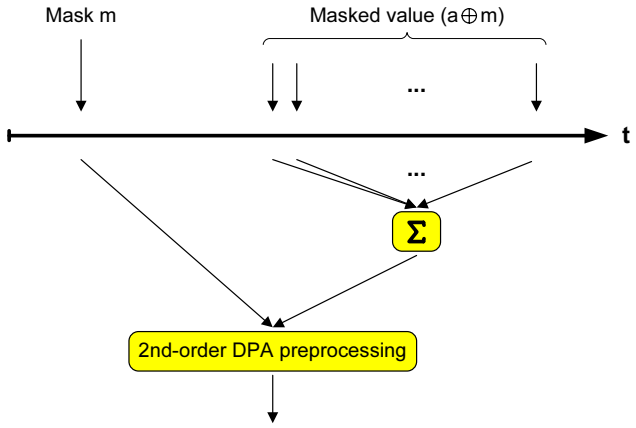

Figure 11.5: Information extraction from power traces for windowing followed by second-order DPA.

For a randomization degree of $N = 1$, the correlation is about $\rho = 0.013$ for 8-bit masks and $\rho = 0.012$ for 32-bit masks. Doubling the randomization degree approximately halves the correlation coefficient. As can be seen, this attack is even rather ineffective in our evaluation setting with reduced noise.

11.5 Performance Estimation

The security evaluation of the last section has shown that there are powerful attacks which can break implementations even when they employ very sophisticated countermeasures. Under the assumptions of our analysis, one might be inclined to regard the use of software countermeasures as futile. Nevertheless, there are scenarios where a protected implementation might be desired, even if our evaluations indicate only a rather moderate protection:

- In our evaluations, all odds are tipped in the attacks favor. It is unlikely, that this will also be the case in practical attack scenarios.
- In a device with a fixed processor, the use of software countermeasures is likely to be the only available option. In some applications, a certain degree of implementation security could still be much better than none at all.
- The most powerful attacks used in our security evaluation might not be applicable due to other security measures of the device (e.g., limited num-

[10]The noise distributions differ for $n \cdot a$ (multiple of a single distribution) and b (as the sum of n independent distributions).

ber of AES encryptions/decryptions, plaintext/ciphertext not selectable by the attacker, etc.).

- The device has some hardware countermeasures (e.g., noise generators) and the resistance against power analysis should be amplified by the software countermeasures.

In order to provide performance estimations for different countermeasures, we have implemented AES-128 encryption with both masking and a scalable randomization. With the help of this implementation we have estimated the performance for several design options and degrees of randomization. First, we present the most important design decisions and implementation characteristics of our solution. Then we give the performance figures for interesting implementation variants regarding expected security level, speed, and memory requirements.

11.5.1 Features of Our Protected AES Implementation

Some basic design decisions for our 32-bit implementation are similar to the secure AES implementation for 8-bit microcontrollers presented in [115]. This mainly concerns the basic types of countermeasures (masking and randomization), the concept of randomized zones, etc. We assume the availability of a random number generator to provide mask values and randomization parameters.

The masking scheme requires six distinct byte masks as input. Two mask bytes are used to derive a masked S-box lookup table with input mask M and output mask M'. The remaining four bytes (denoted M1, M2, M3, and M4) mask each input column of the MixColumns transformation. The corresponding output masks can be derived by performing MixColumns on the mask values. More precisely, M1 to M4 are used as an input column for the MixColumns transformation, resulting in the output masks M1', M2', M3', and M4'.

All operations which yield intermediate results depending on a relatively small portion of the key are executed in a randomized fashion. Randomization is achieved both by shuffling of operations as well as the addition of dummy operations. The processing of the AES State is shuffled so that each byte is processed at one of 16 moments in time with equal probability. Dummy operations are inserted as normal AES round transformations, but work on a random State (*dummy State*). The processing of the *genuine State* is randomly embedded in between the processing of several dummy States. The parts of the encryption where execution is randomized are denoted as *randomized zones*. The randomized zone at the beginning of the AES encryption reaches up to and including the SubBytes operation of round 2, while the randomized zone at the end starts with SubBytes in round 9. Figure 11.6 gives a general overview of the program flow for the AES implementation and shows the masks on the State as well as the randomized transformations.

Randomization of operations is costly in terms of performance. Therefore it is desirable to keep the randomized zones as short as possible. In our implementation, we have reordered the round transformations so that ShiftRows is not

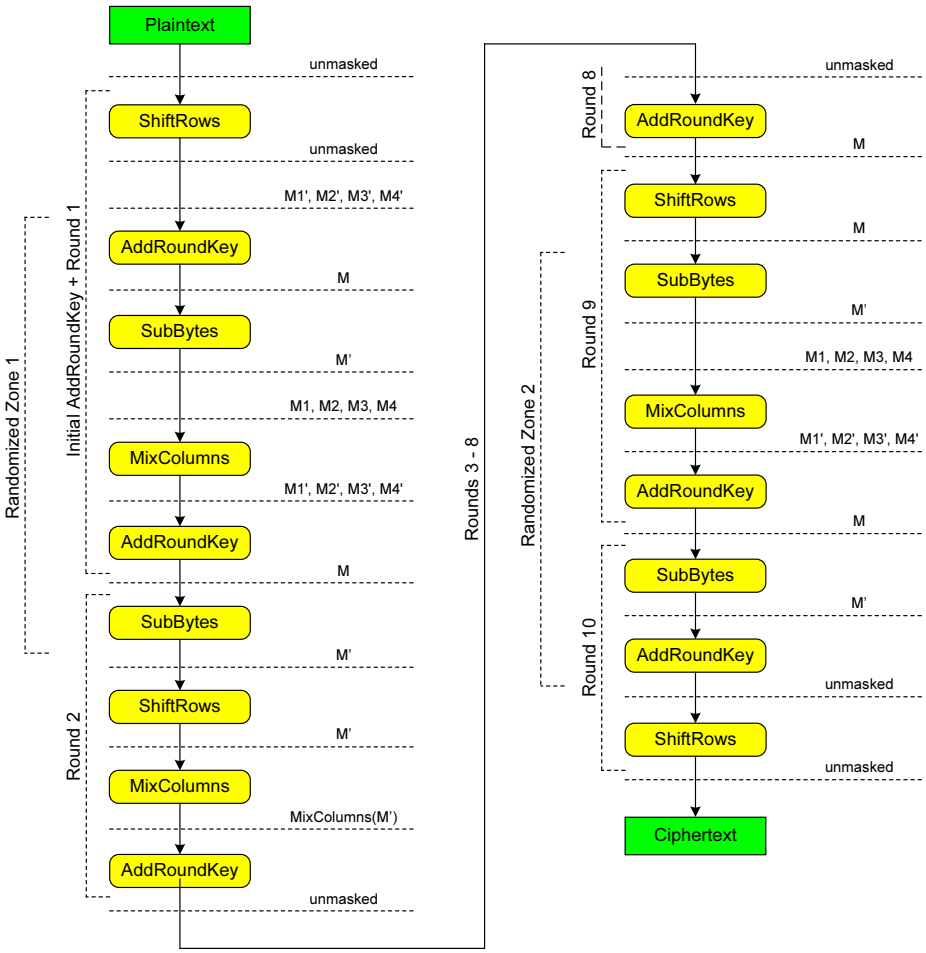

Figure 11.6: Program flow of masked and randomized AES encryption.

included in the randomization. This reordering requires the first and last round key to be transformed with ShiftRows and InvShiftRows, respectively.

In order to reduce the overhead for masking, the AddRoundKey operation is used for remasking whenever possible. This requires masks to be applied on some of the round keys. The masks on these round keys must be renewed whenever the masks change. When the masks are changed for each AES encryption—which is the ideal case—then it would be equally efficient to change the mask explicitly during the AES encryption.

In our implementation, the rounds 3 to 8 are not masked. AddRoundKey of round 2 removes the masks from the State, and AddRoundKey of round 8 masks the State again. All unmasked intermediate values have therefore been subjected to three AddRoundKey transformations and depend on sufficiently

many key bytes, to prevent an efficient DPA attack[11]. The advantage of the unmasked inner rounds is that the AES instruction set extensions can be fully used.

The randomization follows the concepts described in Section 11.3.3. In the randomized zones, only values which depend on a single State byte are processed. This allows for a randomization degree of $(N + 1) \cdot 16$, where N is the number of dummy States.

11.5.2 Performance Figures

Table 11.1 contains the execution times and RAM requirements for several implementations of AES-128 encryption with masking and randomization countermeasures. The performance figures are given for the case without instruction set extensions (pure software) as well as with instruction set extensions (ISE). The RAM requirements for a specific implementation are always the same for both cases. The cycle counts are given in dependence of the number of dummy States (N).

We have given performance figures for three protected implementations, which employ both masking and randomization countermeasures. The cycle counts include all overhead when the masks are refreshed for each new encryption. The first implementation (1SB_WR) uses 1 masked S-box and a weak randomization (weak in the sense defined in Section 11.3.4). The second implementation (4SB_WR) is similar, but uses 4 masked S-boxes. The last implementation (4SB_SR) has a strong randomization.

For comparison, the performance figures of an unprotected implementation, as stated in [237], are provided.

Table 11.1: Performance and RAM requirements of AES-128 encryption implementations.

Countermeasures	**Pure Software** cycles	**ISE** cycles	**Memory (RAM)** bytes
None	1,637	196	176
1SB_WR	$6,465 + 1,888N$	$2,023 + 1,028N$	476
4SB_WR	$14,958 + 1,888N$	$3,631 + 1,028N$	1,248
4SB_SR	$15,332 + 2,208N$	$3,978 + 1,348N$	1,388

Table 11.2 gives a complete analysis of the security/performance trade-off for the three protected implementations. Note that SW denotes the software implementation, while ISE denotes the respective implementation with instruction set extensions. The table lists the estimated correlation coefficients for the four attacks presented in Section 11.4: Biasing masks and windowing (BM), second-order DPA followed by windowing (2W), attack on weak randomization (WR),

[11] In the best case for the attacker, 2^{40} hypotheses are required to mount an attack on the unprotected inner rounds.

and windowing followed by second-order DPA (W2). The maximum correlation coefficient is listed in the last column.

Table 11.2: Analysis of the security/performance trade-off.

Implementation	Performance	BM	2W	WR	W2	max(ρ)
1SB_WR (SW), $N = 0$	6,465	-0.14	0.06	0.06	0.03	-0.14
1SB_WR (SW), $N = 3$	12,129	-0.07	0.03	0.03	< 0.01	-0.07
1SB_WR (SW), $N = 5$	15,905	-0.06	0.02	0.02	< 0.01	-0.06
1SB_WR (SW), $N = 11$	27,233	-0.04	0.02	0.02	< 0.01	-0.04
1SB_WR (ISE), $N = 0$	2,023	-0.14	0.06	0.06	0.03	-0.14
4SB_WR (ISE), $N = 0$	3,631	-0.05	0.03	0.06	0.02	0.06
4SB_SR (ISE), $N = 0$	3,978	-0.05	0.03	n/a	0.02	-0.05
4SB_SR (ISE), $N = 1$	5,326	-0.04	0.02	n/a	0.01	-0.04
4SB_SR (ISE), $N = 3$	8,022	-0.03	0.01	n/a	< 0.01	-0.03
4SB_SR (ISE), $N = 5$	10,718	-0.02	0.01	n/a	< 0.01	-0.02
4SB_SR (ISE), $N = 11$	18,806	-0.01	< 0.01	n/a	< 0.01	-0.01

For the pure software implementation, the biased-mask attack (BM) is the most powerful one. In software, the only option is to increase the randomization degree N. However, the correlation coefficient only decreases very slowly with rising N. When instruction set extensions are available, we can work exclusively with 32-bit masks if we use four masked S-boxes instead of one (4SB_WR). In that case, the attack exploiting the weak randomization becomes the most efficient one. In order to counteract, we use the implementation with strong randomization (4SB_SR), which makes this attack inapplicable. Then the biased-mask attack becomes again the most effective one. With heavy randomization ($N = 11$), the correlation coefficient can be pushed down to $\rho = -0.01$. This corresponds to an increase of the security level by four orders of magnitude in comparison to an unprotected implementation. This comes at the price of an execution time, which is increased by two orders of magnitude (cf. Table 11.1). Compared to the unprotected pure software implementation, the execution time is increased by one order of magnitude.

11.6 Summary and Conclusions

In this chapter we have provided a thorough evaluation of power analysis countermeasures in software in the face of state-of-the-art attacks. We have concentrated on 32-bit embedded processors, but most of the results could also be applied to 8-bit and 16-bit processors. By means of an AES implementation we have shown the impact of power analysis countermeasures on the performance and RAM requirements. When restricted to the original instruction set architecture, the attainable degree of protection of our protected implementation is increased by three orders of magnitude. If the processor is equipped with custom instructions for AES, then a protection level of four orders of magnitude is achievable. However, the performance penalty is rather high, so that it is

probably not acceptable for all applications.

In Chapter 12 we investigate the use of hardware countermeasures to facilitate the secure use of instruction set extensions. In Chapter 13 we demonstrate the practicability of the advanced DPA attacks presented in this chapter.

12

Secure Implementation of Instruction-Set Extensions on 32-bit Processors

In this chapter we discuss and analyze different techniques for increasing the side-channel resistance of AES software implementations which use instruction set extensions. As instruction set extensions emphasize the hardware/software co-design paradigm, we also investigate the problem of secure implementation by having an integrated view of both hardware and software components. We propose a combination of hardware and software-related countermeasures and investigate the resulting effects on performance, cost, and security. Our experimental results show that a moderate degree of protection can be achieved with a simple software countermeasure. Hardware countermeasures can lead to a much higher resistance against side-channel attacks at the cost of a moderate increase in silicon area and power consumption. Although we put a focus on the situation for AES, most of our proposed countermeasures are generic and can principally be used for the secure implementation of any cryptographic algorithm.

12.1 Side-Channel Attacks on Instruction Set Extensions

Generally, side-channel attacks on a cryptographic device are possible when intermediate values with a close relation to the key have an influence on an externally observable physical value. If an attacker can measure this physical value while the device performs operations with some unknown key, a portion or even all of this key might be disclosed. Well-known examples of such physical values are execution time, power consumption, and electromagnetic emanation [166].

Side-channel information can be used in different methods to extract sensitive data. So-called differential methods are especially powerful as they employ statistical analysis to extract information even from very noisy measurements. In the case of the power consumption side channel, the corresponding method is *differential power analysis (DPA)*, which has first been published by Kocher et al. [148]. Our work centers around methods to increase the resistance of AES implementations against DPA.

DPA targets intermediate values which depend on a small portion of the key and the plaintext or ciphertext. We denote such vulnerable intermediate results as *critical data* and manipulations of such values as *critical operations*. For secret-key algorithms, critical data is commonly processed at the beginning and at the end of the algorithm, where the intermediate results are strongly related to the plaintext or ciphertext. In order to defend against DPA, we first need to analyze the flow of critical data through the processor and its potential impact on the overall power consumption.

Figure 12.1 depicts a typical datapath of a RISC processor augmented with a custom functional unit. This example shows four pipeline stages:

- Decode stage between the *register file* and the *op1* and *op2* registers.
- Execute stage between the *op1* and *op2* registers and the *result* register.
- Memory stage between the *result* register and the *wr.result* register.
- Write stage between the *wr.result* register and the *register file*.

There are several feedback paths from different stages back into the decode and execute stage. The memory stage is connected to the data cache. Also depicted are inputs and outputs for the control flow (*ex.PC*, *jmpl address*, *wr.PC*). The functional unit for the instruction set extensions (ISE FU) is located in parallel to the original ALU/Shifter unit.

In order to analyze the side-channel leakage of this datapath, we assume that the *reg1* bus carries critical data, which can be subjected to a side-channel attack. The parts of the datapath indicated by bold lines process or hold values that are related to the original value of *reg1*. It is important to note that the critical data passes through both the ALU/Shifter as well as the ISE FU, irrespective of the result selected by the following multiplexer. Also note that all feedback paths, the *jmpl address*, as well as the data cache input carry critical values.

An analysis of AES software using the instruction set extensions has revealed that almost all instructions which process critical data can be easily rearranged in a way so that no feedback path needs to be used (i.e., no forwarding occurs). Moreover, critical data is not used for control transfer and is never stored in memory. Therefore, our first step to improve implementation security is to detach these paths from the ISE datapath. The resulting, slightly modified datapath is shown in Figure 12.2.

Two important changes can be seen in this figure. Firstly, operands for the functional units (*op1* and *op2*) can be blocked from entering the ALU/Shifter

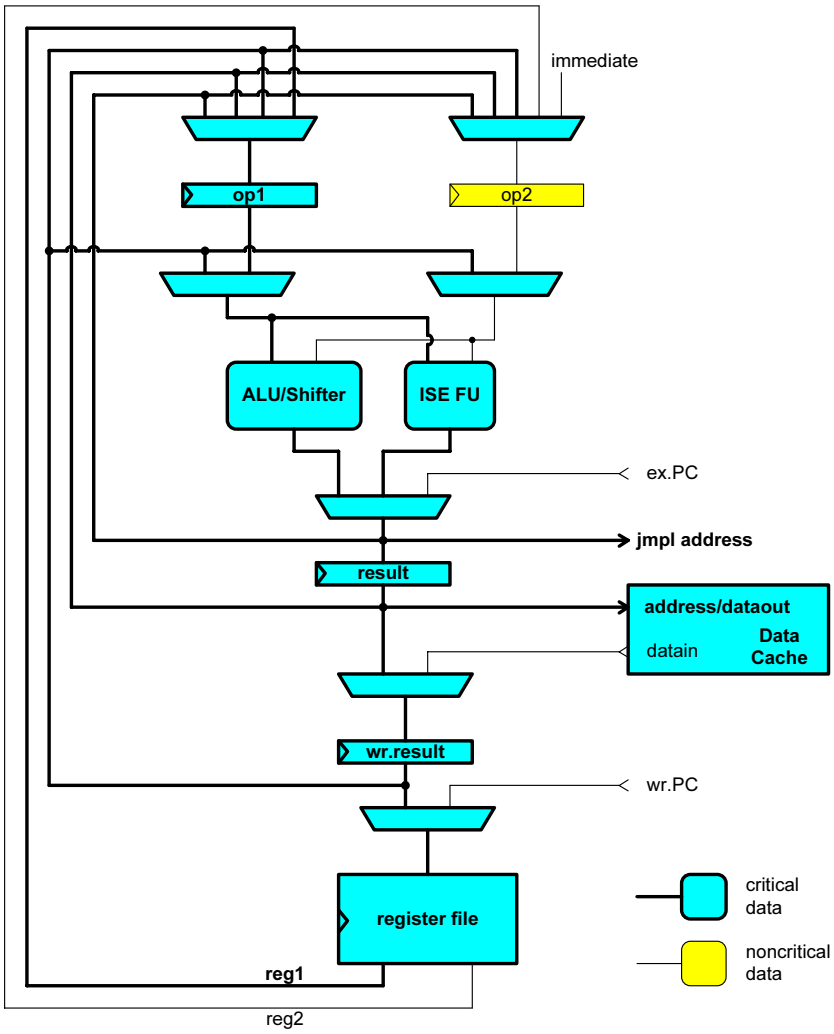

Figure 12.1: Typical datapath of a processor with instruction set extensions.

whenever the ISE FU is active. Secondly, additional multiplexers in the feedback paths and output paths allow to prevent propagation of critical data. All multiplexers which suppress the propagation of the critical value related to *reg1* are marked with dashed lines. Through these simple measures, the portion of the datapath carrying critical data (and hence requiring side-channel analysis countermeasures) is reduced significantly in comparison to the datapath in Figure 12.1.

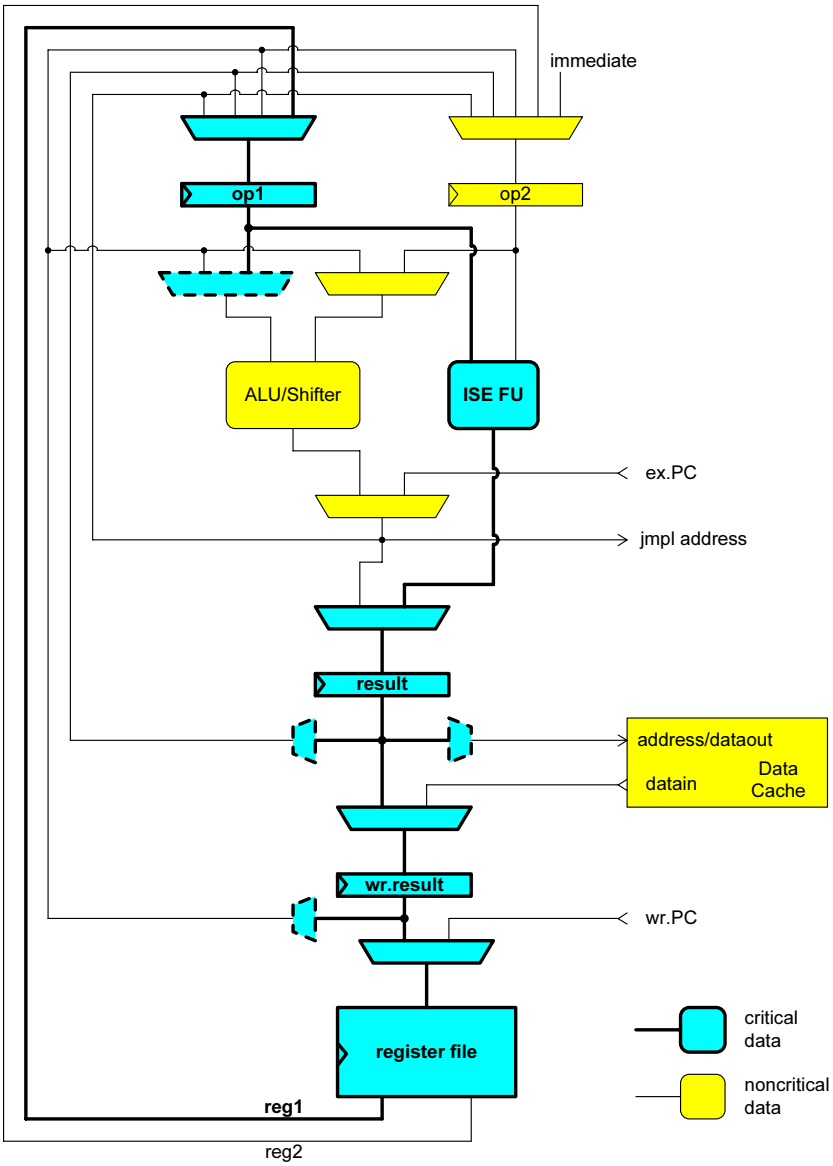

Figure 12.2: Modified processor datapath to suppress propagation of critical values.

12.1.1 Proposed Countermeasures

Based on the analysis of the critical sections of a typical processor datapath, we examine three different strategies to increase the resistance against side-channel attacks in Sections 12.2 to 12.4. These three solutions differ with respect to the ease of implementation, the level of security, and the impact on the critical path,

area, and power consumption. These properties are examined in Section 12.5.

12.2 Complete Datapath in Secure Logic

The straightforward way to secure the processor containing the extended datapath shown in Figure 12.2 is to implement all critical parts in a DPA-resistant logic style. For example, the circuit could be built using Wave Dynamic Differential Logic (WDDL), as proposed by Tiri and Verbauwhede in [246]. This solution is applicable to secure all types of cryptographic instruction set extensions, provided that the critical values do not leave the secured datapath, e.g., are stored to unprotected cache or memory.

Note that it is also necessary to include the register file in the secure implementation. As modern embedded RISC processors can have a large number of registers, securing the register file can become very costly in terms of silicon area. For example, SPARC V8 processors feature a number of so-called register windows. Each such window features 24 individual 32-bit registers and there are eight additional registers shared between all windows. In order to limit the overhead in terms of area, it is possible to implement only a part of the registers in a secure logic style, and restrict critical values to these registers.

12.3 Random Precharging

Another measure to increase resistance against power analysis is to randomly charge the datapath before and after a critical value is processed (precharge, postcharge). This measure can be implemented solely by careful modification of the cryptographic software, but delivers at best a moderate increase in security. A similar countermeasure was proposed in [36] for hardware implementations.

Random precharging can be realized in software by prefixing and suffixing each instruction that processes critical data with the same instruction using random operand values. As the same instruction is used, the same paths will be active and loaded with random values. An example of this is given for the `sbox4s` instruction (see also Section 7.2.4) in Figure 12.3.

We show in this section that for the critical instructions for AES, the result of the execute stage is a uniformly distributed random value if both operands are also uniformly distributed. This property is necessary to ensure that all vulnerable paths after the functional unit are also charged with uniformly distributed random values.

The random charging of the datapath before and after critical instructions can, of course, not completely prevent side-channel leakage. Nevertheless, the switching activity for the critical values is randomized to a certain degree. When the preceding value of a bit line is randomized, then the transition characteristics with respect to a fixed second value will also be randomized. When the preceding value of a critical value was fixed to 0 and becomes uniformly distributed, the switching characteristics change accordingly:

```
! pre: Operands for sbox4s in %o1 and %o2
!      Random precharge values stored in memory
!      starting at address 'rand'
! post: S-box output in %o0

! Load random values
ld [rand], %o3
ld [rand+4], %o4
ld [rand+8], %o5
ld [rand+12], %o6

; Precharge
sbox4s %o3, %o4, %o0

; Actual S-box lookup
sbox4s %o1, %o2, %o0

; Postcharge
; Result discarded (write to %g0)
sbox4s %o5, %o6, %g0
```

Figure 12.3: S-box transformation using `sbox4s` instruction with random precharging.

- $0 \rightarrow 0$ transitions become $0 \rightarrow 0$ or $1 \rightarrow 0$ transitions.
- $0 \rightarrow 1$ transitions become $0 \rightarrow 1$ or $1 \rightarrow 1$ transitions.

A similar situation occurs when a preceding value fixed to 1 gets randomized:

- $1 \rightarrow 0$ transitions become $1 \rightarrow 0$ or $0 \rightarrow 0$ transitions.
- $1 \rightarrow 1$ transitions become $1 \rightarrow 1$ or $0 \rightarrow 0$ transitions.

Without loss of generality, we can limit our analysis to the first case. We can observe that a 0 for the second (i.e., the critical) value will result in a $0 \rightarrow 0$ or $1 \rightarrow 0$ transition with equal probability. If the critical value is 1, then a $0 \rightarrow 1$ or $1 \rightarrow 1$ transition can occur. An attacker who correctly predicts a 0 or 1 of the critical value will have to distinguish the effects of $0 \rightarrow 0$ and $1 \rightarrow 0$ transitions from $0 \rightarrow 1$ and $1 \rightarrow 1$ transitions. When no random precharging is employed, it is sufficient to distinguish $0 \rightarrow 0$ from $0 \rightarrow 1$ transitions, which is generally much easier. By the same argument, an attacker has to distinguish $0 \rightarrow 0$ and $0 \rightarrow 1$ transitions from $1 \rightarrow 0$ and $1 \rightarrow 1$ transitions for an instruction with random operands occurring after the attacked instruction.

We now examine the case for AES instruction set extensions in more depth. We use the high-performance extensions presented in Section 7.2.4, namely the

instruction families `sbox4s/isbox4s/sbox4r` and `mixcol4s/imixcol4s`. The `sbox4s` instruction takes two bytes each from the first and second operand and substitutes them according to the AES S-box. The `isbox4s` instruction does the exact same with the inverse S-box. The `sbox4r` instruction substitutes the four bytes of the first operand, followed by a rotation to the left of the result by one byte. For `mixcol4s`, an AES State column is assembled from two bytes of each of the operands. This column is transformed according to the AES MixColumns operation, yielding the corresponding output column of the AES State. The `imixcol4s` instruction does the same with the AES InvMixColumns transformation.

These two families of instructions are almost sufficient to implement manipulations of critical data for the whole AES algorithm. The only additional instruction required is the `xor` instruction.

One important point to show is that it suffices to execute a custom instruction with uniformly distributed random operands to charge every node in the extended datapath to a uniformly distributed random value. To this end, we have to examine the result for each extended instruction with random operands:

- `sbox4s/isbox4s/sbox4r`: The input to the ISE functional unit is assembled and calculated from independent bytes of the two operands. The AES S-box itself is a bijective mapping. Therefore, two uniformly distributed random operands produce a uniformly distributed result.
- `mixcol4s/imixcol4s`: The input to the ISE FU is again assembled from independent bytes of the operands. For each byte of the resulting output column, the four bytes of the input column are linearly transformed (multiplied by a constant in $GF(2^8)$), and then added (XORed) together. Both does not change the uniform distribution of the input operands.
- `xor`: An XOR of two uniformly distributed operands, yields a uniformly distributed result.

12.3.1 Further Implementation Details

It is important to know which instructions process critical data with respect to power analysis and therefore need to be precharged. Generally, these are all instructions which process data that depends on the plaintext or ciphertext and a small portion of one or more round keys. We consider the initial AddRoundKey, all four transformations of the first round, and SubBytes of the second round as potentially vulnerable. At the end of the AES algorithm, all transformations of round nine and ten need to be protected. The intermediate values in the middle rounds depend on five or more bytes of the cipher key, thus requiring at least 2^{40} hypotheses for a DPA attack. We consider this level of protection to be sufficient for most applications. However, it is not difficult to extend the protection to further rounds.

As we have already noted in Section 12.1, we want to avoid the use of feedback paths for critical data whenever possible. In our example of the 4-stage data

pipeline, we can prevent the propagation of critical values via feedback paths by inserting at least two unrelated instructions between the instruction generating the critical value and the first instruction using it as an operand. In the case of the examined AES instructions set extensions, this criterion can be fulfilled with no or little performance loss, by rearranging the instructions in an assembly implementation.

For the examined AES extensions and the previously outlined degree of protection, it is sufficient to protect 20 instructions at the beginning and 20 instructions at the end of the algorithm. This means that a total number of 160 random 32-bit words (i.e., 640 random bytes) are required for each block cipher call. Depending on the implementation of the random number generator, they could be generated on-the-fly or loaded from a pregenerated array of random values in memory. Loading from memory should be done in a consecutive manner, so that only the Hamming distances of different random values leak. This way, the threat of second-order DPA attacks can be alleviated. The number of required random 32-bit words could be reduced in an elegant way through the use of the AES extensions themselves. Once a random 32-bit word has been used for precharging, its four bytes could be transformed using the AES S-box (e.g., with the `sbox4s` instruction), yielding a new 32-bit word suited for precharging. For an attacker who does not know the input word, the new word would appear equally unpredictable.

12.4 Protected Mask Unit

In this section we present another method for securing cryptographic instruction set extensions based on the application of secure logic styles. The basic idea is to split the processor into an insecure and a (relatively small) secure zone[1]. The hardware of the insecure zone is left unchanged, e.g. it remains in standard CMOS. Protection of critical data in the insecure zone is achieved by the use of Boolean masks. Critical values always appear with a Boolean mask. This mask is never reused and does not occur separately in the insecure zone. First-order DPA attacks are prevented by the mask. Second-order and higher-order DPA attacks cannot be performed, as there is no second data value for preprocessing available. The secure zone contains storage for the mask currently used in the insecure zone, a mask generator and the functional units for the extensions. The secure zone is implemented in a secure logic style in order to protect the masks. Note that only in the secure zone the masks only occur separated from the values they protect.

Figure 12.4 depicts the principal structure of the secure zone. The register addresses of the masked operands $op1_m$ and $op2_m$ select the corresponding masks from the storage section. Then the operands are unmasked and processed by the functional unit, yielding the result res. The generator produces a new random mask and applies it to the result res before it leaves the secure zone. This new

[1] The terms insecure and secure refer to the protection offered by the hardware alone.

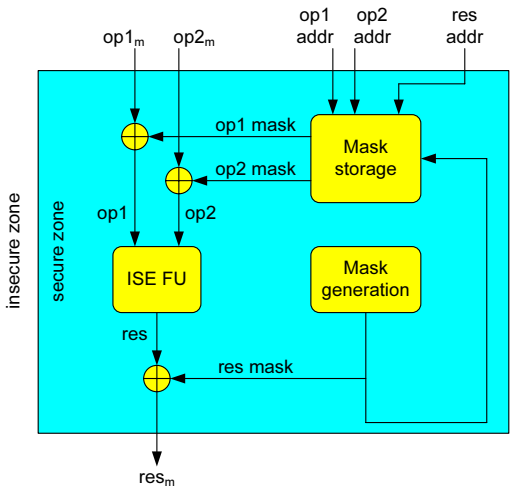

Figure 12.4: Secure zone of the processor.

mask is also saved in the mask storage for the register which receives res_m. Outside of the secure zone, the critical values remain masked and can therefore be handled without restrictions, e.g., stored to cache and memory. The insecure zone does not require any special architectural modifications or a more careful implementation in comparison with the corresponding part of the unprotected processor.

This countermeasure confines the processing of the unmasked critical values almost exclusively to the secure zone, as these values retain their Boolean mask in the insecure zone. An important exception from this rule is the XOR operation of a masked value (u_m) with an unmasked value (v), as this does not change the mask: $u_m \oplus v = u \oplus m \oplus v = (u \oplus v) \oplus m = (u \oplus v)_m$. The AddRoundKey operation of AES is an example where the State is the masked value and the corresponding round key is the unmasked value. Note that the round key is not dependent on the plaintext or ciphertext and is therefore not directly vulnerable to a DPA attack. Template attacks which target the round key are believed to be infeasible on 32-bit processors (see Chapter 11 for further details).

In the case of the AES extensions, it is sufficient to include the functional units for the `sbox4s` and `mixcol4s` instruction families in the secure zone. The critical values can be confined to seven 32-bit registers during execution of the AES algorithm, so the mask storage unit only needs to be able to store seven 32-bit masks. SubBytes, ShiftRows, and MixColumns would then be performed unmasked in the secure zone, while AddRoundKey could be done using the masked State in the insecure zone.

The registers whose masks are held in the mask storage could either be fixed or dynamically changed with the help of a tiny content-addressable memory (CAM), similar to a cache. A mechanism for masking plaintext and unmasking ciphertext is also required, for example, a dedicated instruction.

The cycle count for AES operations would remain almost the same for a protection solution with a mask unit (requiring just a few additional cycles for masking at the beginning and unmasking at the end). The architecture of the ISE FU can also remain unchanged since the critical values are processed unmasked. A drawback of the solution with the mask unit is that it introduces a resource (namely the mask storage unit) which can only be used by a single processor task. To share this mask unit between tasks, its content needs to be saved on a task switch, e.g., in memory. In this way, the masks would leave the secure zone and open up a potential vulnerability against second-order DPA attacks, as now both the masked value and the corresponding mask would appear in the insecure zone. However, it might still be possible to save the contents of the mask storage into the unprotected memory on a task switch, without impairing the overall security. If an attacker cannot control or predict task switches, then it would be very hard to observe leakage related to specific critical values. Such functionality could be implemented both in the processor's hardware or in the operating system.

12.5 Security and Performance Analysis

In this section we compare the three proposed solutions with respect to a number of implementation aspects, whereby we try to keep the scope as broad as possible. However, for some metrics it is necessary to take certain design choices into account. We have selected WDDL [245, 246] as an example of a secure logic style. All figures for silicon area, critical path, and power consumption are given for an implementation based on a 0.13 µm standard-cell library. The cycle count refers to an AES-128 encryption using instruction set extensions.

12.5.1 Applicability and Implementation Complexity

Implementing the complete datapath in secure logic (cf. Section 12.2) can be applied to any kind of instruction set extension. It requires a careful partitioning of the processor into an insecure and a secure part as well as the implementation of the latter in a secure logic style. The software only needs to be modified to restrict critical operations to the secure datapath. Random precharging (cf. Section 12.3) is also generally applicable to cryptographic instruction set extensions. Being a software countermeasure, it can be flexibly adapted to protect specific operations which are deemed vulnerable. The hardware can be left unchanged. The solution with a protected mask unit (cf. Section 12.4) is only applicable if almost all critical operations can be limited to the secure zone. Its implementation is easier than a complete datapath in secure logic, as the boundary between insecure and secure zone is simple and well-defined. Changes to the software are not necessary, apart from explicit masking and unmasking operations at the beginning and end of the cryptographic operation.

12.5.2 Security

We express security as the factor by which the number of power traces necessary for mounting a successful DPA attack increases[2]. For the solutions employing a secure logic style (cf. Sections 12.2 and 12.4), this factor solely depends on the security of the chosen logic style. For a careful implementation of WDDL in an ASIC, an empirical evaluation has shown this factor to be over 750 [245].

In order to assess the security of random precharging (cf. Sections 12.3), we have compared an unprotected and a protected AES software implementation. We have prototyped the LEON2-CIS processor with AES extensions on an FPGA-board designed for power measurements [214, 215]. Then, we have performed a DPA attack on both AES software implementations using 100,000 power traces. Although the absolute results may differ from those of an ASIC implementation, the relative comparison gives an indicator for the security improvement. Figures 12.5 and 12.6 show the result for the DPA attack on the unprotected and the protected AES implementation, respectively.

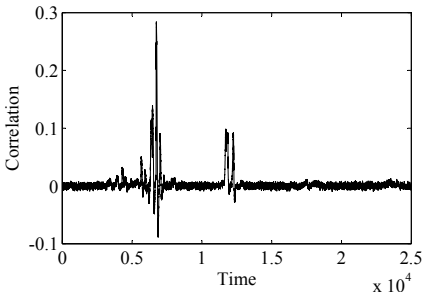

Figure 12.5: Result of DPA attack on unprotected AES implementation.

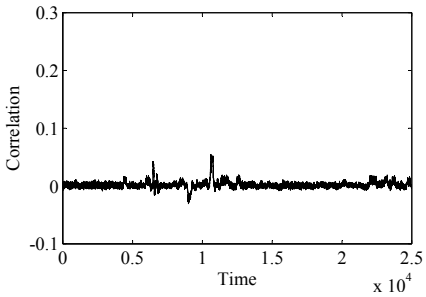

Figure 12.6: Result of DPA attack on AES implementation with random precharging.

The empirical results indicate that the correlation coefficient ρ can be reduced significantly from 0.284 to 0.055. Therefore, the maximum correlation is reduced by a factor of 5.16, which corresponds to an increase of the number of required power traces by a factor of over 26 [166].

12.5.3 Performance

Table 12.1 shows how the different solutions impact the performance of a single AES-128 encryption. Using a secure logic style such as WDDL requires an extra precharge cycle for every operation performed in the secure part. This concerns every instruction when the complete datapath is in secure logic, and therefore leads to a doubling of the cycle count. For the protected mask unit, we assume that only the critical instructions—which constitute only a small fraction of the total instructions—require an extra precharge cycle for the secure logic. Random

[2]Note however that side-channel attacks remain a major research topic and that no evaluation of a countermeasure can so far guarantee any fixed increase in absolute security.

Table 12.1: Cycle count for AES-128 encryption.

Implementation	**Performance** cycles	**Overhead** %
Baseline implementation [237]	196	0%
Complete datapath in secure logic	392	100%
Random precharging	~ 600	$\sim 206\%$
Protected mask unit	~ 230	$\sim 17\%$

precharging in software leads to a higher overhead as it needs one precharge and one postcharge instruction per critical instruction, and also extra instructions to load or generate the random precharge values.

In summary, the cycle count for our protected implementations ranges approximately between 230 and 600 cycles for a single 128-bit AES encryption. An unprotected implementation on a processor without instruction set extensions requires about 1,637 cycles (cf. Chapter 11). When compared to the overhead of the software countermeasures introduced in [243], which ranges between a factor of ten and 100, our proposed solutions remain very fast.

12.5.4 Implementation Overhead

For the random precharging solution, the hardware—and therefore also silicon area, critical path and power consumption—remain unchanged. For the other two options, the employed secure logic style plays a crucial role in determining the overhead for these factors. For a standard-cell implementation using WDDL, the area typically increases by a factor of about 3.2–3.6. The increase of the critical path has been shown to range between 1.07–1.42 while the power consumption increases by a factor of about 3.5 [246]. For this evaluation we will use a factor of 3.5 for area, 1.2 for the critical path, and 3.5 for power consumption. In our case, only a fraction of the total processor needs to be implemented in the secure logic style, which alleviates the total overhead factors.

We now try to give a rough estimation of this overhead for the case of the LEON2-CIS processor. The extended integer unit was reported to have a size of 16,370 gate equivalents (GEs) at a delay of 4 ns (UMC 0.13 µm technology). About half of the integer unit will have to be implemented in the secure logic style for protecting the complete datapath (cf. Section 12.2), requiring about $16,370 \cdot \frac{1}{2} \cdot 2.5 \approx 20,500$ GEs extra[3]. Additionally, the register file also needs to be implemented securely. This will lead to a considerable increase in silicon area due to two reasons: Firstly, it will be necessary to change the register file implementation from a custom memory block to flip-flops. Secondly, if WDDL flip-flops are used, the area for a single flip-flop is increased by at least a factor of 4 [246]. The size of a 32-bit CMOS register is about 235 GEs, and therefore a 32-bit WDDL register costs about 940 GEs. Depending on the number R of

[3]Note that if the size increases by 3.5, the extra area is $(3.5 - 1) = 2.5$ times the original area.

secured registers, we get an area of about $940R$ GEs for the secure register file. For example, a complete register window of a SPARC V8 processor consisting of 32 registers would cost about 30,000 GEs for secure implementation. This illustrates that the complete datapath in secure logic is very costly in terms of silicon area. Taking our estimated overhead factor of 1.2, the critical path will increase to 4.8 ns.

For the protected mask unit, the secure part of the circuit mainly consists of three parts: The ISE functional unit (FU), the mask storage, and the mask generator. The area of the ISE FU is about 3,000 GEs [237]. The mask storage includes seven 32-bit registers, which requires about 1,700 GEs. For the mask generator, we estimate the size of a radix-32 version of the key generator of the Trivium stream cipher, based on the figures reported in [108]. Such a circuit would require about 5,500 GEs and will be able to deliver 32 fresh pseudo-random bits per cycle. Taking into account circa 1,000 GEs for additional circuitry (e.g., XOR gates), the total overhead would be $(3,000 + 1,700 + 5,500 + 1,000) \cdot 2.5 = 28,000$ GEs. The critical path through the ISE FU would also be slightly increased by the XOR stages and the decode logic in the mask storage unit (see Figure 12.4), so that it can be estimated to about 5 ns when compared to the complete datapath in secure logic. Table 12.2 summarizes the overhead estimations. It should be noted that these figures should only give a rough idea of the magnitude of overhead.

Table 12.2: Estimation of overhead for silicon area and critical path.

Implementation	**Silicon Area**[4] GE	**Critical Path** ns
Complete datapath in secure logic	$+ (20,500 + 940R)$	+ 0.8 (+ 20%)
Random precharging	–	–
Protected mask unit	+ 28,000	+ 1.0 (+ 25%)

12.5.5 Combination with Other Countermeasures

In Chapter 11, software countermeasures were proposed to secure AES implementations using instruction set extensions. Since implementing the complete datapath in secure logic is an orthogonal solution, both countermeasures could be directly combined to increase implementation security further. Both random precharging and the protected mask unit can be combined with the countermeasures of [243] to a certain degree (e.g., randomization of operations), which also leads to a further increase in resistance against power analysis attacks.

[4] R denotes the number of secured 32-bit registers.

12.6 Summary and Conclusions

In this chapter we have investigated three different approaches for increasing the DPA-resistance of AES software implementations executed on a 32-bit RISC processor with cryptography extensions. The first approach is to implement all security-critical parts of the processor datapath—including (parts of) the register file and functional units—using a DPA-resistant logic style. This approach comes at a high cost in terms of area since a significant portion of the processor has to be implemented in DPA-resistant logic. The security depends primarily on the security of the used logic style.

The second approach can be realized as a software countermeasure and uses random precharging at the instruction level. A major advantage of this approach is that it does not require any modifications of the processor hardware. However, the achievable security gain is rather moderate. Furthermore, random precharging also increases execution time. Due to the instruction set extensions the performance of the protected implementation is still significantly higher than that of an unprotected implementation on a conventional SPARC V8 processor (600 cycles vs. 1,637 cycles).

The third approach involves both hardware and software. It is based on the use of a secure mask unit. This approach splits the processor into a secure zone and an insecure zone, whereby the secure zone—containing storage for the mask, a mask generator, and the functional units for the extensions—is protected with a DPA-resistant logic style. The hardware cost of this approach has been estimated to be 28,000 GEs, which is significantly less than the hardware cost of the first approach. Similar to the first approach, the exact security and additional area depends on the properties of the used DPA-resistant logic style. The benefit of this approach is that it has only a minimal impact on performance, and that it does not impose any restrictions on the operations outside of the secure zone.

In summary, the side-channel countermeasures discussed in this paper enable different tradeoffs between performance, hardware cost and security, which are not achievable with pure software or pure hardware implementations.

13

Practical Evaluation of State-of-the-Art Software Countermeasures for AES

In Chapter 11, software countermeasures against power analysis attacks have been evaluated in a more or less theoretical approach. New combinations of existing attacks have been proposed and evaluated. In this chapter, we examine the practicability of these proposed methods by applying them to break a protected AES software implementation on a programmable smart card. We demonstrate with two different successful attacks that they indeed pose a security threat. Our results also allow comparison of the effectiveness in relation to traditional DPA attacks on unprotected implementations. We also refine and improve the original attacks, so that they can be mounted more efficiently. Our practical results indicate that the effort required for attacking the protected implementation with the examined methods is more than two orders of magnitude higher compared to an attack on an unprotected implementation.

Note that we did not have the goal to develop new attacks on these countermeasures. Furthermore, we stress that—as with any practical evaluation—our results may not necessarily be optimal for the targeted device. Therefore, the increase in the number of required power traces for the protected implementation indicated in Section 13.4 should not be taken as a fixed "security gain factor" but only as an upper border of this factor.

13.1 Protected AES Software Implementation

The protection of our AES software implementation is based on the strategy of combining masking, shuffling and dummy operations as published in [115]. At

the beginning and at the end of the AES operations there are so-called *randomization zones*. Within each zone all intermediate values are protected with a single mask and the sequence of processing of the bytes of the State is randomized. In [115], the initial zone extends to the first MixColumns transformation. However, Jaffe showed that it is possible to attack the AES after the MixColumns operation [138]. This principally means that a protection of the first round alone would be insufficient. Nevertheless, in Chapter 11 we showed that it is quite easy to extend this randomization zone beyond the second SubBytes operation [243], thus thwarting Jaffe's attack[1]. Note that in our implementation we are only examining the operations until the first MixColumns operation because our attacks are mounted on the output of the first SubBytes transformation, and therefore it is sufficient to protect this part of the AES.

We are using the same masking scheme with six different random masks as published in [115]. One mask is used for S-box inputs (M), one for S-box outputs (M'), and one for each State row before MixColumns (M1, M2, M3, and M4). All other occurring masks (e.g., after MixColumns: M1', M2', M3', and M4') are derived from these six masks. All intermediate values of the State and the key are masked. The masks used during the steps of AES encryption are shown graphically in Figure 13.1.

In our implementation, the randomization is achieved by shuffling of the sequence of operations on the bytes of the State. Furthermore, it would be possible to process additional dummy States to increase the degree of randomization. The randomization is controlled by random values which are—similarly to the mask bytes—unknown to the attacker. We denote the degree of randomization with R, i.e., the number of points in time where a specific State byte can be processed during a specific AES encryption. In our case, the 16 bytes of the AES State are fully shuffled and no dummy States are added. This means that the randomization degree $R = 16$. Hence, a specific byte of the State will be processed with a probability of $p = \frac{1}{16}$ at one of the 16 possible points in time. One protected AES encryption requires less than 12,000 clock cycles on our targeted platform, which is described in Section 13.3. The cost per additional dummy State would be about 1,000 clock cycles.

The consequences of this randomization for a plain first-order DPA attack are illustrated in the following. In Figure 13.2, the result of an attack on the first S-box output of an unmasked and randomized implementation is shown. Note that only the section of the trace which corresponds to the SubBytes transformation was used. One can clearly identify four groups of peaks. When zooming into one of these groups, again four pairs of peaks can be identified. Each group of peaks corresponds to the transformation of a single State column, which again encompasses four State bytes. As a specific State byte can occur at any of these times, there are all in all 16 different positions where the output value of the unmasked S-box correlates. Furthermore if we compare the achieved correlation of this attack to the achieved correlation of an attack on the unprotected im-

[1] The randomization zone at the end can be extended in a similar fashion to protect against Jaffe's attack.

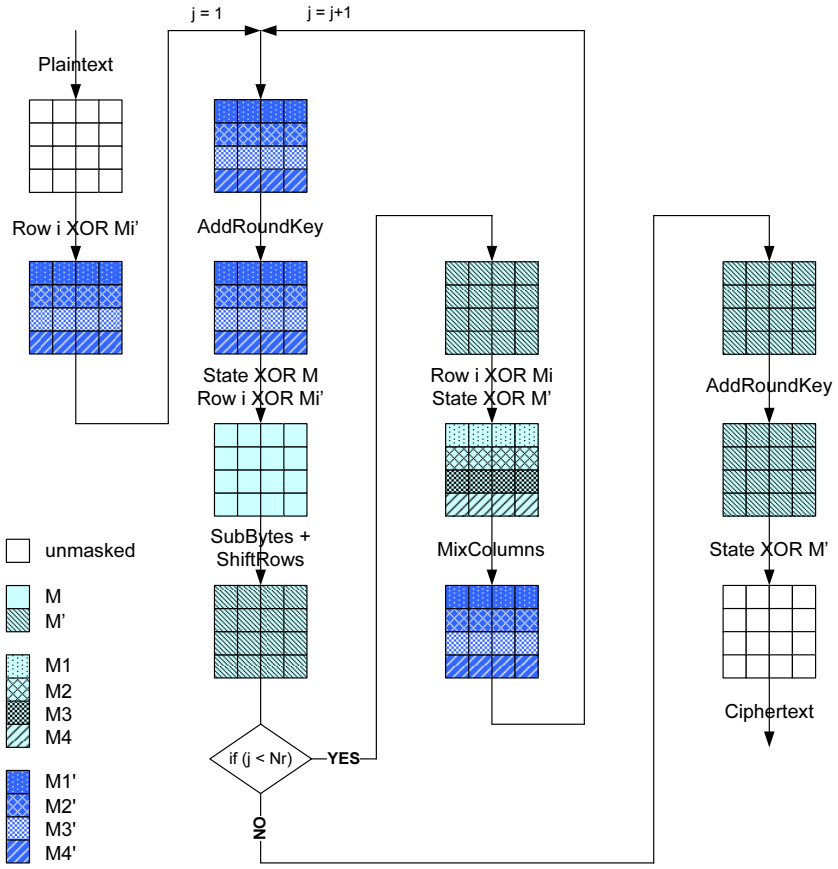

Figure 13.1: Masks used in protected AES encryption.

plementation (cf. also Figure 13.6 in Section 13.3) we end up with a correlation which is reduced by a factor of approximately 16. As expected, the correlation coefficient scales down linearly with the degree of randomization R for a simple first-order DPA attack [166].

13.2 Description of the Advanced DPA Attacks

From an implementor's view, a combination of masking and operation randomization countermeasures is a good bet for protecting software implementations of secret-key cryptographic algorithms against power analysis. Proper masking can prevent first-order DPA attacks while the randomization of operations in time can offer some protection against more elaborate attacks like higher-order DPA and template-based methods. At the same time, the implementation complexity and overhead can be kept within somewhat acceptable limits.

Higher-order DPA attacks and template attacks have been shown to be

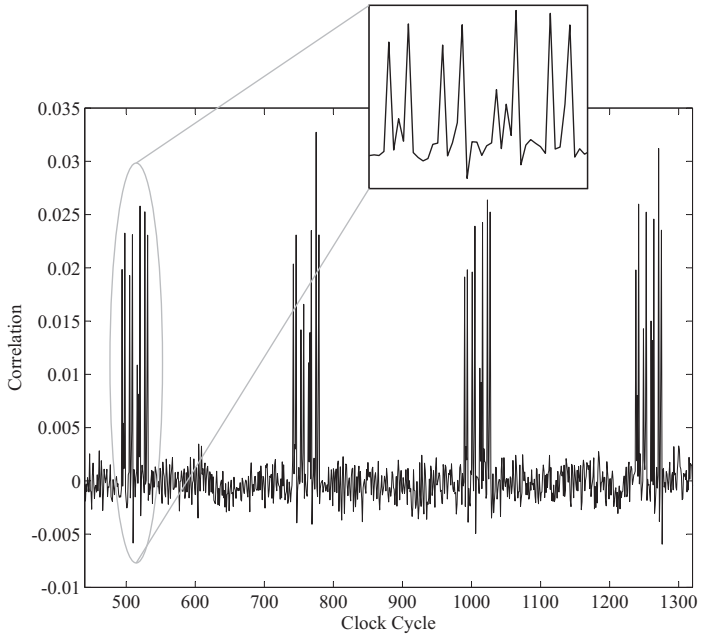

Figure 13.2: Result of DPA attack on randomized AES implementation using the Hamming weight model and compressed traces.

very effective to circumvent masking. In higher-order DPA attacks, power leakage of several intermediate values is combined in such a way that the resulting power consumption value is again dependent on the original unmasked value [140, 191, 226, 251]. Template-based methods can be applied to enhance higher-order DPA attacks or to make first-order DPA attacks feasible [166, 190]. On the other side, the technique known as *windowing* is a good way to limit the protection of randomization of operations [52]. In this method, all R possibilities of appearance of a protected value are considered and combined so that the effective protection of the countermeasure is lowered.

We have demonstrated in Chapter 11 that the attacks on masking and randomization can be combined to form effective attacks on implementations which employ a single mask and which randomize the course of operations. Three possible combinations have been presented and their effectiveness has been compared by estimations techniques and high-level simulation which neglected most of the noise components.

Figure 13.3 shows the timeline for a part of the execution of a protected secret-key cipher implementation. Towards the beginning at time index t_0, the mask m is processed in some form (it is generated, stored, used in some precomputation, etc.). Subsequently, the intermediate value a masked with m appears. The occurrence of this masked intermediate value is protected by randomiza-

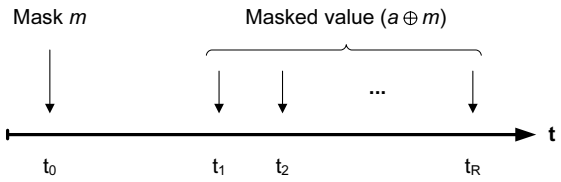

Figure 13.3: Points in time relevant for an attack.

tion, i.e., in each execution the specific value $a \oplus m$ can appear in any of the R points at the times $t_1..t_R$ with equal probability $p = \frac{1}{R}$. The power consumption at these points in time is used in all attacks described in Chapter 11. An adversary must therefore be able to find these points in time. It will be shown in Section 13.5 that the constraints for $t_1..t_R$ can be relaxed, so that knowledge of the exact time indices is not necessary. The attacks are summarized in the following.

13.2.1 Biasing Masks and Windowing

One precondition for effective masking is that the used masks must be uniformly distributed. If this condition is not met, first-order DPA attacks can become feasible. Therefore, the principal idea of this attack is to try to determine the Hamming weight of the used mask and to select a subset of the collected power traces, where the mask fulfills some property, e.g., has a high Hamming weight. This selection of power traces effectively equals a bias which is introduced into the distribution of the mask values. The selected power traces are then only protected by the randomization of the operations ($t_1..t_R$), whose effect can be minimized by windowing. A subsequent first-order DPA attack can be successfully applied on the selected power traces. This attack has been shown to be rather effective under most circumstances in Chapter 11.

13.2.2 Second-Order DPA Followed by Windowing

The idea of this attack is to take the randomization of the operations into account during second-order DPA preprocessing. The power consumption values for m at t_0 and $a \oplus m$ at $t_1..t_R$ are pairwise combined (second-order DPA preprocessing). For each pair, this preprocessing results in a joint leakage value of the two points. If the correct points in time have been chosen, one of these R joint leakage points is always dependent on the actual unmasked value. Which of these points is the correct one is determined by the randomized course of operations of the corresponding execution. When all points are windowed (i.e., summed up), the correct one is inevitably included and the resulting power consumption is to some degree dependent on the unmasked value. A first-order DPA attack can then be mounted to determine the correct key hypothesis. This attack has been evaluated in Chapter 11 to be less effective than the first one in most cases.

13.2.3 Windowing Followed by Second-Order DPA

A third option for attacking is to reverse the order of windowing and second-order DPA preprocessing. First, randomization is compensated by summing up all R points in time where the targeted masked value $a \oplus m$ can occur. Then, second-order DPA preprocessing is performed with this sum and a point of the power trace which depends on the corresponding mask m. The result of this preprocessing can be attacked with a first-order DPA attack. In Chapter 11, this attack variant has been shown to be rather ineffective in comparison to the previous two methods.

13.3 Attacked Device

The device used to implement and attack the protected AES software implementation is a smart card with an ATMega163 core [14]. The ATMega163 is an 8-bit microcontroller based on AVR, which is a single-cycle instruction RISC architecture from Atmel. It is equipped with 1,024 bytes of internal RAM, 16 KB in-system self-programmable FLASH and 512 bytes of EEPROM. The core of the controller contains 32 general-purpose registers which are directly connected to the arithmetic-logic unit (ALU). Three register pairs can be used to store a 16-bit address for operation on the internal memory.

The used smart card is shipped without any software or operating system. This means the card is under full control of the designer and all parts of the software (including boot code and operating system) have to be implemented from scratch. In our scenario, there is only a minimum version of an operating system implemented which can handle the basic functions of the T=1 protocol specified in ISO 7816 [132]. The card can execute the protected AES implementation described in Section 13.1. For the sake of performance, the randomization parameters and mask bytes are sent to the smart card along with the plaintext. The key used for the AES encryption is stored in the EEPROM of the smart card.

The following paragraph gives a brief overview of the leakage behavior of the used device. In general, the device leaks the Hamming weight as well as the Hamming distance of the processed values. When attacking the non-randomized S-box output using the Hamming weight model, the maximum achievable correlation is 0.458. In a similar attack using the Hamming distance between the S-box input and the S-box output—which occur as subsequent values of a register—the maximum correlation is 0.257. The results for these attacks on uncompressed traces are displayed in Figure 13.4 and in Figure 13.5. It can be seen that for this device and the sequence of instructions used to implement the S-box lookup, the Hamming weight model leads to a higher correlation.

For minimization of the amount of data used for an attack, it is a common technique to compress the measured traces [166]. When using the compression described in Section 13.4.2, the achievable correlation using the Hamming weight model reduces from 0.458 to 0.383. With the Hamming distance model the

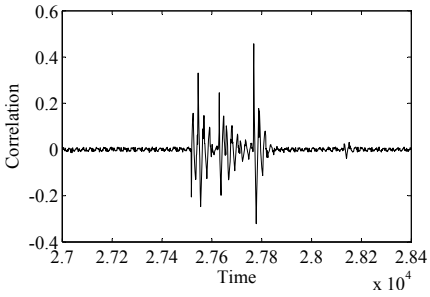

Figure 13.4: Result of DPA attack on unprotected AES implementation using the Hamming weight model and uncompressed traces.

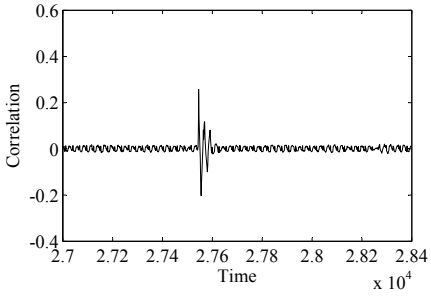

Figure 13.5: Result of DPA attack on unprotected AES implementation using the Hamming distance model and uncompressed traces.

correlation reduces from 0.257 to 0.236. The results of an attack on compressed traces can be seen in Figure 13.6 and Figure 13.7. For our practical attacks, this means that most of the information is preserved in the compressed power traces.

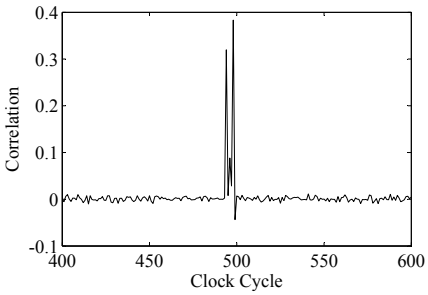

Figure 13.6: Result of DPA attack on unprotected AES implementation using the Hamming weight model and compressed traces.

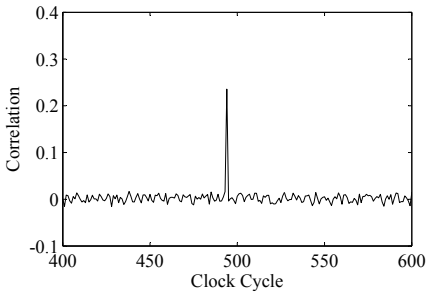

Figure 13.7: Result of DPA attack on unprotected AES implementation using the Hamming distance model and compressed traces.

13.4 Practical Results

In order to demonstrate the effectiveness and practicability of the methods described in Chapter 11, we have attacked the protected AES software implementation presented in Section 13.1. For the randomization of operations, we have used full shuffling of the 16 State bytes, i.e., $R = 16$. Power traces were collected with a LeCroy LC584AM digital oscilloscope and a differential probe by measurement over a $1\,\Omega$ resistor in the ground line of the smart card reader. A trigger signal has been supplied by the smart card at the beginning of encryption.

We have collected a set of 500,000 power traces, which took about 134 hours in our measurement setup, i.e., a rate of approximately one trace per second. The uncompressed traces required about 50 GB of disk space. For comparison, a set of compressed traces was between 700 MB and 2 GB in size, depending on the actual compression function. An uncompressed trace contained 100,000 points, whereas a compressed trace consisted of about 1,800 points (one per clock cycle).

The power traces included about 1,800 clock cycles at the start of AES encryption spanning over various precomputations (parts of the masked key scheduling, mask preprocessing, and the masking of the plaintext), the initial AddRoundKey, and the first AES round. In order to keep the size of the traces small, the sampling rate has been limited to $200 \cdot 10^6$ samples/second.

All statistical analyses were carried out on a PC featuring a quad-core Intel Xeon processor at 2.33 GHz and with 8 GB of RAM. Attack times were generally determined by the number and size of the analyzed traces, and not by the kind of statistical analysis. An attack using all 500,000 power traces took about 140 s for compressed traces and about 1,7 hours for uncompressed traces. For an attack with biased masks, the template-building took about 160 s when 100 traces were used for each of the nine templates. The time for attacks involving fewer traces would scale down almost linearly.

For our attacks we have used the S-box output of the first round as intermediate value a and the S-box output mask as corresponding mask m. The time indices $t_1..t_{16}$ for $a \oplus m$ were determined by first-order DPA attacks using the known masked S-box outputs as attacked intermediate values. Suitable indices t_0 were found accordingly by using the S-box output mask as attacked value. For both cases, the time indices resulting in high correlation values were used.

13.4.1 Biasing Masks and Windowing Followed by First-Order DPA Attack

For the purpose introducing a bias in the mask values, we have used templates to estimate the Hamming weight of the mask m. Templates were built from the uncompressed traces for each Hamming weight of the mask. We used 100 traces per template with 16 interesting points per trace. The multivariate Gaussian distribution model has been employed. The traces used for the first-order DPA attack have been compressed (integration of absolute values per clock cycle). Randomization has been countered by windowing, i.e., the power consumption values at times $t_1..t_{16}$ have been summed up. The Hamming weight of an unmasked S-box output byte has been used as predicted power consumption. The attack itself is therefore similar to the one described in [190] ("Templates During Preprocessing"), except for the additional compensation of the randomization countermeasure.

In practice, it is important to find a good tradeoff between a sharp bias and a minimal number of discarded traces. For example, only choosing traces with a mask Hamming weight of 8 (i.e., $m = 0$xFF) will lead to the highest correlation but on the other hand, this would mean to discard $\frac{255}{256} = 99.6\%$ of the recorded

power traces. Selecting masks with a Hamming weight greater or equal to six has been shown to be a good choice [166, 190] and therefore we have also used it for our evaluations.

The effectiveness of the attack depends on the accuracy of the biasing process. In order to show the best outcome, we have also biased the masks following their actual values (ideal case). The results for biasing with the actual Hamming weights and with the Hamming weights predicted by template matching are show in Figures 13.8 and 13.9, respectively.

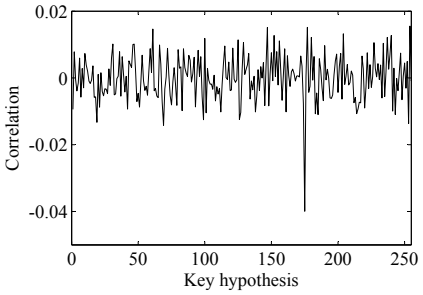

Figure 13.8: Result of attack with ideal mask biasing.

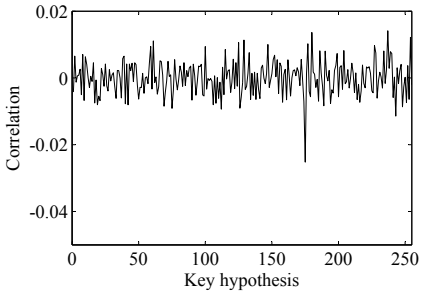

Figure 13.9: Result of attack with mask biasing through templates.

In the ideal case, the correlation for the correct key hypothesis was about -0.04 while the use of template matching yielded a correlation of about -0.025. With increasing accuracy of the template method in predicting the actual Hamming weight of the used mask, the result of the attack should get closer to that of the ideal case. We use the rule of thumb from [166] to estimate the required number of power traces for a successful attack. We have also taken those traces into account which were discarded during the biasing process (about 85% of all traces). For ideal biasing, about 122,000 power trace are sufficient, for our biasing with templates, about 305,000 power traces are required.

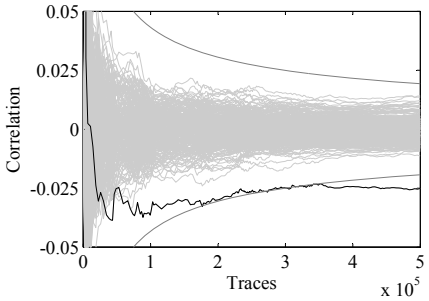

Figure 13.10: Evolution of correlation in dependence on number of traces for mask biasing through templates.

Figure 13.10 shows the evolution of the correlation with increasing number of power traces for the attack depicted in Figure 13.9. Note that the trace count

on the x-axis also includes the discarded traces. The correct key hypothesis is plotted in black, the incorrect hypotheses are displayed in light gray. The outer dark gray lines indicate the confidence interval for $\rho = 0$. Roughly speaking, this is the expected region for incorrect key hypotheses. The point where the correct key hypothesis leaves this region gives another estimation for the number of traces required for a successful attack. In this case, the estimate lies in the vicinity of 300,000 traces, which is in line with the result from the rule of thumb.

13.4.2 Second-Order DPA Followed by Windowing

This attack can be seen as multiple second-order DPA attacks in parallel, with their results combined by windowing. Nevertheless, a successful attack is not quite as simple to achieve as in a conventional second-order DPA attack. Normally, most second-order DPA attacks can be conducted in a more or less "brute-force" manner. More precisely, it is not necessary to determine the exact points in time which carry the most information about the targeted intermediate values and which are therefore suited best for second-order DPA preprocessing. In fact, it is sufficient to predict the general regions of the power traces which are expected to contain the required points. By examining all possible combinations of the points of both regions in a first-order DPA attack, the correct key hypothesis can be identified without giving much thought to the actual points of the power trace which carry the required information.

When there is a need to compensate for the randomization countermeasure as well, it quickly becomes evident that this "brute-force" approach is no longer feasible. Even if all of the R parallel second-order DPA attacks could be done in this manner, the subsequent windowing of the results requires that only those points are summed up which might contain information about the unmasked intermediate values. Our experiments have shown that the attack result is extremely sensitive even to slight variations in time of the two input points to the second-order DPA preprocessing function. Therefore, choosing the best points from the power trace becomes a crucial precondition for windowing. Unfortunately, the best points only become known after a successful attack.

An effective way out of this dilemma can be made with a suitable compression function. If there is only a single point per clock cycle in the power trace, the second-order DPA preprocessing and windowing can be done at the exact points in time where the maximal information is contained. However, care must be taken that not too much information is lost during compression. Our experiments have shown that none of the standard compression functions (maximum extraction, integration) [166] deliver satisfying results. After careful analysis of the second-order DPA leakage profile of the attacked device, we have developed a new compression function, which retains most of the required information and hence delivers good results. Our new compression function extracts a small range of points around the maximum of each clock cycle and forms the average value of those points. At our sampling rate of $200 \cdot 10^6$ samples/second, a range of two points around the maximum (i.e., 5 points in total per clock cycle) was sufficient to achieve satisfying results.

As second-order DPA preprocessing function we have employed the absolute of the difference of the two input points, as it has the best correspondence to single bits of unmasked values and still a good correspondence to the Hamming weight of larger values [166]. We have used the bit model (LSB of the S-box output) as power model for our attack. The results are shown in Figure 13.11.

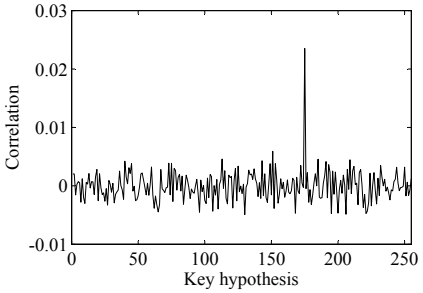

Figure 13.11: Result of second-order DPA followed by windowing.

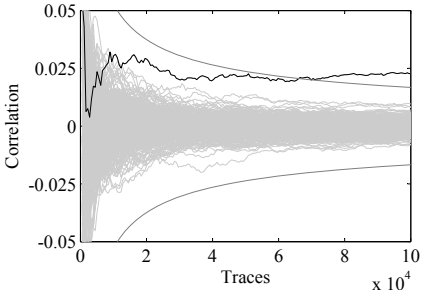

Figure 13.12: Evolution of correlation in dependence on number of traces.

The correlation peak for the correct key hypothesis has a height of approximately 0.024, requiring about 50,000 power traces for a successful attack [166].

Figure 13.12 shows the evolution of the correlation with the number of power traces used in the attack. In this case, the correlation curve for the correct key hypothesis leaves the confidence interval for $\rho = 0$ at about 65,000 traces. Note that this number is a bit higher than the estimate from the rule of thumb from [166]. In our experience, the evolution of the correlation is quite dependent on the measurements, so the rule of thumb should normally be preferred as a more general prediction.

13.4.3 Windowing Followed by Second-Order DPA

This attack had already a very low effectiveness in the simulated evaluation of Chapter 11. For completeness, we have conducted attacks with this method on all 500,000 power traces. As suspected, this number of power traces was not sufficient to lead to a correct prediction of the key.

13.4.4 Dependence of Attack Efficiency on Randomization Degree

The AES implementation allows to change the degree of randomization R in order to trade performance against security. Table 13.1 shows how the effectiveness of the two attacks from Section 13.2.1 and 13.2.2 changes with increasing R. Conceptually, the correlation coefficient should scale down with a factor of $\sqrt{R}$ (cf. Chapter 11). It can be seen from Table 13.1 that both attacks approximately follow this behavior, whereby the second one (second-order DPA followed by windowing), tends to perform worse at a higher R.

Table 13.1: Maximal absolute correlation coefficient in dependence on randomization degree R.

	Randomization degree R				
Attack	1	2	4	8	16
Biasing masks and windowing	0.104	0.072	0.052	0.035	0.025
Second-order DPA and windowing	0.125	0.102	0.073	0.042	0.024

13.5 Discussion of the Practicality of the Attacks

As shown in Section 13.4, two of the three examined attacks succeeded with a reasonable amount of samples. Mask biasing turned out to lead to a potentially higher correlation, which is in line with the estimation results from Chapter 11. However, this attack requires to discard a large number of power traces, which increases the total number of required power traces considerably. Furthermore, the second-order DPA followed by windowing puts less demands on the attacker's knowledge and control over the device. An attack based on mask biasing succeeded with about 305,000 power traces (including discarded ones), while the second-order DPA followed by windowing required only about 50,000 power traces.

In order to introduce a bias in the mask, templates for the mask values have to be built. This requires the availability of a device for profiling which is sufficiently similar to the attacked one. Moreover, the profiling device must offer the possibility to extract some information about the actually occurring mask values in order to allow template building. Depending on the attack scenario, these preconditions might not be always given.

Both methods require a windowing for the time indices $t_1..t_R$, i.e., all points in time where the attacked masked intermediate value can occur due to the randomization countermeasure (cf. Figure 13.3). However, for a practical attack it is not necessary to identify the exact points in time, but it is sufficient to know the distance between those points. In our attack, we have first compressed the power traces (see Section 13.4.2) so that there was only a single power consumption value per clock cycle. When the distance (in clock cycles) between the R points in time is known, all possible combinations of points with this distance can be used in the attack. The selection of possible combinations of R points can be seen as pulling a comb with R teeth over the power trace. The distances between the comb's teeth correspond to the clock cycle distances between the possible occurrences of the masked value $a \oplus m$. This is illustrated in Figure 13.13.

For each position of the comb and each point in time where the mask value m is suspected to appear ("area for mask m" in Figure 13.13), a new correlation value can be calculated. More precisely, second-order DPA preprocessing is applied with the current mask point and each comb tooth. The resulting 16 preprocessed points are then summed up (windowed) to produce a single predicted

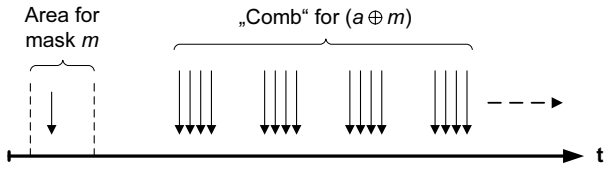

Figure 13.13: Extracting viable points for second-order DPA preprocessing.

power consumption value for the attack.

Normally, an attacker has some general idea of the order of operations which are performed by the attacked device so that it is possible to specify some areas in a power trace where certain values are likely to appear. This limits the number of possible combinations of comb positions and mask positions and makes the attack faster. But even if all possible combinations are used in our case, the total number ranges around $3 \cdot 10^6$, which is quite feasible for an attack[2].

Depending on the scenario, an attacker can obtain information about the distance of the R randomized points (i.e., the distance between the teeth of the comb) through different means. If a device is available for profiling, then the relevant points in time can be determined through a DPA attack with known intermediate values. When the relevant sections of the implementation's source code are available, the distances can be derived with a cycle-accurate simulator. Even if those options are not available, some general knowledge about the protected implementation can be enough to establish a set of *candidate combs* with different distances between the teeth. If this set is not too large, the correct comb can be determined by trying out all combs of the set in an attack.

For our protected AES implementation it would be sufficient to know that there is a randomization of the columns of the State and a randomization of the bytes within each column. The distance between the processing of the columns and between the four S-box lookups of one column are constant. Thus there are only two configuration parameters for the comb, resulting in a manageable set of candidate combs.

Hence, the method of second-order DPA followed by windowing can be regarded as a fairly generic attack which requires only little more knowledge about the implementation than a plain first-order DPA attack.

13.6 Summary and Conclusions

In this chapter we have practically demonstrated the effectiveness of advanced DPA attacks on an AES smart card implementation with state-of-the-art software countermeasures. We have evaluated the three principal attack methods described in Chapter 11, which have so far only been subject to theoretical

[2]There are about 1,800 points per compressed trace, which is hence the upper limit for comb positions and mask positions. The maximal number of combinations is therefore $1,800 \cdot 1,800 = 3,240,000$.

estimation and high-level simulation. Two of these methods work well in defeating the masking and randomization of operations countermeasures of the AES software implementation. One of the methods leads to a potentially higher correlation, but requires the attacker to be able to profile the attacked device in detail. The second attack is not quite as effective but is more general. It can be mounted without profiling and requires only little knowledge about the implementation. Nevertheless, it must not be overlooked that the effort for the attacks is considerably larger than for an unprotected implementation. While 100 power traces are usually enough to break the unprotected implementation, the advanced DPA attacks on the protected implementation require a minimal number of about 50,000 traces for success. Hence, DPA becomes more than two orders of magnitude (i.e., 100 times) harder under use of the described attacks. Although there is no guarantee that there are no better attacks on a specific implementation, our work has delivered empirical evidence that a combination of masking and operation randomization can offer significant protection against advanced DPA attacks.

14 Conclusions

The work described in this thesis has investigated the efficient and secure support of cryptography on embedded processors. This endeavor is of special significance as modern applications of digital computing shift more and more computational tasks to constrained devices like smart cards, wireless sensor nodes, or RFID tags. With the potential gains and risks of such applications, the inclusion of sound security measures becomes imperative. It is of utmost importance that the designers of future embedded applications are provided with efficient and secure implementations of cryptographic algorithms as a solid foundation to build up such security measures. Otherwise, cryptography risks as being perceived as too costly and many application designers might opt for no or inferior ad-hoc security solutions, which will in turn lead to a multitude of security problems in the long run.

We have focused on the promising approach of instruction set extensions, which integrates cryptography support directly into the core of the processor, and makes this support available by means of additional custom instructions. While the field of research has been reasonably covered for public-key cryptography, our efforts have concentrated on secret-key algorithms. We believe that future generations of embedded processors should support a core set of algorithms with dedicated support in hardware, while retain the flexibility to cater for legacy algorithms in software. We envision Elliptic Curve Cryptography and the Advanced Encryption Standard as key components of such a core set. Therefore, our work has centered around the efficient and secure support of the Advanced Encryption Standard and possible synergies with Elliptic Curve Cryptography.

We have given a motivation of our work and a description of important publications in Chapter 1. Chapter 2 contains a brief introduction to cryptography

and provides details on ECC and the AES. The implementation options for cryptographic algorithms on embedded systems are reviewed in Chapter 3 and special focus is put on the approach of instruction set extensions.

We have motivated the approach of instruction set extensions for cryptography with an example for Elliptic Curve Cryptography over GF(2^m) in Chapter 4. We have presented two inexpensive multiply-step instructions which allow to speed up Elliptic Curve point multiplication by a factor between 1.9 and 10 (depending on the implementation's flexibility to deal with different curve parameters). In Chapter 5, we have investigated the synergies of public-key cryptography and secret-key cryptography. We have demonstrated how elaborate instructions for ECC support can also be used to speed up the AES by up to 25%.

We have investigated corresponding instruction set support for different embedded processor platforms. The main focus has been set on 32-bit architectures, where we have used the SPARC V8-compatible LEON2 processor in our practical work as a typical representative. In Chapter 6, table lookups have been identified as a potential bottleneck for implementations on constrained devices as well as a hazard for cache-based side-channel attacks. We have proposed and a simple, yet flexible instruction to remedy the problem, which increases performance by a factor of up to 1.43, while reducing the code size by over 40%. In Chapter 7, a performance-oriented investigation has yielded several sets of possible custom instructions for AES which offer different performance-area tradeoffs. But even the high-performance variant is relatively cheap with a cost of about 3 kGates. In return, AES performance is increased by a factor of up to 10 and code size can be reduced by well over 80%. Furthermore, the proposed extensions have very favorable properties for implementation in superscalar processors. We also proposed a new functional unit for the support of both ECC and AES in Chapter 8. The cost for the added AES support range around 1.3 kGates. In Chapter 9, we have described our design of AES instruction support for a typical modern 8-bit architecture which can achieve a speedup of up to 3.6. At a cost of about 800 gates, code size can be reduced by up to 75%. An outstanding feature of our optimized reference AES implementation are the extremely low RAM requirements of only 4 bytes.

Different design options for the AES S-box in hardware with standard-cells have also been examined in Chapter 10. For the metrics of area, critical path, power consumption, and combinations thereof we have determined the more favorable candidates. We have also evaluated state-of-the-art software countermeasures against side-channel attacks both theoretically and practically. The role of the proposed instruction set extensions as well as new attacks have been investigated in detail in Chapter 11. Hardware countermeasures for processors with cryptography enhancements have been proposed in Chapter 12. A practical evaluation of state-of-the-art software countermeasures has be described in Chapter 13. While side-channel attacks continue to remain a serious threat to device security we believe that our software and hardware countermeasures can be used to increase implementation security significantly.

In any case we need to stress that strong cryptography is only a building block for sound security. Correct application of these primitives in cryptographic protocols and security architectures is of equal importance, as security can only be as good as it's weakest link. Preparing the embedded world for the security challenges, which arise with new applications, is therefore an ongoing process. We hope that our work serves as a useful contribution to this process.

Bibliography

[1] Advanced Micro Devices, Inc. *3DNow! Technology Manual*, 2000.

[2] H. Aigner, H. Bock, M. Hüutter, and J. Wolkerstorfer. A Low-Cost ECC Coprocessor for Smartcards. In M. Joye and J.-J. Quisquater, editors, *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2004.

[3] M.-L. Akkar and C. Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Çetin Kaya Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001.

[4] American National Standards Institute (ANSI). AMERICAN NATIONAL STANDARD X9.62-1998. Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). Working Draft, September 1998.

[5] American National Standards Institute (ANSI). AMERICAN NATIONAL STANDARD X9.63-2001. Public Key Cryptography for the Financial Services Industry, Key Agreement and Key Transport Using Elliptic Curve Cryptography, 2001.

[6] American National Standards Institute (ANSI). AMERICAN NATIONAL STANDARD X9.62-2005. Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA), 2005.

[7] R. Anderson and E. Biham. Tiger: A Fast New Hash Function. In D. Gollmann, editor, *Fast Software Encryption, Third International Workshop Cambridge, UK, February 2123 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 89–97. Springer, 1996.

[8] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. Available online at `http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf`, June 1998. NIST AES Proposal.

[9] K. Aoki and H. Lipmaa. Fast Implementations of AES Candidates. In *Proceedings of the Third AES Candidate Conference, NIST*, pages 106–120, 2000.

[10] ARC International. ARChitect Processor Configurator: The Power of Configurable Processing At Your Fingertips, White Paper. Available online at `http://www.arc.com/configurablecores/architect/`, 2005.

[11] ARM Ltd. SecurCore Family Website. `http://www.arm.com/products/CPUs/families/SecurCoreFamily.html`.

[12] K. Atasu, L. Breveglieri, and M. Macchetti. Efficient AES Implementations for ARM Based Platforms. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC 2004)*, pages 841–845. ACM Press, 2004.

[13] K. Atasu, L. Pozzi, and P. Ienne. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. In *40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003, Proceedings*. ACM Press, 2003.

[14] Atmel Corporation. 8-bit Microcontroller with 16K Bytes In-System Programmable Flash. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc1142.pdf`, February 2003.

[15] Atmel Corporation. 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, August 2007.

[16] Atmel Corporation. 8-bit AVR Instruction Set. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`, May 2008.

[17] Atmel Corporation. AVR XMEGA, 8/16-bit High Performance Low Power Flash Microcontrollers. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc7925.pdf`, 2008.

[18] Atmel Corporation. AVR1317: Using the XMEGA built-in DES accelerator. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc8105.pdf`, April 2008.

[19] Atmel Corporation. AVR1318: Using the XMEGA built-in AES accelerator. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc8106.pdf`, April 2008.

[20] L. S. Au and N. Burgess. A (4:2) Adder for Unified GF(p) and GF(2^m) Galois Field Multipliers. In *Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems and Computers, 2002.*, volume 2, pages 1619–1623. IEEE, November 2002.

[21] G. Bai, Z. Huang, H. Yuan, H. Chen, M. Liu, G. Chen, T. Zhou, and Z. Chen. A High Performance VLSI Chip of the Elliptic Curve Cryptosystems. In *Proceedings of the 7th International Conference on Solid-State and Integrated Circuits Technology, 2004.*, volume 3, pages 2059–2062. IEEE, October 2004.

[22] M. Barr. Embedded Systems Glossary. Available online at `http://www.netrino.com/Publications/Glossary/`.

[23] D. J. Bernstein. Cache-timing attacks on AES. Available online at `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, April 2005.

[24] G. Bertoni, A. Bircan, L. Breveglieri, P. Fragneto, M. Macchetti, and V. Zaccaria. About the Performances of the Advanced Encryption Standard in Embedded Systems with Cache Memory. In *Proceedings of the 36th IEEE International Symposium on Circuits and Systems (ISCAS 2003)*, volume 5, pages 145–148. IEEE, 2003.

[25] G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin. Efficient Software Implementation of AES on 32-Bit Platforms. In B. S. K. Jr., Çetin Kaya Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 159–171. Springer, 2003.

[26] G. Bertoni, L. Breveglieri, F. Roberto, and F. Regazzoni. Speeding Up AES By Extending a 32-Bit Processor Instruction Set. In *Proceedings of the 17th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2006)*, pages 275–282. IEEE Computer Society, September 2006.

[27] G. Bertoni, M. Macchetti, L. Negri, and P. Fragneto. Power-efficient ASIC Synthesis of Cryptographic Sboxes. In D. Garrett, J. Lach, and C. A. Zukowski, editors, *14th ACM Great Lakes Symposium on VLSI 2004, Boston, MA, USA, April 26-28, 2004, Proceedings*, pages 277–281. ACM Press, 2004.

[28] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. AES Power Attack Based on Induced Cache Miss and Countermeasure. In *International Conference on Information Technology: Coding and Computing (ITCC 2005), April 4-6, 2005, Las Vegas, Nevada, USA, Proceedings*, volume 1, pages 586–591. IEEE Computer Society, April 2005. ISBN 0-7695-2315-3.

[29] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne. ISEGEN: Generation of High-Quality Instruction Set Extensions by Iterative Improvement. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 1246–1251. IEEE Computer Society, 2005.

[30] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne, and N. Dutt. Introduction of Local Memory Elements in Instruction Set Extensions. In *41st Design Automation Conference, DAC 2004, San Diego, California, USA, June 7-11, 2004, Proceedings*, pages 729–734. ACM Press, 2004.

[31] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*, volume 265 of *London Mathematical Society Lecture Notes Series*. Cambridge University Press, Cambridge, UK, 1999.

[32] J. Blömer, J. Guajardo, and V. Krummel. Provably Secure Masking of AES. In H. Handschuh and M. A. Hasan, editors, *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*, volume 3357 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2005.

[33] A. Bosselaers. The RIPEMD-160 page. `http://homes.esat.kuleuven.be/~bosselae/ripemd160.html`.

[34] M. K. Brown, D. R. Hankerson, J. C. L. Hernández, and A. J. Menezes. Software Implementation of the NIST Elliptic Curves Over Prime Fields. In D. Naccache, editor, *Topics in Cryptology - CT-RSA 2001, The Cryptographers' Track at the RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2001.

[35] M. Bucci. Dual Mode (Integer, Polynomial) Fast Modular Multipliers, May 1997. Rump session talk given at EUROCRYPT'97, May 13, 1997, Konstanz, Germany.

[36] M. Bucci, M. Guglielmo, R. Luzzi, and A. Trifiletti. A Power Consumption Randomization Countermeasure for DPA-Resistant Cryptographic Processors. In E. Macii, O. G. Koufopavlou, and V. Paliouras, editors, *14th International Workshop on Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation, PATMOS 2004, Santorini, Greece, September 15-17, 2004, Proceedings*, volume 3254 of *Lecture Notes in Computer Science*, pages 481–490. Springer, 2004.

[37] R. Buchty. *Cryptonite — A Programmable Crypto Processor Architecture for High-Bandwidth Applications*. Ph.d. thesis, Technische Universität München, LRR, September 2002. Available online at `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/buchty.pdf`.

[38] J. Burke, J. McDonald, and T. Austin. Architectural Support for Fast Symmetric-Key Cryptography. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000*, pages 178–189, New York, NY, USA, 2000. ACM Press.

[39] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. M. Jr., L. O'Connor, M. Peyravian, D. Safford, and N. Zunic. MARS - a candidate cipher for AES. Available online at `http://www.research.ibm.com/security/mars.pdf`, September 1999. NIST AES Proposal.

[40] C. D. Canniére and C. Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In X. Lai and K. Chen, editors, *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[41] D. Canright. A Very Compact S-Box for AES. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.

[42] Çetin Kaya Koç. Analysis of MMX Technology for Implementing Number-Theoretic Cryptosystems. Technical report, Intel Corporation, February 1997.

[43] Çetin Kaya Koç and T. Acar. Montgomery Multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.

[44] Çetin Kaya Koç, T. Acar, and B. S. K. Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[45] Certicom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0. Available online at `http://www.secg.org/`, September 2000.

[46] A. Chandrakasan, W. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE, 1st edition, 2001.

[47] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[48] C.-C. Chia and S.-S. Wang. Efficient Design of an Embedded Microcontroller for Advanced Encryption Standard. In *Proceedings of the 2005 Workshop on Consumer Electronics and Signal Processing (WCEsp 2005)*, 2005. Available online at `http://www.mee.chu.edu.tw/labweb/WCEsp2005/96.pdf`.

[49] P. Chodowiec and K. Gaj. Very Compact FPGA Implementation of the AES Algorithm. In C. D. Walter, Çetin Kaya Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2003.

[50] N. Clark, H. Zhong, and S. Mahlke. Processor Acceleration Through Automated Instruction Set Customization. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 129–140. IEEE Computer Society, 2003.

[51] N. Clark, H. Zhong, W. Tang, and S. Mahlke. Automatic Design of Application Specific Instruction Set Extensions through Dataflow Graph Exploration. *International Journal of Parallel Programming*, 31(6):429–449, December 2003.

[52] C. Clavier, J.-S. Coron, and N. Dabbous. Differential Power Analysis in the Presence of Hardware Countermeasures. In Çetin Kaya Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2000.

[53] J.-S. Coron, E. Prouff, and M. Rivain. Side Channel Cryptanalysis of a Higher Order Masking Scheme. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 28–44. Springer, September 2007.

[54] CoWare, Inc. Processor Designer Datasheet. Available online at `http://www.coware.com/products/processordesigner_datasheet.php`.

[55] Crossbow Technology, Inc. The Crossbow Technology Website. `http://www.xbow.com/`.

[56] L. Dadda, M. Macchetti, and J. Owen. The Design of a High Speed ASIC Unit for the Hash Function SHA-256 (384, 512). In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, volume 3, pages 70–75. IEEE Computer Society, February 2004. ISBN 0-7695-2085-5.

[57] J. Daemen and V. Rijmen. *The Design of Rijndael*. Information Security and Cryptography. Springer, 2002. ISBN 3-540-42580-2.

[58] B. D. de Dinechin. From Machine Scheduling to VLIW Instruction Scheduling. *ST Journal of Research - Processor Architecture and Compilation for Embedded Systems*, 1(2):1–32, September 2004. Special Report.

[59] J.-F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, May 1998.

[60] T. Dierks. RFC 4346: The Transport Layer Security (TLS) Protocol Version 1.1. Available online at `http://www.ietf.org/rfc/rfc4346.txt`, April 2005.

[61] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strengthened version of RIPEMD. In D. Gollmann, editor, *Fast Software Encryption, Third International Workshop Cambridge, UK, February 2123 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 1996.

[62] W. Drescher, K. Bachmann, and G. Fettweis. VLSI Architecture for Datapath Integration of Arithmetic Over GF(2^m) on Digital Signal Processors. In *IEEE International Conference on Acoustics, Speech, and Signal Processing, 1997 (ICASSP-97)*, volume 1, pages 631–634. IEEE Computer Society, April 1997.

[63] M. Dubash. Moore's Law is dead, says Gordon Moore. Available online at `http://www.techworld.com/opsys/news/index.cfm?NewsID=3477`, April 2005.

[64] H. Eberle, N. Gura, S. C. Shantz, V. Gupta, and L. Rarick. A Public-key Cryptographic Processor for RSA and ECC. In *Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2004)*, pages 98–110. IEEE Computer Society, September 2004.

[65] H. Eberle, A. Wander, N. Gura, S. Chang-Shantz, and V. Gupta. Architectural Extensions for Elliptic Curve Cryptography over GF(2^m) on 8-bit Microprocessors. In *Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2005)*, pages 343–349. IEEE Computer Society, July 2005.

[66] ECRYPT. eSTREAM - The ECRYPT Stream Cipher Project Website. `http://www.ecrypt.eu.org/stream/`.

[67] A. J. Elbirt. Fast and Efficient Implementation of AES via Instruction Set Extensions. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW 2007)*, volume 1, pages 396–403. IEEE Computer Society, May 2007.

[68] G. Elias, L. S. Cheng, A. Miri, and H. Yeap. An Improved FPGA Implementation of a Hyperelliptic Cryptosystem Coprocessor. In *Canadian Conference on Electrical and Computer Engineering, 2004.*, volume 2, pages 773–776. IEEE, May 2004.

[69] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong Authentication for RFID Systems using the AES Algorithm. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004, 6th International Workshop, Cambridge, MA, USA, August 11-13, 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 357–370. Springer, August 2004.

[70] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen. AES Implementation on a Grain of Sand. *IEE Proceedings on Information Security*, 152(1):13–20, October 2005.

[71] A. M. Fiskiran and R. B. Lee. Evaluating instruction set extensions for fast arithmetic on binary finite fields. In J. Cavallaro, T. Noll, S. Rajopadhye, and L. Thiele, editors, *Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2004)*, pages 125–136. IEEE Computer Society, 2004.

[72] A. M. Fiskiran and R. B. Lee. *PAX: A Datapath-Scalable Minimalist Cryptographic Processor For Mobile Environments*, chapter 2. Nova Science, NY, October 2004.

[73] A. M. Fiskiran and R. B. Lee. Performance Scaling of Cryptography Operations in Servers and Mobile Clients. In *Proceedings of the Workshop on Building Block Engine Architectures for Computer Networks (BEACON 2004)*, October 2004. Available online at `http://palms.ee.princeton.edu/PALMSopen/fiskiran04performance_with_citation.pdf`.

[74] A. M. Fiskiran and R. B. Lee. Fast Parallel Table Lookups to Accelerate Symmetric-Key Cryptography. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2005)*, volume 1, pages 526–531. IEEE Computer Society, 2005.

[75] A. M. Fiskiran and R. B. Lee. On-Chip Lookup Tables for Fast Symmetric-Key Encryption. In *Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2005)*, pages 356–363. IEEE, IEEE Press, July 2005.

[76] M. A. Fiskiran and R. B. Lee. Performance Impact of Addressing Modes on Encryption Algorithms. In *Proceedings of the 2001 IEEE International Conference on Computer Design (ICCD 2001)*, pages 542–545. IEEE, September 2001.

[77] C. K. Fong, D. R. Hankerson, J. C. L. Hernández, A. J. Menezes, and M. B. Tucker. Performance Comparisons of Elliptic Curve Systems in Software. Available online at `http://www.cacr.math.uwaterloo.ca/conferences/2001/ecc/hankerson.pdf`, 2001. Presentation at the 5th Workshop on Elliptic Curve Cryptography (ECC 2001).

[78] P. A. L. for Multimedia and S. (PALMS). PAX Project Website. http://palms.ee.princeton.edu/PAX/.

[79] Freescale Semiconductor. *AltiVec Technology Programming Environments Manual*. Freescale Semiconductor Technical Information Center, CH370, 1300 N. Alma School Road, Chandler, Arizona 85224, April 2006.

[80] Gaisler Research. LEON2 Processor Users Manual. XST Edition. Available online at http://www.gaisler.com/doc/leon2-1.0.30-xst.pdf, July 2005. Version 1.0.30.

[81] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In Çetin Kaya Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

[82] B. Gladman. Implementations of AES (Rijndael) in C/C++ and Assembler. Available online at http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm.

[83] B. Gladman. AES Algorithm Efficiency. Available online at http://fp.gladman.plus.com/cryptography_technology/aesr1/, 1999.

[84] R. E. Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, March/April 2000.

[85] J. Goodman and A. Chandrakasan. An Energy Efficient Reconfigurable Public-Key Cryptography Processor Architecture. In Çetin Kaya Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2000.

[86] J. Goodman and A. Chandrakasan. An Energy Efficient Reconfigurable Public-Key Cryptography Processor Architecture. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, November 2001.

[87] J. R. Goodman. *Energy Scalable Reconfigurable Cryptographic Hardware for Portable Applications*. Ph.d. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, August 2000.

[88] L. Goubin and J. Patarin. DES and Differential Power Analysis – The Duplication Method. In Çetin Kaya Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES'99, First International Workshop, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

[89] J. Großschädl. A unified radix-4 partial product generator for integers and binary polynomials. In *Proceedings of the 35th IEEE International Symposium on Circuits and Systems (ISCAS 2002)*, volume III, pages 567–570. IEEE, 2002.

[90] J. Großschädl. Instruction Set Extension for Long Integer Modulo Arithmetic on RISC-Based Smart Cards. In *Proceedings of the 14th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2002)*, pages 13–19. IEEE Computer Society, 2002.

[91] J. Großschädl. Architectural Support for Long Integer Modulo Arithmetic on Risc-Based Smart Cards. *International Journal of High Performance Computing Applications*, 17(2):135–146, Summer 2003.

[92] J. Großschädl, R. M. Avanzi, E. Savaş, and S. Tillich. Energy-Efficient Software Implementation of Long Integer Modular Arithmetic. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2005.

[93] J. Großschädl, P. Ienne, L. Pozzi, S. Tillich, and A. K. Verma. Combining Algorithm Exploration with Instruction Set Design: A Case Study in Elliptic Curve Cryptography. In *Proceedings of the 9th Conference on Design, Automation and Test in Europe (DATE 2006), Munich, Germany, March 6 - 10, 2006*, pages 218–223. European Design and Automation Association, 2006.

[94] J. Großschädl and G.-A. Kamendje. A Single-Cycle $(32 \times 32 + 32 + 64)$-bit Multiply/Accumulate Unit for Digital Signal Processing and Public-Key Cryptography. In *Proceedings of the 2003 10th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2003)*, volume 2, pages 739–741. IEEE, December 2003.

[95] J. Großschädl and G.-A. Kamendje. Architectural Enhancements for Montgomery Multiplication on Embedded RISC Processors. In J. Zhou, M. Yung, and Y. Han, editors, *Applied Cryptography and Network Security, First International Conference, ACNS 2003. Kunming, China, October 16-19, 2003, Proceedings*, volume 2846 of *Lecture Notes in Computer Science*, pages 418–434. Springer, October 2003.

[96] J. Großschädl and G.-A. Kamendje. Instruction Set Extension for Fast Elliptic Curve Cryptography over Binary Finite Fields GF(2^m). In E. Deprettere, S. Bhattacharyya, J. Cavallaro, A. Darte, and L. Thiele, editors, *Proceedings of the 14th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pages 455–468. IEEE Computer Society, June 2003.

[97] J. Großschädl and G.-A. Kamendje. Low-Power Design of a Functional Unit for Arithmetic in Finite Fields GF(p) and GF(2^m). In K. Chae and M. Yung, editors, *Information Security Applications (WISA 2003)*, volume 2908 of *Lecture Notes in Computer Science*, pages 227–243. Springer, 2003.

[98] J. Großschädl and G.-A. Kamendje. Optimized RISC Architecture for Multiple-Precision Modular Arithmetic. In D. Hutte, G. Müller, W. Stephan, and M. Ullmann, editors, *Security in Pervasive Computing - SPC 2003*, volume 2802 of *Lecture Notes in Computer Science*, pages 253–270. Springer, 2003.

[99] J. Großschädl, S. S. Kumar, and C. Paar. Architectural Support for Arithmetic in Optimal Extension Fields. In J. Cavallaro, T. Noll, S. Rajopadhye, and L. Thiele, editors, *Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2004)*, pages 111–124. IEEE Computer Society, 2004.

[100] J. Großschädl, K. C. Posch, and S. Tillich. Architectural Enhancements to Support Digital Signal Processing and Public-Key Cryptography. In B. Rinner and W. Elmenreich, editors, *Proceedings of the 2nd Workshop on Intelligent Solutions in Embedded Systems (WISES 2004)*, pages 129–143, June 2004.

[101] J. Großschädl and E. Savaş. Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m). In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004, 6th International Workshop, Cambridge, MA, USA, August 11-13, 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 133–147. Springer, August 2004.

[102] J. Großschädl, A. Szekely, and S. Tillich. Algorithm Exploration for Long Integer Modular Arithmetic on a SPARC V8 Processor with Cryptography Extensions. In *Proceedings of the 2nd International Conference on Embedded Software and Systems (ICESS 2005)*, pages 187–194. IEEE Computer Society, 2005.

[103] J. Großschädl, A. Szekely, and S. Tillich. The Energy Cost of Cryptographic Key Establishment in Wireless Sensor Networks. In R. H. Deng and P. Samarati, editors, *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS 2007)*, pages 380–382. ACM Press, 2007.

[104] J. Großschädl, S. Tillich, C. Rechberger, M. Hofmann, and M. Medwed. Energy Evaluation of Software Implementations of Block Ciphers under Memory Constraints. In R. Lauwereins and J. Madsen, editors, *2007 Design, Automation and Test in Europe Conference and Exposition (DATE 2007), April 16-20, 2007, Nice, France*, pages 1110–1115. ACM Press, April 2007. ISBN 978-3-9810801-2-4.

[105] J. Großschädl, S. Tillich, and A. Szekely. Performance Evaluation of Instruction Set Extensions for Long Integer Modular Arithmetic on a SPARC V8 Processor. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 680–689. IEEE Computer Society, August 2007.

[106] J. Guajardo, R. Blümel, U. Krieger, and C. Paar. Efficient Implementation of Elliptic Curve Cryptosystems on the TI MSP430x33x Family of Microcontrollers. In K. Kim, editor, *Public Key Cryptography: 4th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2001, Cheju Island, Korea, February 13-15, 2001. Proceedings*, volume 1992 of *Lecture Notes in Computer Science*, pages 365–382. Springer, 2001.

[107] F. K. Gürkaynak, A. Burg, N. Felber, W. Fichtner, D. Gasser, F. Hug, and H. Kaeslin. A 2GB/s balanced AES crypto-chip implementation. In D. Garrett, J. Lach, and C. A. Zukowski, editors, *14th ACM Great Lakes Symposium on VLSI 2004, Boston, MA, USA, April 26-28, 2004, Proceedings*, pages 39–44. ACM Press, 2004.

[108] F. K. Gürkaynak, P. Luethi, N. Bernold, R. Blattmann, V. Goode, M. Marghitola, H. Kaeslin, N. Felber, and W. Fichtner. Hardware Evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt. Record of The State of the Art of Stream Ciphers (SASC) Workshop 2006, February 2006.

[109] C.-S. Ha, J. H. Lee, D. S. Leem, M.-S. Park, and B.-Y. Choi. ASIC Design of IPSec Hardware Accelerator for Network Security. In *Proceedings of 2004 IEEE Asia-Pacific Conference on Advanced System Integrated Circuits 2004.*, pages 168–171. IEEE, August 2004.

[110] G. Hachez, F. Koeune, and J.-J. Quisquater. cAESar results: Implementation of Four AES Candidates on Two Smart Cards. In *Second Advanced Encryption Standard Candidate Conference*, pages 95–108, Hotel Quirinale, Rome, Italy, March 1999.

[111] Y. Han, P.-C. Leong, P.-C. Tan, and J. Zhang. Fast Algorithms for Elliptic Curve Cryptosystems over Binary Finite Field. In K.-Y. Lam, E. Okamoto, and C. Xing, editors, *Advances in Cryptology - ASIACRYPT '99, International Conference on the Theory and Applications of Cryptology and Information Security, Singapore, November 14-18, 1999, Proceedings*, volume 1716 of *Lecture Notes in Computer Science*, pages 75–85. Springer, 1999.

[112] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2004.

[113] D. R. Hankerson, J. C. López Hernandez, and A. J. Menezes. Software Implementation of Elliptic Curve Cryptography Over Binary Fields. In Çetin Kaya Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2000.

[114] T. Hasegawa, J. Nakajima, and M. Matsui. A Practical Implementation of Elliptic Curve Cryptosystems over GF(p) on a 16-Bit Microcomputer. In H. Imai and Y. Zheng, editors, *Public Key Cryptography (PCK 1998)*, volume 1431 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 1998.

[115] C. Herbst, E. Oswald, and S. Mangard. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In J. Zhou, M. Yung, and F. Bao, editors, *Applied Cryptography and Network Security, Second International Conference, ACNS 2006*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 2006.

[116] Y. Hitchcock, E. Dawson, A. Clark, and P. Montague. Implementing an efficient elliptic curve cryptosystem over GF(p) on a smart card. In K. Burrage and R. B. Sidje, editors, *Proceedings of the 10th International Conference on Computational Techniques and Applications (CTAC 2001)*, pages 354–381. Australian Mathematical Society, 2001. Available for download at `http://anziamj.austms.org.au/V44/CTAC2001/Hitc/Hitc.pdf`.

[117] A. Hodjat, D. D. Hwang, B.-C. Lai, K. Tiri, and I. Verbauwhede. A 3.84 Gbits/s AES crypto coprocessor with modes of operation in a 0.18-um CMOS Technology. In J. Lach, G. Qu, and Y. I. Ismail, editors, *15th ACM Great Lakes Symposium on VLSI 2005, Chicago, Illinois, USA, April 17-19, 2005, Proceedings*, pages 60–63. ACM, ACM Press, April 2005. Available online at `http://portal.acm.org/ft_gateway.cfm?id=1057677\&type=pdf\&coll=GUIDE\&dl=GUIDE\&CFID=50585284\&CFTOKEN=38947284`.

[118] A. Hodjat and I. Verbauwhede. High-Throughput Programmable Cryptocoprocessor. *IEEE Micro*, 24(3):34–45, May/June 2004.

[119] A. Hodjat and I. Verbauwhede. Interfacing a High Speed Crypto Accelerator to an Embedded CPU. In *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems, and Computers, 2004*, volume 1, pages 488–492. IEEE, November 2004.

[120] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A Ring-Based Public Key Cryptosystem. In J. Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998.

[121] IAIK. Hash function zoo. http://ehash.iaik.tugraz.at/index.php/HashFunctionZoo.

[122] IBM. The Cell project at IBM Research. http://www.research.ibm.com/cell/.

[123] IEEE. IEEE 802.3 Higher Speed Study Group (HSSG) Website. http://grouper.ieee.org/groups/802/3/hssg/index.html.

[124] IEEE. IEEE P802.3ba 40Gb/s and 100Gb/s Ethernet Task Force. http://www.ieee802.org/3/ba/.

[125] IEEE. IEEE Standard 1363-2000: IEEE Standard Specifications for Public-Key Cryptography. Available online at http://ieeexplore.ieee.org/servlet/opac?punumber=7168, 2000.

[126] IEEE. IEEE Standard 1363a-2004: IEEE Standard Specifications for Public-Key Cryptography, Amendment 1: Additional Techniques. Available online at http://ieeexplore.ieee.org/servlet/opac?punumber=9276, September 2004.

[127] IEEE. IEEE Standard 802.11i-2004: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications. Amendment 6: Medium Access Control (MAC) Security Enhancements. Available online at http://standards.ieee.org/getieee802/, July 2004.

[128] IEEE. IEEE Standard 802.3-2005: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. Available online at http://standards.ieee.org/getieee802/, December 2005.

[129] Intel Corporation. Advanced Encryption Standard (AES) Instructions Set. http://softwarecommunity.intel.com/articles/eng/3788.htm.

[130] Intel Corporation. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency. http://softwarecommunity.intel.com/articles/eng/3775.htm.

[131] Intel Corporation. Itanium 2 Processor Website. http://www.intel.com/products/processor/itanium2/index.htm.

[132] International Organisation for Standardization (ISO). ISO/IEC 7816-3: Information technology - Identification cards - Integrated circuit(s) cards with contacts - Part 3: Electronic signals and transmission protocols. Available online at http://www.iso.org, September 1997.

[133] J. Irwin and D. Page. Using Media Processors for Low-Memory AES Implementation. In E. Deprettere, S. Bhattacharyya, J. Cavallaro, A. Darte, and L. Thiele, editors, *Proceedings of the 14th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pages 144–154. IEEE Computer Society, 2003.

[134] ISEC project. Instruction Set Extensions for Cryptography (ISEC): Project Webpage. http://www.iaik.tugraz.at/isec.

[135] ISO/IEC. Information technology—Security techniques—Hash-functions—Part 3: Dedicated hash-functions. Available from http://www.iso.org/ (with costs), 2004.

[136] K. Itoh, M. Takenaka, and N. Torii. DPA Countermeasure Based on the Masking Method. In K. Kim, editor, *Information Security and Cryptology - ICISC 2001, 4th International Conference Seoul, Korea, December 6-7, 2001, Proceedings*, volume 2288 of *Lecture Notes in Computer Science*, pages 440–456. Springer, 2002.

[137] ITRS. International Technology Roadmap for Semiconductors (ITRS) Website. http://www.itrs.net/.

[138] J. Jaffe. Introduction to Differential Power Analysis, June 2006. Presented at ECRYPT Summerschool on Cryptographic Hardware, Side Channel and Fault Analysis.

[139] J. Jaffe. More Differential Power Analysis: Selected DPA Attacks, June 2006. Presented at ECRYPT Summerschool on Cryptographic Hardware, Side Channel and Fault Analysis.

[140] M. Joye, P. Paillier, and B. Schoenmakers. On Second-Order Differential Power Analysis. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.

[141] C. Karlof, N. Sastry, and D. Wagner. TinySec: A Link Layer Security Architecture for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys 2004)*, pages 162–175. ACM Press, November 2004.

[142] U. Kastens, D. K. Le, A. Slowik, and M. Thies. Feedback Driven Instruction-Set Extension. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Language, Compiler and Tool Support for Embedded Systems (LCTES 2004)*, pages 126–135. ACM Press, 2004.

[143] S. Kent and K. Seo. RFC 4301: Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), Dec 2005.

[144] H. W. Kim and S. Lee. Design and Implementation of a Private and Public Key Crypto Processor and Its Application to a Security System. *IEEE Transactions on Consumer Electronics*, 50(1):214–224, February 2004.

[145] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[146] N. Koblitz. Hyperelliptic cryptosystems. *Journal of Cryptology*, 1(3):139–150, October 1989.

[147] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer, 1996.

[148] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[149] R. Könighofer. A Fast and Cache-Timing Resistant Implementation of the AES. In T. Malkin, editor, *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008, Proceedings*, volume 4964 of *Lecture Notes in Computer Science*, pages 187–202. Springer, April 2008.

[150] M. Koschuch, J. Großschädl, and D. Page. Hardware/Software Co-Design of Public-Key Cryptography for SSL Protocol Execution in Embedded Systems. In *Proceedings of the 2nd Workshop on Embedded Systems Security (WESS 2007)*, 2007. To be published.

[151] M. Koschuch, J. Lechner, A. Weitzer, J. Großschädl, A. Szekely, S. Tillich, and J. Wolkerstorfer. Hardware/Software Co-design of Elliptic Curve Cryptography on an 8051 Microcontroller. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 430–444. Springer, 2006.

[152] O. G. Koufopavlou, G. N. Selimis, N. Sklavos, and P. Kitsos. Cryptography: Circuits and Systems Approach. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology (ISSPIT 2005)*, pages 918–923. IEEE, 2005.

[153] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104: HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), Feb 1997.

[154] X. Lai and J. L. Massey. A Proposal for a New Block Encryption Standard. In I. Damgård, editor, *Workshop on the Theory and Application of Cryptographic Techniques Aarhus, Denmark, May 2124, 1990, Proceedings*, volume 473 of *Lecture Notes in Computer Science*, pages 389–404. Springer, 1990.

[155] R. Lee and J. Huck. 64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture. In *Compcon '96. 'Technologies for the Information Superhighway' Digest of Papers*, pages 152–160. IEEE, February 1996.

[156] R. B. Lee, Z. Shi, and X. Yang. Efficient Permutation Instructions for Fast Software Cryptography. *IEEE Micro*, 21(6):56–69, December 2001.

[157] A. K. Lenstra and E. R. Verheul. The XTR Public Key System. In M. Bellare, editor, *20th Annual International Cryptology Conference Santa Barbara, California, USA, August 2024, 2000 Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2000.

[158] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, 2nd edition, 1994.

[159] W. M. Lim and M. Benaissa. Subword Parallel GF(2^m) ALU: An Implementation for a Cryptographic Processor. In *IEEE Workshop on Signal Processing Systems (SIPS 2003)*, pages 63–68. IEEE, 2003.

[160] H. Lipmaa. AES/Rijndael: speed. Available online at `http://www.adastral.ucl.ac.uk/~helger/research/aes/rijndael.html`.

[161] J. López and R. Dahab. Fast Multiplication on Elliptic Curves over GF(2^{m}) without Precomputation. In Çetin Kaya Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES'99, First International Workshop, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 1999.

[162] J. López and R. Dahab. High-Speed Software Multiplication in $\mathbb{F}_{2^m}$. In B. K. Roy and E. Okamoto, editors, *Progress in Cryptology - INDOCRYPT 2000: First International Conference in Cryptology in India, Calcutta, India, December 2000. Proceedings*, volume 1977 of *Lecture Notes in Computer Science*, pages 203–212. Springer, December 2000.

[163] M. Macchetti and G. Bertoni. Hardware Implementation of the Rijndael SBOX: A Case Study. *ST Journal of System Research*, (0):84–91, July 2003. Available online at `http://www.st.com/stonline/press/magazine/stjournal/vol00/art08.htm`.

[164] S. Mangard. A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. In P. J. Lee and C. H. Lim, editors, *Information Security and Cryptology - ICISC 2002, 5th International Conference Seoul, Korea, November 28-29, 2002, Revised Papers*, volume 2587 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2003.

[165] S. Mangard. Hardware Countermeasures against DPA – A Statistical Analysis of Their Effectiveness. In T. Okamoto, editor, *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004,*

San Francisco, CA, USA, February 23-27, 2004, Proceedings, volume 2964 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 2004.

[166] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007. ISBN 978-0-387-30857-9.

[167] M. Matsui. How Far Can We Go on the x64 Processors? In M. J. B. Robshaw, editor, *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*, pages 341–358. Springer, March 2006.

[168] M. Matsui and S. Fukuda. How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors. In B. K. Roy and W. Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 398–412. Springer, February 2004.

[169] M. Matsui and J. Nakajima. On the Power of Bitslice Implementation on Intel Core2 Processor. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134. Springer, September 2007.

[170] D. May, H. L. Muller, and N. P. Smart. Non-deterministic Processors. In V. Varadharajan and Y. Mu, editors, *Information Security and Privacy, 6th Australasian Conference, ACISP 2001, Sydney, Australia, July 11-13, 2001, Proceedings*, volume 2119 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2001.

[171] J. P. McGregor and R. B. Lee. Architectural Enhancements for Fast Subword Permutations with Repetitions in Cryptographic Applications. In *Proceedings of the 2001 IEEE International Conference on Computer Design (ICCD 2001)*, pages 453–461. IEEE, September 2001.

[172] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box. In A. Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 323–333. Springer, 2005.

[173] V. S. Miller. Use of Elliptic Curves in Cryptography. In H. C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1986.

[174] MIPS Technologies, Inc. SmartMIPS AES Product Website. http://www.mips.com/products/architectures/SmartMIPS_ASE.php.

[175] MIPS Technologies, Inc. The MIPS Technologies Website. http://www.mips.com/.

[176] P. L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44:519–521, 1985.

[177] G. E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965. 1965.

[178] S. Morioka and A. Satoh. An Optimized S-Box Circuit Architecture for Low Power AES Design. In B. S. K. Jr., Çetin Kaya Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 2003.

[179] D. Mukhopadhyay and D. RoyChowdhury. An Efficient End To End Design of Rijndael Cryptosystem in 0.18 μ CMOS. In *18th International Conference on VLSI Design, 2005.*, pages 405–410. IEEE, January 2005.

[180] K. Nadehara, M. Ikekawa, and I. Kuroda. Extended Instructions for the AES Cryptography and their Efficient Implementation. In *IEEE Workshop on Signal Processing Systems (SIPS'04)*, pages 152–157, Austin, Texas, USA, October 2004. IEEE Press.

[181] E. Nahum, S. O'Malley, H. Orman, and R. Schroeppel. Towards High Performance Cryptographic Software. In *Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems, 1995 (HPCS '95)*, pages 69–72. IEEE, August 1995.

[182] National Institute of Standards and Technology (NIST). FIPS-185: Escrowed Encryption Standard (EES). Available online at http://www.itl.nist.gov/fipspubs/fip185.htm, February 1994.

[183] National Institute of Standards and Technology (NIST). FIPS-46-3: Data Encryption Standard, October 1999. Available online at http://www.itl.nist.gov/fipspubs/.

[184] National Institute of Standards and Technology (NIST). FIPS-186-2: Digital Signature Standard (DSS), January 2000. Available online at http://www.itl.nist.gov/fipspubs/.

[185] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at http://www.itl.nist.gov/fipspubs/.

[186] National Institute of Standards and Technology (NIST). FIPS-180-2: Secure Hash Standard, August 2002. Available online at `http://www.itl.nist.gov/fipspubs/`.

[187] R. M. Needham and D. J. Wheeler. Tea extensions. Technical report, Computer Laboratory, University of Cambridge, October 1997.

[188] K. Okeya and K. Sakurai. Efficient Elliptic Curve Cryptosystems from a Scalar Multiplication Algorithm with Recovery of the y-Coordinate on a Montgomery-Form Elliptic Curve. In Çetin Kaya Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2001.

[189] D. Oliva, R. Buchty, and N. Heintze. AES and the Cryptonite Crypto Processor. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 198–209, New York, NY, USA, 2003. ACM Press.

[190] E. Oswald and S. Mangard. Template Attacks on Masking—Resistance is Futile. In M. Abe, editor, *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, volume 4377 of *Lecture Notes in Computer Science*, pages 243–256. Springer, February 2007. ISBN 978-3-540-69327-7.

[191] E. Oswald, S. Mangard, C. Herbst, and S. Tillich. Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In D. Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2006.

[192] E. Oswald, S. Mangard, N. Pramstaller, and V. Rijmen. A Side-Channel Analysis Resistant Description of the AES S-box. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption, 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2005.

[193] E. Oswald and K. Schramm. An Efficient Masking Scheme for AES Software Implementations. In J. Song, T. Kwon, and M. Yung, editors, *Information Security Applications, 6th International Workshop, WISA 2005, Jeju Island, Korea, August 22-24, 2005, Revised Selected Papers*, volume 3786 of *Lecture Notes in Computer Science*, pages 292–305. Springer, August 2005.

[194] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, University of Bristol, Department of Computer Science, June 2002. Available online at `http://www.cs.bris.ac.uk/Publications/Papers/1000625.pdf`.

[195] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[196] Pender Electronic Design GmbH. GR-PCI-XC2V Product Website. `http://www.pender.ch/products_pci_xc2v.shtml`.

[197] T. Popp and S. Mangard. Masked Dual-Rail Pre-Charge Logic: DPA-Resistance without Routing Constraints. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 2005.

[198] L. Pozzi, M. Vuletić, and P. Ienne. Automatic Topology-Based Identification of Instruction-Set Extensions for Embedded Processors. Technical Report CS 01/377, Swiss Federal Institute of Technology Lausanne, Processor Architecture Laboratory, IN-F Ecublens, 1015 Lausanne, Switzerland, December 2001. Available online at `http://lap.epfl.ch/publications/PozziDec01_AutomaticTopologyBasedIdentification_TR.pdf`.

[199] L. Pozzi, M. Vuletić, and P. Ienne. Automatic Topology-Based Identification of Instruction-Set Extensions for Embedded Processors. In *Proceedings of the conference on Design, automation and test in Europe (DATE 2002)*, page 1138. IEEE Computer Society, 2002.

[200] N. Pramstaller and J. Wolkerstorfer. A Universal and Efficient AES Co-Processor for Field Programmable Logic Arrays. In J. Becker, M. Platzner, and S. Vernalde, editors, *Field Programmable Logic and Application, 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004, Proceedings*, volume 3203 of *Lecture Notes in Computer Science*, pages 565–574. Springer, August 2004. ISBN 978-3-540-22989-6.

[201] J.-J. Quisquater and D. Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In I. Attali and T. P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[202] S. Ravi, A. Raghunathan, N. Potlapally, and M. Sankaradass. System design methodologies for a wireless security processing platform. In *39th Design Automation Conference, DAC 2002, New Orleans, Louisiana, USA, June 10-14, 2002, Proceedings*, pages 777–782, New York, NY, USA, 2002. ACM Press.

[203] S. Rinne, T. Eisenbarth, and C. Paar. Performance Analysis of Contemporary Light-Weight Block Ciphers on 8-bit Microcontrollers. Available online at `http://www.crypto.ruhr-uni-bochum.de/imperia/md/content/texte/publications/conferences/lw_speed2007.pdf`, June 2007.

[204] L. Risch. Pushing CMOS beyond the roadmap. In *Proceedings of the 31st European Solid-State Circuits Conference, 2005. ESSCIRC 2005.*, pages 63–68. IEEE, September 2005.

[205] R. L. Rivest. The MD4 Message Digest Algorithm. In A. J. Menezes and S. A. Vanstone, editors, *Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, volume 573 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1990.

[206] R. L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. Available online at `http://www.ietf.org/rfc/rfc1321.txt`, April 1992.

[207] R. L. Rivest, M. Robshaw, R. Sidney, and Y. Yin. The RC6 Block Cipher. Available online at `ftp://ftp.rsasecurity.com/pub/rsalabs/rc6/rc6v11.pdf`, August 1998. NIST AES Proposal.

[208] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978. ISSN 0001-0782.

[209] A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In C. Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2001.

[210] E. Savaş, A. F. Tenca, and Çetin Kaya Koç. A Scalable and Unified Multiplier Architecture for Finite Fields GF(p) and GF(2^m). In Çetin Kaya Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 277–292. Springer, August 2000.

[211] H. Scharwaechter, D. Kammler, A. Wieferink, M. Hohenauer, K. Karuri, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr. ASIP Architecture Exploration for Efficient Ipsec Encryption: A Case Study. In H. Schepers, editor, *Software and Compilers for Embedded Systems, 8th International Workshop, SCOPES 2004, Amsterdam, The Netherlands, September 2-3, 2004, Proceedings*, volume 3199 of *Lecture Notes in Computer Science*, pages 33–46. Springer, September 2004.

[212] H. Scharwaechter, D. Kammler, A. Wieferink, M. Hohenauer, K. Karuri, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr. ASIP Architecture Exploration for Efficient IPSec Encryption: A Case Study. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(2):Article 12, May 2007.

[213] P. Schaumont, K. Sakiyama, A. Hodjat, and I. Verbauwhede. Embedded Software Integration for Coarse-Grain Reconfigurable Systems. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*, pages 137–142. IEEE Computer Society, April 2004.

[214] K. Schgaguler. Assay of the DPA Vulnerability of Micro Electric Circuits Based on FPGA Measurements. Master's thesis, Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria, October 2005.

[215] K. Schgaguler, S. Tillich, and H. Bock. A Dual-FGPA Approach for Evaluation of Countermeasures against Power Analysis. In P. Balog and M. Horauer, editors, *Proceedings of Austrochip 2006, October 11, 2006, Vienna, Austria*, pages 163–168. Fachhochschule Technikum Wien, October 2006. ISBN-13: 978-3-200-00770-3.

[216] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. Performance Comparison of the AES Submissions. In *Proceedings of the Second AES Candidate Conference, NIST*, pages 15–34, March 1999.

[217] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. *The Twofish Encryption Algorithm: A 128-bit Block Cipher*. John Wiley & Sons, 1999.

[218] K. Schramm and C. Paar. Higher Order Masking of the AES. In D. Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006.

[219] R. C. Schroeppel, H. K. Orman, S. W. O'Malley, and O. Spatscheck. Fast Key Exchange with Elliptic Curve Systems. In D. Coppersmith, editor, *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August, 1995, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 1995.

[220] R. Sever, A. N. İsmailoğlu, Y. C. Tedmen, and M. Aşkar. A High Speed ASIC Implementation of the Rijndael Algorithm. In *International Symposium on Circuits and Systems (ISCAS 2004), Vancouver, British Columbia, Canada, May 23-26, 2004, Proceedings*, volume 2, pages 541–544. IEEE, May 2004.

[221] Z. Shi and R. B. Lee. Bit Permutation Instructions for Accelerating Software Cryptography. In *Proceedings of the 11th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2000)*, pages 138–148. IEEE, 2000.

[222] R. Shively. Dual-Core Intel Itanium 2 Processors Deliver Unbeatable Flexibility and Performance to the Enterprise. Available online at `http://www.intel.com/technology/magazine/computing/dual-core-itanium-0806.htm`, 2006.

[223] A. Sinha and A. Chandrakasan. JouleTrack – A Web Based Tool for Software Energy Profiling. In *38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001, Proceedings*, pages 220–225. ACM Press, June 2001.

[224] N. Smyth, M. McLoone, and J. V. McCanny. Reconfigurable Cryptographic RISC Microprocessor. In *Proceedings of the IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test (VLSI-TSA-DAT 2005)*, pages 29–32. IEEE, 2005.

[225] SPARC International, Inc. *The SPARC Architecture Manual, Version 8*. 535 Middlefield Road, Suite 210, Menlo Park, CA 94025, 1992. Revision SAV080SI9308.

[226] F.-X. Standaert, E. Peeters, and J.-J. Quisquater. On the Masking Countermeasure and Higher-Order Power Analysis Attacks. In *International Conference on Information Technology: Coding and Computing (ITCC 2005), April 4-6, 2005, Las Vegas, Nevada, USA, Proceedings*, volume 1, pages 562–567. IEEE Computer Society, April 2005. ISBN 0-7695-2315-3.

[227] STMicroelectronics. Smartcard ICs, 32-bit Platform ICs Website. `http://www.st.com/stonline/products/families/smartcard/sc_sol_ics_st22.htm`.

[228] Sun Microsystems, Inc. UltraSPARC T2 Processor Website. `http://www.sun.com/processors/UltraSPARC-T2/`.

[229] Sun Microsystems, Inc. The VIS Instruction Set, A White Paper. Available online at `http://www.sun.com/processors/vis/download/vis/vis_whitepaper.pdf`, June 2002.

[230] Tensilica, Inc. Xtensa Architecture and Performance, White Paper. Available online at `http://www.tensilica.com/pdf/xtensa_arch_white_paper.pdf`, October 2005.

[231] S. Thakkar and T. Huff. Internet Streaming SIMD Extensions. *IEEE Computer*, 32(12):26–34, December 1999.

[232] S. Tillich, M. Feldhofer, and J. Großschädl. Area, Delay, and Power Characteristics of Standard-Cell Implementations of the AES S-Box. In S. Vassiliadis, S. Wong, and T. Hämäläinen, editors, *6th International Workshop*

on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2006, Samos, Greece, July 17-20, 2006, Proceedings, volume 4017 of *Lecture Notes in Computer Science*, pages 457–466. Springer, July 2006.

[233] S. Tillich, M. Feldhofer, J. Großschädl, and T. Popp. Area, Delay, and Power Characteristics of Standard-Cell Implementations of the AES S-Box. *Journal of Signal Processing Systems*, 50(2):251–261, January 2008.

[234] S. Tillich and J. Großschädl. A Simple Architectural Enhancement for Fast and Flexible Elliptic Curve Cryptography over Binary Finite Fields GF(2^m). In P.-C. Yew and J. Xue, editors, *Advances in Computer Systems Architecture, 9th Asia-Pacific Conference, ACSAC 2004, Beijing, China, September 2004, Proceedings*, volume 3189 of *Lecture Notes in Computer Science*, pages 282–295. Springer, September 2004.

[235] S. Tillich and J. Großschädl. A Survey of Public-Key Cryptography on J2ME-Enabled Mobile Devices. In C. Aykanat, T. Dayar, and I. Körpeoglu, editors, *Computer and Information Sciences - ISCIS 2004, 19th International Symposium, Kemer-Antalya, Turkey, October 2004, Proceedings*, volume 3280 of *Lecture Notes in Computer Science*, pages 935–944. Springer, October 2004.

[236] S. Tillich and J. Großschädl. Accelerating AES Using Instruction Set Extensions for Elliptic Curve Cryptography. In M. Gavrilova, Y. Mun, D. Taniar, O. Gervasi, K. Tan, and V. Kumar, editors, *Computational Science and Its Applications - ICCSA 2005*, volume 3481 of *Lecture Notes in Computer Science*, pages 665–675. Springer, May 2005.

[237] S. Tillich and J. Großschädl. Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2006.

[238] S. Tillich and J. Großschädl. Power-Analysis Resistant AES Implementation with Instruction Set Extensions. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 303–319. Springer, September 2007.

[239] S. Tillich and J. Großschädl. VLSI Implementation of a Functional Unit to Accelerate ECC and AES on 32-bit Processors. In C. Carlet and B. Sunar, editors, *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 2007, Proceedings*, volume 4547 of *Lecture Notes in Computer Science*, pages 40–54. Springer, June 2007.

[240] S. Tillich, J. Großschädl, and A. Szekely. An Instruction Set Extension for Fast and Memory-Efficient AES Implementation. In J. Dittmann, S. Katzenbeisser, and A. Uhl, editors, *Communications and Multimedia Security — 9th IFIP TC-6 TC-11 International Conference, CMS 2005, Salzburg, Austria, September 2005, Proceedings*, volume 3677 of *Lecture Notes in Computer Science*, pages 11–21. Springer, September 2005.

[241] S. Tillich and C. Herbst. Attacking State-of-the-Art Software Countermeasures—A Case Study for AES. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008, 10th International Workshop, Washington DC, USA, August 10-13, 2008, Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 228–243. Springer, August 2008.

[242] S. Tillich and C. Herbst. Boosting AES Performance on a Tiny Processor Core. In T. Malkin, editor, *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008, Proceedings*, volume 4964 of *Lecture Notes in Computer Science*, pages 170–186. Springer, April 2008.

[243] S. Tillich, C. Herbst, and S. Mangard. Protecting AES Software Implementations on 32-bit Processors against Power Analysis. In J. Katz and M. Yung, editors, *Proceedings of the 5th International Conference on Applied Cryptography and Network Security (ACNS 2007)*, volume 4521 of *Lecture Notes in Computer Science*, pages 141–157. Springer, June 2007.

[244] K. Tiri, M. Akmal, and I. Verbauwhede. A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards. In *28th European Solid-State Circuits Conference - ESSCIRC 2002, Florence, Italy, September 24-26, 2002, Proceedings*, pages 403–406. IEEE, September 2002.

[245] K. Tiri, D. D. Hwang, A. Hodjat, B.-C. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. Prototype IC with WDDL and Differential Routing - DPA Resistance Assessment. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 354–365. Springer, 2005.

[246] K. Tiri and I. Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, volume 1, pages 246–251. IEEE Computer Society, February 2004. ISBN 0-7695-2085-5.

[247] E. Trichina, M. Bucci, D. D. Seta, and R. Luzzi. Supplemental Cryptographic Hardware for Smart Cards. *IEEE Micro*, 21(6):26–35, November/December 2001.

[248] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In *International Symposium on Information Theory and Its Applications (ISITA 2002)*, October 2002.

[249] T. Vejda, D. Page, and J. Großschädl. Instruction Set Extensions for Pairing-Based Cryptography. In *Pairing-Based Cryptography (PAIRING 2007)*, volume 4575 of *Lecture Notes in Computer Science*, pages 208–224. Springer, July 2007.

[250] VIA Technologies, Inc. VIA PadLock Security Initiative Website. `http://www.via.com.tw/en/initiatives/padlock/index.jsp`.

[251] J. Waddle and D. Wagner. Towards Efficient Second-Order Power Analysis. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004, 6th International Workshop, Cambridge, MA, USA, August 11-13, 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

[252] C. S. Wallace. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, 1964.

[253] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In V. Shoup, editor, *Advances in Cryptology - CRYPTO 2005, 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[254] C. Weaver, R. Krishna, L. Wu, and T. Austin. Application Specific Architectures: A Recipe for Fast, Flexible and Power Efficient Designs. In *Proceedings of the 2001 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2001)*, pages 181–185. ACM Press, 2001.

[255] A. Weimerskirch, D. Stebila, and S. C. Shantz. Generic GF(2^m) Arithmetic in Software and its Application to ECC. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Information Security and Privacy (ACISP 2003)*, volume 2727 of *Lecture Notes in Computer Science*, pages 79–92. Springer, 2003.

[256] D. J. Wheeler and R. M. Needham. TEA, a Tiny Encryption Algorithm. In B. Preneel, editor, *Second International Workshop on Fast Software Encryption (FSE94), Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 363–366. Springer, 1995.

[257] D. J. Wheeler and R. M. Needham. Correction to xtea. Technical report, Computer Laboratory, University of Cambridge, October 1998.

[258] J. Wolkerstorfer. An ASIC Implementation of the AES MixColumn-operation. In P. Rössler and A. Döderlein, editors, *Proceedings of Austrochip 2001, October 12, 2001, Vienna, Austria*, pages 129–132, October 2001. ISBN 3-9501517-0-2.

[259] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES SBoxes. In B. Preneel, editor, *Topics in Cryptology - CT-RSA 2002, The Cryptographers' Track at the RSA Conference 2002, San Jose, CA, USA, February 18-22, 2002, Proceedings*, volume 2271 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2002.

[260] A. D. Woodbury. Efficient Algorithms for Elliptic Curve Cryptosystems on Embedded Systems. M.Sc. Thesis, Worcester Polytechnic Institute, September 2001.

[261] A. D. Woodbury, D. V. Bailey, and C. Paar. Elliptic Curve Cryptography on Smart Cards without Coprocessors. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Proceedings of the Fourth Working Conference on Smart Card Research and Advanced Applications (CARDIS 2000)*, volume 180, pages 71–92. Kluwer Academic Publishers, 2000.

[262] L. Wu, C. Weaver, and T. Austin. CryptoManiac: A Fast Flexible Architecture for Secure Communication. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 110–119, New York, NY, USA, 2001. ACM Press.

[263] XESS Corporation. XSV Board V1.1 Manual. Available online at `http://www.xess.com/manuals/xsv-manual-v1_1.pdf`, September 2001.

[264] X. Zeng, C. Chen, and Q. Zhang. A Reconfigurable Public-Key Cryptography Coprocessor. In *Proceedings of 2004 IEEE Asia-Pacific Conference on Advanced System Integrated Circuits 2004.*, pages 172–175. IEEE, August 2004.

[265] X. Zhang and K. K. Parhiter. High-Speed VLSI Architectures for the AES Algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(9):957–967, September 2004.