# Energy-Efficient Protocols and Hardware Architectures for Transport Layer Security

by

Utsav Banerjee

B. Tech. (Hons.), Indian Institute of Technology, Kharagpur (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
May 5, 2017

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Anantha P. Chandrakasan
Vannevar Bush Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

# Energy-Efficient Protocols and Hardware Architectures for Transport Layer Security

by

Utsav Banerjee

Submitted to the Department of Electrical Engineering and Computer Science
on May 5, 2017, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

## Abstract

The Internet of Things (IoT) has introduced a vision of an Internet where computing and sensing devices are interconnected. Digitally connected devices are encroaching on every aspect of our lives, including our homes, cars, offices, and even our bodies. Researchers estimate that there will be over 50 billion wireless connected devices by 2020 [1]. On one hand, the IoT enables fundamentally new applications, but on the other, these devices are attractive targets for cyber attackers, thus making IoT security a major concern. Datagram Transport Layer Security (DTLS) is considered to be one of the most suited protocols for securing the IoT. However, computation and communication overheads make it very expensive to implement DTLS on resource-constrained IoT sensor nodes.

In this work, we profile the energy costs of DTLS version 1.3, using experimental models for cryptographic computations and radio-frequency (RF) communications. Based on this analysis, we propose protocol optimizations that can reduce the overall energy consumption of DTLS up to 45%, while still maintaining the same security strength of the standard DTLS. We discuss energy-efficient architectures for implementing the standard cryptographic primitives AES (Advanced Encryption Standard), SHA (Secure Hash Algorithm) and ECC (Elliptic Curve Cryptography) in hardware. Our hardware can provide more than 2,500 times reduction in energy consumption compared to traditional software implementations. These hardware primitives are integrated with dedicated control and memory to design a DTLS co-processor that can accelerate the complete DTLS state machine in hardware, thus minimizing the energy consumption due to DTLS computations. The proposed DTLS core is integrated with a RISC-V micro-processor to accurately profile these functions, as well as design custom protocols using standalone cryptographic instructions.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Vannevar Bush Professor of Electrical Engineering and Computer Science

# Acknowledgments

First, I would like to thank my advisor Prof. Anantha Chandrakasan for giving me the opportunity to be a part of his research group. Working on this project has been an extremely rewarding experience for me, and I thank Anantha for introducing me to this interesting area of research. I look forward to a wonderful journey during my PhD under his guidance and mentorship.

I am grateful to all past and present members of *Anantha Group* for creating such an amazing work environment. Thanks to everybody for helping me with so many different issues throughout the course of this project. It has been my pleasure collaborating with Chiraag on all the security-related projects, and I thank him for being such a great mentor. Thanks to Chiraag, Phil, Mehul, Priyanka and Michael for helping me with the tedious steps involved in taping out a chip. It has been great working with all of you.

I would like to thank Samuel Fuller for the insightful discussions we have had over the last couple of years. Thanks to Andy Wright for collaborating with me on the RISC-V design.

I would like to thank the Irwin Mark Jacobs and Joan Klein Jacobs MIT Presidential Fellowship, the Qualcomm Innovation Fellowship and Analog Devices Inc. for financial support during various phases of this project. I also thank the TSMC University Shuttle Program for helping with chip fabrication.

Finally, I would like to thank my parents for their unconditional love and support. Thank you for everything. Thank you for always being there for me.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The Internet forms the backbone of modern global communication systems. Ever since its creation, the Internet has been growing in leaps and bounds, and has acted as a fuel for innovation in the fields of telecommunications, computer networks, software platforms and hardware architectures. Some of the most important evolutions of the Internet have been the rise of wireless communications, Internet Protocol (IP)-based networking, and cloud computing. It is estimated that 47% of the global population has access to the Internet[1], and around 75% of the global Internet usage is from wireless mobile devices[2]. The IP protocol suite provides a scalable platform for connecting these electronic devices to the Internet, and enabling communication with dedicated servers that constitute the "cloud". IPv4 supports 32-bit addresses, that is, around 4 billion devices, and the world has already witnessed IPv4 address exhaustion over the past few years, another effect of the rapid growth of digital connectivity.

The next major evolution of the Internet is expected to be triggered by the concept of Internet of Things (IoT). The IoT has introduced a vision of an Internet where all computing and sensing devices are interconnected. The IoT is expected to enable new applications and services that bridge our physical and virtual worlds through machine-to-machine (M2M) communications. Fig. 1-1 shows a typical IoT sensor node. An important component of such a device is a sensor that collects useful information

[1]https://en.wikipedia.org/wiki/Internet

[2]https://fortune.com/2016/10/28/internet-use-mobile/

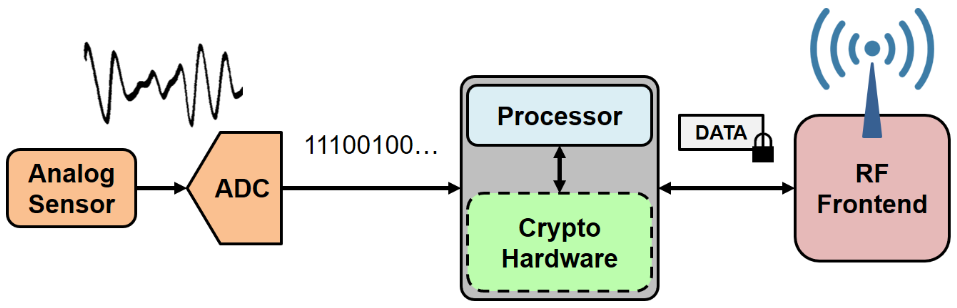


Figure 1-1: System diagram of a typical IoT wireless sensor node.

about its immediate environment. An analog-to-digital converter (ADC) is used to convert this analog information into digital bits. A low-power micro-processor is used to extract meaningful data out of these bits, and a radio-frequency (RF) front-end transmits the data into the wireless network. Since the data packets transmitted by these sensor nodes are likely to contain private information, it is important to encrypt them and ensure a secure channel to the cloud. For this purpose, sensor nodes may also contain additional hardware, also known as *co-processor* or *accelerator*, to efficiently execute cryptographic computations such as data encryption and authentication. The goal of this research is to design energy-efficient security protocols and dedicated cryptographic hardware for such wireless sensor nodes that constitute the IoT.

## 1.1 Motivation

Researchers estimate that there will be over 50 billion wireless connected devices by 2020 [1]. On one hand, the IoT enables fundamentally new applications, but on the other, these devices are attractive targets for cyber attackers, thus making IoT security a major concern. IOActive's 2016 security survey[3] reveals that less than 10% IoT products have adequate security. Symantec reports that there were 528 mobile security vulnerabilities and around 1.1 million web attacks blocked per day in 2015 [2]. There have also been an increasing number of attacks, both proof-of-concept and real-world, on IoT systems such as cars, smart home solutions, implantable medical devices and other consumer electronics [2]. Researchers have recently demonstrated

[3]https://www.infosecurity-magazine.com/news/survey-less-than-10-of-iot-devices/

an attack where malware-equipped drones were used to remotely hack smart lights in an office building and cause irreversible damage such as blackouts, data breach and even epileptic seizures [3]. Proof-of-concept attacks have also been demonstrated on implantable medical devices, such as pacemakers, that can potentially have lethal effects [4].

The number of devices in an IoT network can vary from tens (e.g., smart home and health-care solutions) to hundreds (e.g., agricultural and industrial automation) depending on the application. Most of these devices are required to operate unattended for extended periods of time, and are either battery-powered or rely on energy harvesting, thus making them severely energy-constrained. Cyber security experts unanimously agree that traditional IT security approaches do not fit directly in the IoT scenario, one of the primary reasons being resource constraints. Therefore, security architectures for these devices should consume minimal resources, while still providing strong cryptographic guarantees.

## 1.2 Transport Layer Security

Since the IoT will be integrated with the conventional Internet, most IoT devices are expected to use Internet Protocol (IP) addresses as unique identifiers, and IPv6 will be used to fulfill the large address space requirements. Therefore, IP-based security protocols become the first choice for securing these networks. Transport Layer Security (TLS) is a cryptographic protocol widely used by the Internet community to provide secure and reliable data communications for applications such as e-mail and financial transactions, and this forms the basis of HTTPS (secure HTTP). TLS has been standardized by the IETF [5] to secure connection-based Internet services such as Transmission Control Protocol (TCP); but TCP is not suitable for low-power wireless networks, primarily because of protocol overheads. The User Datagram Protocol (UDP) has emerged as the transport layer protocol-of-choice for the IoT. UDP-based services are connection-less and light-weight, hence they require low bandwidth and minimal memory usage on embedded devices. The Datagram Transport Layer Security (DTLS) protocol is based on TLS, and is intended to secure UDP-based communications. Connection-less services are unreliable, and present unique challenges such as packet re-ordering, packet loss and packet fragmentation. DTLS is designed to not only handle these problems seamlessly, but also counter replay and denial-of-service (DoS) attacks. DTLS has also been standardized by the IETF [6], and is considered as one of the most suited protocols for securing the IoT [7].

The Network Working Group of IETF is in the process of standardizing the next version of TLS – TLS 1.3. Both TLS 1.3 [8] and DTLS 1.3 [9] currently exist in the form of working drafts. (D)TLS consists of two layers – *record protocol* and *handshake protocol*. The record protocol encrypts application layer payloads, fragments and encapsulates them into structured packets, called *records*, and provides message authentication. The handshake protocol allows the communicating parties (*client* and *server*) to negotiate security settings, perform mutual authentication and establish a secure channel for the exchange of encrypted records. (D)TLS 1.3 proposes a major overhaul of the handshake protocol, removes support for weaker cryptographic primitives and adds stronger security measures than its predecessor.

## 1.3 Cryptographic Primitives

The primary objectives of security protocols are confidentiality, integrity, authenticity and availability, and they use mathematical tools, called *cryptographic primitives*, to achieve these goals. Cryptographic primitives can be very broadly classified into two types – *symmetric* and *asymmetric*. Symmetric algorithms use the same secret key to encrypt and decrypt messages, while for asymmetric algorithms, the encryption and decryption keys are different. Asymmetric algorithms, also known as public-key cryptography, is based on intractable mathematical problems, and is much more computationally intensive than symmetric-key algorithms. This section provides a quick overview of some of the cryptographic primitives standardized by the U.S. National Institute of Standards and Technology (NIST) for commercial and government use. For detailed introduction to cryptography and related mathematical background, the reader is encouraged to refer to [10] and [11].

### 1.3.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data, established by NIST in 2001 [12], and is a subset of the "Rijndael" cipher designed by Vincent Rijmen and Joan Daemen. AES is a symmetric block cipher, which uses a substitution-permutation network to process data in blocks of 128 bits, and the substitution function, called *S-Box*, is the only non-linear component of the algorithm. AES supports key sizes of 128, 192 and 256 bits, and consists of 10, 12, or 14 iteration rounds, respectively. AES can be used in the GCM (Galois/Counter Mode) configuration [13] to perform authenticated encryption, that is, it not only encrypts the data (confidentiality) but also generates a MAC (Message Authentication Code) which can be used to verify that the encrypted message has not been corrupted (integrity and authenticity).

While practical attacks have been demonstrated on reduced round versions of AES, the fastest attack on full 10-round AES-128, known as the *biclique attack*, has a time complexity of $O(2^{126})$ and a space complexity of $O(2^{56})$ [14]. Therefore, there is no known practical attack that would allow someone without knowledge of the secret key to read data encrypted by a correct implementation of AES. Side channel attacks have been demonstrated on software and hardware implementations of AES, but they make use of additional knowledge about the secret key and AES round computations that get leaked through side channels like execution timing, power consumption, etc. Several software-based side-channel attacks exploit incorrect implementations of the algorithm, and may also require special user privileges on the device.

### 1.3.2 Secure Hash Algorithm

The Secure Hash Algorithm (SHA) is a family of cryptographic hash functions, that is, they can be used to map data of arbitrary sizes to a digest of fixed size. NIST published SHA-1, the first version of the SHA family, in 1995. The SHA-2 family was published in 2001 [15], and it consists of six hash functions with digests that are 224, 256, 384 or 512 bits. SHA-3 was standardized in 2015 [16], based on the "Keccak" function designed by Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche, and supports arbitrary digest sizes. These cryptographic hash functions can be used in security protocols to generate Message Digests (MD) and HMACs (Hash-Based Message Authentication Code) [20].

For a hash function with $L$-bit message digest, finding a message that corresponds to a given message digest can always be done using a brute force search in $2^L$ evaluations. This is called a *pre-image attack* and may or may not be practical depending on $L$ and the particular computing environment. Finding two different messages that produce the same message digest, known as a *collision*, requires on average only $2^{L/2}$ evaluations using a "birthday attack". Researchers have recently been able to successfully generate collisions with full SHA-1, thus rendering it insecure [17]. Currently, the best known attacks break pre-image resistance for 52 rounds of SHA-256 or 57 rounds of SHA-512 [18], and collision resistance for 46 rounds of SHA-256 [19]. While the collision attack is practical, with $O(2^{46})$ time complexity, the pre-image attack has a time complexity of $O(2^{255.5})$, which makes it impossible to implement in practice. No attacks exist on full round SHA-2, and also on SHA-3.

### 1.3.3 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. The use of elliptic curves in cryptography was suggested independently by Neal Koblitz [21] and Victor S. Miller [22] in 1985. For this research, we consider two types of elliptic curves $E$ over finite fields $\mathbb{F}_p$ of prime characteristic $p$ – short Weierstrass curves and Montgomery curves:

- A short Weierstrass curve consists of the set of points $E(\mathbb{F}_p) = \{(x, y) \mid y^2 = x^3 + ax + b \pmod{p}\} \cup \infty$
- A Montgomery curve consists of the set of points $E(\mathbb{F}_p) = \{(x, y) \mid by^2 = x^3 + ax^2 + x \pmod{p}\} \cup \infty$

where $\infty$ is the distinguished point at infinity. A Montgomery curve can also be written in the Weierstrass form using the proper change of variables.

The fundamental operation used in ECC is *point addition* ($R = P + Q$), and its special case is *point doubling* ($R = P + P$). The addition and doubling operations are both governed by the "chord-and-tangent rule" shown in Fig. 1-2.

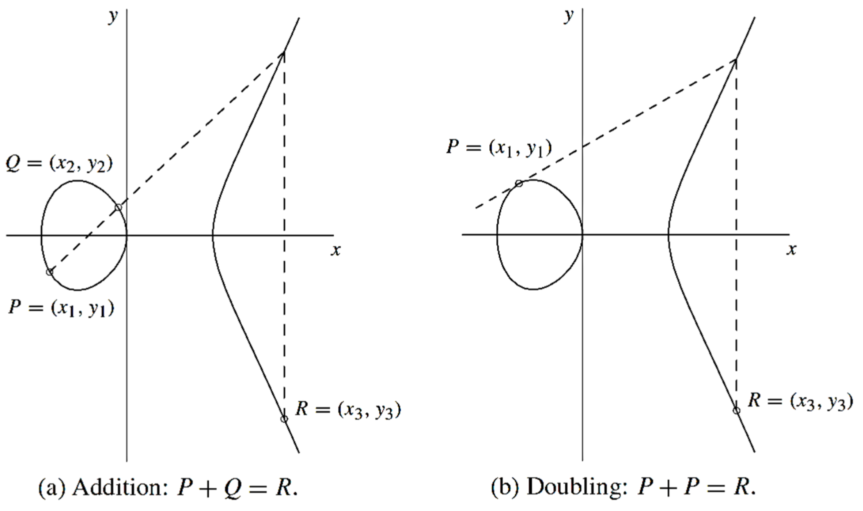


Figure 1-2: Geometric representation of (a) point addition and (b) point doubling on elliptic curves [23].

Together with the addition (and doubling) operation, the set of points $E(\mathbb{F}_p)$ forms an abelian group. The point $\infty$ serves as the identity element of this group, that is, $P + \infty = \infty + P = P$ for all $P \in E(\mathbb{F}_p)$. The order of this group (number of points in $E(\mathbb{F}_p)$) is denoted by $\#E(\mathbb{F}_p) = n$, and $nP = \infty$ for all $P \in E(\mathbb{F}_p)$. Repeated additions of a point $P$ with itself is called "elliptic curve scalar multiplication" (ECSM). For any scalar $k$, the scalar multiple $kP$ is computed as

$$\underbrace{kP = P + P + \cdots + P}_{(k\text{-}1)\text{ point additions}}$$

This computation forms the basis of the "Elliptic Curve Discrete Logarithm Problem" (ECDLP) – determine scalar $k$ given the elliptic curve $E(\mathbb{F}_p)$ of order $n$, and the points $P, Q \in E(\mathbb{F}_p)$ such that $Q = kP$. For a $t$-bit prime $p$, the fastest known algorithms that can solve ECDLP have time complexity $O(2^{t/2})$ [23]. For sufficiently large primes, it is infeasible for a computationally bounded adversary to solve ECDLP, and this guarantees the security of ECC. NIST has published a list of recommended elliptic curves over prime fields, with primes of length 192, 224, 256, 384 and 521 bits [27]. These curves are all short Weierstrass curves, and are recommended for use in all Internet security applications including TLS. Daniel J. Bernstein's Curve25519 [24] is a Montgomery curve, and has been proposed to be included in TLS 1.3.

### Elliptic Curve Diffie-Hellman Key Exchange (ECDH)

The Diffie-Hellman (DH) key exchange is a method of securely establishing a shared secret between two parties communicating over an insecure channel, originally conceptualized by Ralph Merkle and named after Whitfield Diffie and Martin Hellman [25], [26]. DH provides the basis for a variety of authenticated protocols, and is used to provide forward secrecy in TLS and DTLS.

Although the original DH protocol used the multiplicative group of integers modulo a prime $p$, we are going to discuss ECDH (Fig. 1-3), its elliptic curve analogue. The key exchange protocol starts with two parties, Alice and Bob, who wish to compute a shared secret. The adversary, Eve, is monitoring all messages exchanged by them, that is the communications channel is insecure. Both Alice and Bob use the same elliptic curve $E(\mathbb{F}_p)$ with order $n$, and a base point $G$ of prime order on the curve ($nG = \infty$). Alice and Bob generate their private keys, which are the scalars $a, b \in [1, n-1]$ respectively. They compute and exchange their public keys $Q_A = aG$ and $Q_B = bG$. Eve intercepts $Q_A$ and $Q_B$, but cannot determine $a$ and $b$ because of the hardness of ECDLP. Finally,

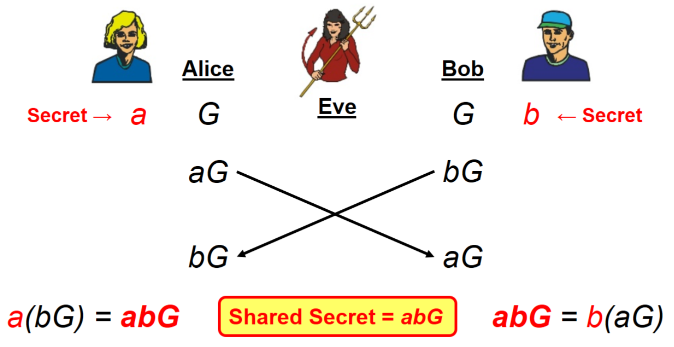


Figure 1-3: Elliptic Curve Diffie-Hellman (ECDH) Key Exchange.

Alice computes $aQ_B = a(bG) = abG$, and Bob computes $bQ_A = b(aG) = abG$, that is, they now share a secret, the point $abG \in E(\mathbb{F}_p)$, which can be used for further cryptographic computations.

### Elliptic Curve Digital Signature Algorithm (ECDSA)

The Digital Signature Algorithm (DSA) is a NIST standard algorithm used to demonstrate the authenticity of digital messages and documents. ECDSA is a variant of DSA, also standardized by NIST [27], which uses elliptic curves for its public-key computations. Both DSA and ECDSA consist of two algorithms - *signature generation* and *signature verification*.

Fig. 1-4 provides a simple description of these two steps, where Alice wants to send a signed message $m$ to Bob, and Bob wants to verify the signature, using ECDSA with a known elliptic curve $E(\mathbb{F}_p)$ with order $n$ and base point $G$. Alice has her private key $a$, and the corresponding public key is $Q_A = aG$. She computes a message digest $H(m)$ using a cryptographic hash function $H$, and generates a signature using the "ECSDA-Sign" algorithm, with $H(m)$ and $a$ as inputs. She then sends the message $m$, the signature and her public key $Q_A$ to Bob. To determine the authenticity of the message, Bob again computes $H(m)$, and verifies the signature using the "ECDSA-Verify" algorithm, with $H(m)$, $Q_A$ and the signature as inputs.

In order for Bob to verify the signature correctly, it is important to have the correct public key $Q_A$. Since Alice sent $Q_A$ over an insecure channel, it is possible

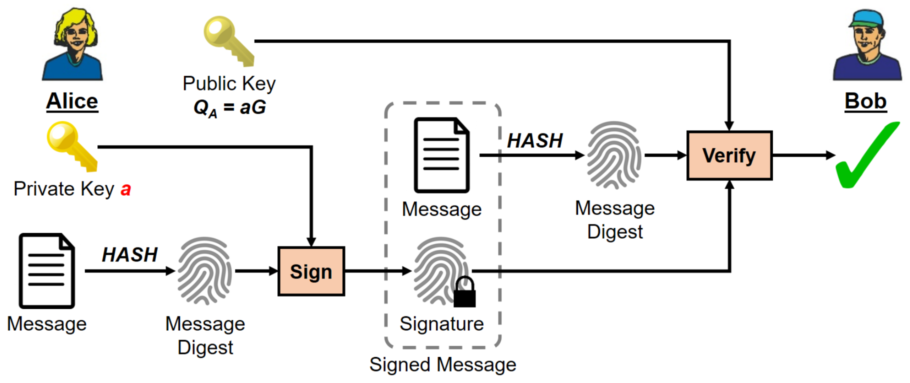


Figure 1-4: Elliptic Curve Digital Signature Algorithm (ECDSA).

that an adversary Eve may modify it. To prevent such attacks, *digital certificates* are used. A digital certificate is a document containing the public key of an entity (person or organization), along with information that defines their identity (name, address, e-mail, etc), which is signed by a *certificate authority* (CA), whose public key is trusted to be secure. Certificates are an integral part of the (D)TLS authentication handshake.

Table 1.1: NIST standard cryptographic primitives and their security strengths.

| **Security Strength (Bits)** | **AES** | **SHA** | **ECC** |
|---|---|---|---|
| 128 | AES-128 | SHA-256 | P-256 |
| 192 | AES-192 | SHA-384 | P-384 |
| 256 | AES-256 | SHA-512 | P-521 |

Table 1.1 shows the security strengths of various NIST standard cryptographic primitives. (D)TLS supports a large number of *cipher suites*, combinations of cryptographic algorithms, that can be used for the handshake and record protocols, and the recommended minimum security level is 128 bits. In this work, we are going to focus on the TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 cipher suite, which uses the previously described NIST standard primitives. Implementation details of these primitives will be discussed in Chapter 3.

## 1.4 Previous Work

This work spans multiple areas of research, including low-overhead TLS protocols, energy-efficient cryptographic hardware and integration of multiple primitives to support protocol acceleration. This section provides a brief overview of some of the existing literature in these fields.

### 1.4.1 TLS Protocol Optimizations

With the advent of low-power wireless links such as IEEE 802.15.4 [28] and BLE (Bluetooth Low Energy) [29], researchers have been trying to optimize the communication overheads of IP-based protocols, so that these new devices can be easily integrated with existing network infrastructure. [30] proposed IPv6/UDP header compression for IEEE 802.15.4 networks, and [31] proposed using the same for IPv6/UDP over BLE. Similar compression schemes were proposed for DTLS in[32], where DTLS record and handshake headers can support variable length compressed formats, and some of the handshake messages like ClientHello and ServerHello can also be compressed. [33] has presented a new protocol architecture where DTLS records are transmitted as CoAP (Constrained Application Protocol) resources, and CoAP acknowledgments are used to prevent message re-ordering and re-transmissions.

### 1.4.2 Cryptographic Hardware Architectures

Researchers have been studying cryptographic algorithms and their implementations for a long time, and research in energy-efficient cryptographic hardware architectures has received increased attention in the recent past. While FPGA (Field Programmable Gate Array) implementations are great for understanding the area and speed limitations of a design, it is important to have ASIC (Application-Specific Integrated Circuit) implementations in order to accurately estimate power and energy consumption. Next, we take a quick look at some of the existing research in ASIC implementations of AES, SHA and prime-field ECC.

Multiple AES-128 architectures are proposed in [34], with different data-path sizes and different levels of parallelism, along with composite field-based compact S-Box designs. Highly serialized architectures for AES-128 are low-area and low-power ([35], [37], [38]), but have high energy consumption. A low-energy 128-bit data-path AES-128 design, with a 4-stage 2-round pipeline, is published in [36]. In [39], several implementations for AES-GCM are presented. [40] also proposes some highly parallel high-throughput AES-GCM designs.

Several serial implementations of SHA-256 are available in literature ([41], [42], [43]), with similar area but different throughput figures. In [44], a low-energy highly parallel reconfigurable SHA accelerator is presented, that supports SHA-1 and SHA-2 for all digest sizes.

ECC hardware architectures have been studied extensively for a large variety of elliptic curves. ECC processors with dual-field arithmetic are presented in [45] and [46]. Low-power low-area prime-field ECDSA implementations, which target RFID (Radio Frequency Identification) applications, are discussed in [47], [48], [49], [50], [52] and [56]. A low-energy full data-path prime-field ECC implementation is published in [51], and a low-area design is proposed in [53]. A low-resource Curve25519-based ECDH implementation is presented in [57].

There have been attempts to integrate multiple cryptographic primitives in hardware, e.g., all the ECDSA designs also contain a hash function by default. [54] presents a high-performance processor that can accelerate AES, SHA-1 and modular arithmetic for TLS handshakes. The processor contains several execution units to run the cryptographic computations, and a configurable micro-code engine controls the scheduling of these algorithms. [55] proposes an energy-efficient reconfigurable public key cryptography processor that supports modular arithmetic and binary-field ECC computations. The cryptographic processor in [56] integrates AES, SHA-1 and ECDSA in a low-area implementation suitable for embedded systems. [57] presents a hardware implementation of the `crypto_box` function of the Networking and Cryptography library (NaCl), which integrates Curve25519-based ECDHE with Salsa20-Poly1305-based authenticated encryption for resource-constrained applications.

These designs are all geared toward high-performance or low-area (and low-power) applications, and do not focus on energy-efficiency. Also, most of the prime-field ECC processors support a single curve, where the modular arithmetic unit has been designed to exploit the special properties of the chosen prime. There is no previous work on dedicated energy-efficient DTLS co-processor that can accelerate the entire protocol in hardware – cryptographic computations and the TLS state machine.

## 1.5 Thesis Overview and Contributions

The objectives of this research are to optimize the DTLS protocol for increased energy-efficiency, without compromising security, and design an energy-efficient co-processor that can accelerate the complete DTLS protocol in hardware.

Chapter 2 provides a comprehensive analysis of the energy costs of DTLS in a typical IoT application. BLE is used as the RF link for this scenario, primarily because of its low energy requirements. Based on this analysis, an energy-optimized variant of DTLS is proposed, which can reduce communication overheads as well as computation costs, independent of the application layer and physical/link layer protocols being used/

Chapter 3 proposes energy-efficient architectures for AES-128, SHA-256 and prime-field ECC (Weierstrass and Montgomery curves up to 256 bits). Rationale behind design decisions are discussed in detail, along with post-synthesis results and comparison with previous work.

Chapter 4 presents a DTLS co-processor, which combines the designs described in Chapter 3 with dedicated control and memory to accelerate the DTLS state machine in hardware. The co-processor is integrated with a standard RISC-V microprocessor [72] to profile the DTLS protocol as well as several other cryptographic functions. This integrated system also provides a highly flexible platform for designing and implementing security protocols using a combination of hardware and software approaches. ASIC implementation and simulation results are presented.

Chapter 5 summarizes the contributions of this thesis, along with key observations and inferences, and future research directions.

# Chapter 2

# DTLS 1.3 – Energy Analysis and Protocol Optimizations

The DTLS protocol has been tried and tested for over a decade, and provides strong security guarantees. However, computation and communication overheads make DTLS very expensive for energy-constrained IoT devices. In this work, we present a comprehensive study of the energy costs of DTLS version 1.3. For this case study, we assume Bluetooth Low Energy (BLE) as the physical and link layer protocol, and use energy models for cryptographic computations and RF communications to analyze how the energy consumption varies over different IoT use cases. Based on this analysis, we propose an optimized low-energy variant of DTLS, which can be used to achieve improved energy-efficiency while providing the same security guarantees of DTLS.

## 2.1 Software Profiling of Cryptographic Algorithms

DTLS owes its security to cryptographic algorithms of varying complexity, such as symmetric key encryption, hashing, public key authentication and key exchange. However, they also add to the computation cost of DTLS, which is a serious concern for resource-constrained embedded devices that constitute the IoT. To accurately profile the energy requirements of these cryptographic primitives, we implemented them in software on the NXP FRDM-KL25Z evaluation board, which contains an ultra-low-

Table 2.1: Profiling of Symmetric Cryptography algorithms on ARM Cortex-M0+.

| **Cryptographic Computation** | **Energy** |
|---|---|
| AES-128-GCM Auth-Encrypt | 0.121 $\mu$J/B |
| AES-128-GCM Auth-Decrypt | 0.124 $\mu$J/B |
| AES-256-GCM Auth-Encrypt | 0.141 $\mu$J/B |
| AES-256-GCM Auth-Decrypt | 0.145 $\mu$J/B |
| SHA-256 Message Digest | 0.043 $\mu$J/B |
| SHA-256 HMAC (64-Byte Key) | 0.052 $\mu$J/B |
| SHA-512 Message Digest | 0.089 $\mu$J/B |
| SHA-512 HMAC (128-Byte Key) | 0.122 $\mu$J/B |

Table 2.2: Profiling of Elliptic Curve Cryptography algorithms on ARM Cortex-M0+.

| **Cryptographic Computation** | **Energy** |
|---|---|
| P-256 ECDHE | 33.06 mJ/Op |
| P-256 ECDSA-Sign | 12.36 mJ/Op |
| P-256 ECDSA-Verify | 34.02 mJ/Op |
| P-384 ECDHE | 69.26 mJ/Op |
| P-384 ECDSA-Sign | 25.43 mJ/Op |
| P-384 ECDSA-Verify | 70.42 mJ/Op |
| P-521 ECDHE | 143.92 mJ/Op |
| P-521 ECDSA-Sign | 52.08 mJ/Op |
| P-521 ECDSA-Verify | 145.49 mJ/Op |

power 90nm ARM Cortex-M0+ micro-processor running at 48 MHz [58]. The software was written as bare-metal C code using the open-source cryptographic libraries from ARM mbedTLS [59]. Average power for the ARM processor was measured, including the memory power, and the ARM *SYS_ TICK* timer was used to accurately measure execution times of the algorithms. Total energy consumption of the processor core and memory is reported in Tables 2.1 and 2.2.

Table 2.1 shows experimental results for symmetric cryptography algorithms – AES and SHA-2. We implemented AES-GCM with 12-byte IV and 13-byte AAD. For SHA-2, both MD and HMAC were implemented. Energy consumption is reported per byte of input message. Table 2.2 shows experimental results for ECDHE and ECDSA. Energy consumption is reported per operation for ECDHE, ECDSA-Sign and ECDSA-Verify. The NIST standard prime curves P-256, P-384 and P-521 were used for this analysis. Windowing methods, with window size $W = 3$, were used for faster elliptic curve operations, along with efficient modular arithmetic owing to the special structure of the NIST primes. Larger window sizes could not be used due to memory constraints of the processor. Clearly, the ECC algorithms are significantly more expensive compared to AES and SHA, and will contribute to majority of DTLS computation costs, as will be discussed later.

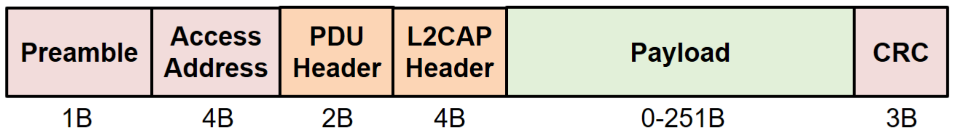


Figure 2-1: Bluetooth Low Energy 4.2 data packet structure (all sizes in bytes).

## 2.2 Energy Model for BLE

BLE is the low-energy version of Bluetooth [29] that operates at 1 Mbps data rate, and employs adaptive frequency hopping spread-spectrum to communicate over the unlicensed 2.4 GHz ISM band. Energy consumption of BLE is much lower than other RF protocols like IEEE 802.15.4 [60], thus making it the popular choice for IoT applications. Fig. 2-1 shows a standard BLE 4.2 data packet. The 4 byte access address is the physical address of the slave device. The PDU (Physical Data Unit) header contains control flags and the payload size (in bytes), while the L2CAP (Logical Link Control and Adaptation Protocol) header contains information about packet fragmentation. The payload can be up to 251 bytes long, and its integrity is protected by a 3-byte CRC (Cyclic Redundancy Check).

BLE networks, called *piconets*, are comprised of multiple *slave* devices connected to a *master* device which coordinates all the communications, that is, a piconet is inherently in star topology. Slave devices are in sleep for most of the time, except for periodic connection events when the slave wakes up to communicate with the master. Connection events always start with a packet being sent by the master, and

Table 2.3: Energy consumption of TI CC2540 BLE transceiver during different phases of a connection event [60], [61].

| **Phase** | **Energy** |
|---|---|
| Wake-Up and Pre-Processing | $E_{WUP} = 15\ \mu$J |
| Receive (RX) | $E_{RX} = 0.528\ \mu$J/B |
| Inter-Frame Space (IFS) | $E_{IFS} = 6.75\ \mu$J |
| Transmit (TX) | $E_{TX} = 0.672\ \mu$J/B |
| Post-Processing and Sleep | $E_{SLP} = 33.6\ \mu$J |

the slave has to wait for 150 $\mu$s, called Inter-Frame Space (IFS), before transmitting data. Table 2.3 shows the energy consumed by the TI CC2540 BLE transceiver [61], [60] during different phases of a connection event, when it has to transmit data to the master. The energy spent by a BLE 4.2 slave device during transmission ($E_T$) and reception ($E_R$) of data can be modeled using the following equations [60]:

$$E_T = E_{WUP} + nl_{HDR}E_{RX} + (2n-1)E_{IFS} + (nl_{HDR} + l_P)E_{TX} + E_{SLP}$$

$$E_R = E_{WUP} + (nl_{HDR} + l_P)E_{RX} + (2n-1)E_{IFS} + nl_{HDR}E_{TX} + E_{SLP}$$

where $l_{HDR}$ (= 14 bytes) is the total size of BLE header and trailer structures, $l_P$ is the total payload being transmitted / received, and $n$ is the number of fragments the payload gets divided into.

## 2.3 DTLS over BLE

Using these models, we can analyze the energy consumption of a duty-cycled BLE sensor node communicating with a cloud server using a DTLS-protected secure channel. As case study, we consider a DTLS 1.3 connection with the following parameters:

- The TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 cipher suite is negotiated, and the elliptic curve used for ECDHE and ECDSA is P-256.
- Only end-point certificates, signed using P-384 ECDSA by a trusted certification authority (CA), are exchanged during handshake.

- The CA public key is known to both parties.

Fig. 2-2 shows the message flow for a full DTLS 1.3 handshake with digital certificate-based mutual authentication and Diffie-Hellman key exchange (blue arrows represent handshake messages and green arrows represent application data; dashed arrows indicate that the messages are encrypted). The client begins the DTLS handshake by sending a *ClientHello* message containing details about supported cipher suites, public-key parameters and key shares for key exchange. The server then computes a stateless cookie and sends it in the *HelloRetryRequest* message. Next, the client sends another *ClientHello*, but now with the cookie it received from the server, and the server replies with a *ServerHello* containing its key share and selected security

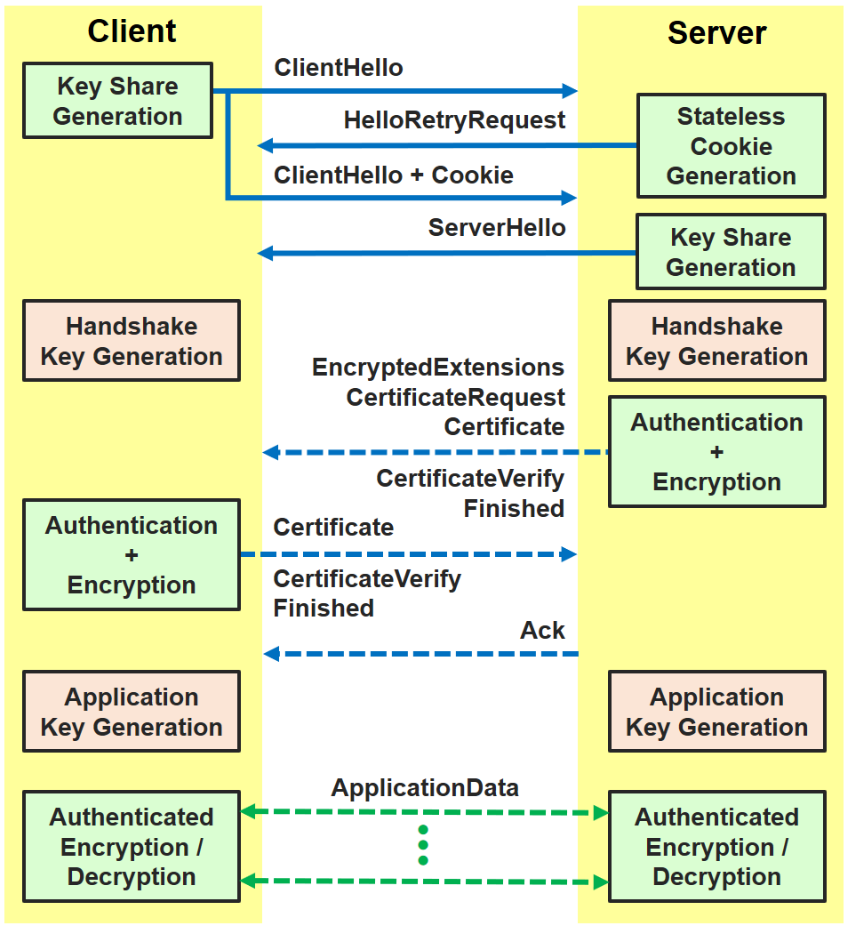


Figure 2-2: Overview of DTLS 1.3 handshake protocol with mutual authentication and Diffie-Hellman key exchange.

parameters. This procedure ensures that attackers cannot mount DoS attacks on the client or the server using forged handshake requests [6]. The remaining part of the handshake is completely encrypted using keys derived from the Diffie-Hellman shared secret. The server continues the handshake with an *EncryptedExtensions* message containing additional protocol settings, and a *CertificateRequest* message to indicate that it requires client authentication. These are followed by the server's *Certificate* and a *CertificateVerify* message that authenticates the server's side of the key exchange. The server ends this flight of messages with a *Finished* message that authenticates the handshake and confirms the security of the encrypted channel. The client replies with its own set of *Certificate*, *CertificateVerify* and *Finished* messages. Since UDP packets may be lost, the DTLS 1.3 server is required to acknowledge the receipt of this final set of messages with an *Ack* message. This ends the handshake, and the two parties can now exchange *ApplicationData* encrypted under a new set of keys derived from the handshake parameters.

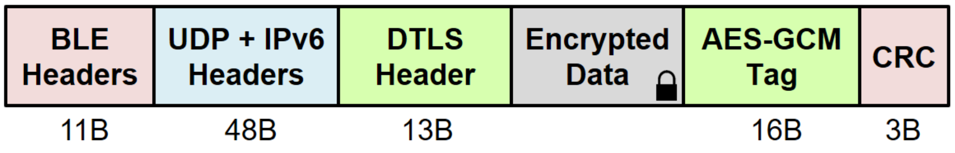


Figure 2-3: Structure of DTLS-protected BLE packet with additional headers contributed by IPv6, UDP and DTLS (all sizes in bytes).

Fig. 2-3 shows the structure of a DTLS-protected BLE packet with encrypted application data and AES-GCM tag for message authentication. Since the total BLE 4.2 payload size is restricted to 251 bytes, maximum size of the encrypted data in a single packet is 174 bytes. Larger application data get fragmented into $n = \lceil l_P/174 \rceil$ packets, where $l_P$ is the total number of bytes to be transmitted. An un-encrypted packet has very similar structure, except for the absence of the 16-byte AES-GCM tag, that is, $n = \lceil l_P/190 \rceil$.

Table 2.4 contains details of the cryptographic computations required in a typical DTLS 1.3 handshake and record protocols [8]. Table 2.5 provides a detailed analysis of the energy spent by a DTLS 1.3 client device in cryptographic computations and

RF communications, both during the handshake and the application data phases. The energy figures were computed using the models presented in the previous section. Typical sizes of DTLS handshake messages are also provided (rounded to the nearest ten bytes) [8]. (T) and (R) indicate whether packets are transmitted or received respectively. It has been assumed that the client periodically transmits 32 bytes of data in the *ApplicationData* phase, that is, after the handshake is completed.

Table 2.4: Cryptographic computations involved in DTLS 1.3.

| **Protocol Phase** | **Cryptographic Computation Details** |
|---|---|
| ClientHello (T) | 0.5 × P-256 ECDHE |
| HelloRetryRequest (R) | - |
| ClientHello + Cookie (T) | - |
| ServerHello (R) | - |
| Handshake Traffic Key Generation | 0.5 × P-256 ECDHE + 1 × SHA-256-MD (570 Bytes) + 8 × SHA-256-HMAC (32 Bytes each) |
| EncryptedExtensions + CertificateRequest (R) | 1 × AES-128-GCM-Auth-Decrypt (50 Bytes) |
| Server Certificate (R) | 1 × AES-128-GCM-Auth-Decrypt (600 Bytes) + 1 × P-384-ECDSA-Verify |
| Server CertificateVerify + Server Finished (R) | 1 × AES-128-GCM-Auth-Decrypt (130 Bytes) + 1 × P-256-ECDSA-Verify + 1 × SHA-256-MD (1300 Bytes) + 2 × SHA-256-HMAC (32 Bytes each) |
| Client Certificate (T) | 1 × AES-128-GCM-Auth-Encrypt (600 Bytes) |
| Client CertificateVerify + Client Finished (T) | 1 × AES-128-GCM-Auth-Encrypt (130 Bytes) + 1 × P-256-ECDSA-Sign + 1 × SHA-256-MD (2030 Bytes) + 2 × SHA-256-HMAC (32 Bytes each) |
| Server Ack (R) | 1 × AES-128-GCM-Auth-Decrypt (20 Bytes) |
| Application Traffic Key Generation | 1 × SHA-256-MD (1350 Bytes) + 7 × SHA-256-HMAC (32 Bytes each) |
| ApplicationData (T) | 1 × AES-128-GCM-Auth-Encrypt (32 Bytes) |

As conjectured earlier, the handshake computations are largely due to ECC. It is important to understand the effect of these expensive, but infrequent, operations on

Table 2.5: Energy costs of DTLS 1.3 – Computations and Communications.

| **Protocol Phase** | **Payload (Bytes)** | **Energy ($\mu$J)** | |
|---|---|---|---|
| | | **Compute** | **RF** |
| ClientHello (T) | 180 | 16528 | 305.4 |
| HelloRetryRequest (R) | 50 | - | 130.8 |
| ClientHello + Cookie (T) | 210 | - | 325.6 |
| ServerHello (R) | 130 | - | 173.0 |
| Handshake Traffic Key Generation | - | 16565.8 | - |
| EncryptedExtensions + CertificateRequest (R) | 50 | 6.2 | 139.2 |
| Server Certificate (R) | 600 | 70497.4 | 642.5 |
| Server CertificateVerify + Server Finished (R) | 130 | 34093.4 | 181.5 |
| Client Certificate (T) | 600 | 72.6 | 773.2 |
| Client CertificateVerify + Client Finished (T) | 130 | 12465.4 | 211.2 |
| Server Ack (R) | 20 | 2.48 | 123.4 |
| Application Traffic Key Generation | - | 69.7 | - |
| **Total Handshake Energy ($\mu$J):** | | $150.3 \times 10^3$ | $3.00 \times 10^3$ |
| ApplicationData (T) | 32 | 3.9 | 145.4 |

the total computation energy in a DTLS session. Fig. 2-4 shows the fraction of total computation energy that is spent in performing the handshake, for application data rates of 32 bytes per hour, per 30 minutes, and per 10 minutes. Session durations vary from 1 day to 365 days. With a typical data rate of 32 bytes per hour, the handshake accounts for 82% of the total computation energy in case of year-long sessions, and 99% for week-long sessions. This percentage becomes lower for faster data rates, e.g., 32 bytes per 10 minutes, which may apply to applications with real-time data requirements.

In order to analyze how much the RF transceiver contributes to the total energy consumption, we consider two prototypical scenarios – (a) session duration = 1 year (365 days), and (b) session duration = 1 week (7 days), both at the same data rate

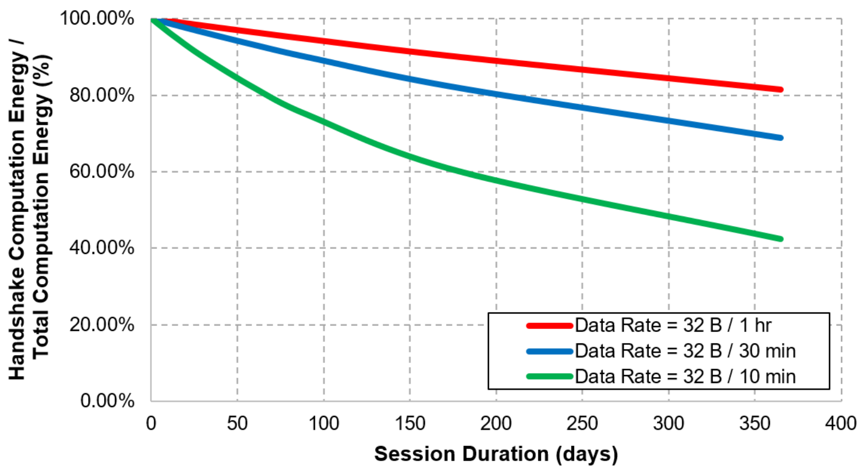


Figure 2-4: Percentage of total computation energy spent in DTLS handshake.

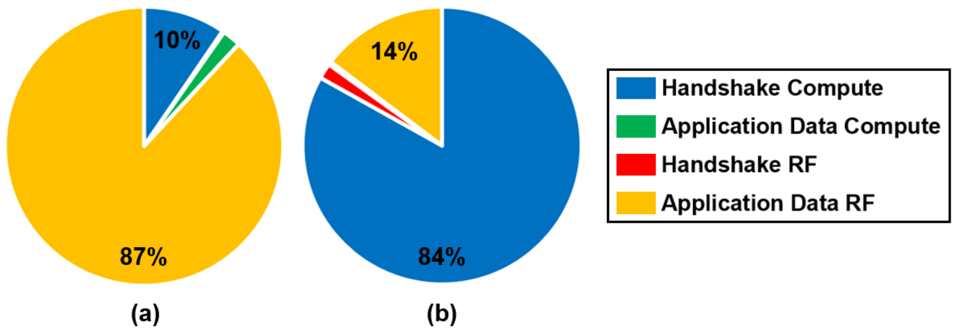


Figure 2-5: Energy breakdown of DTLS session computations and communications, for session durations of (a) 1 year and (b) 1 week, with data rate of 32 bytes per hour.

of 32 bytes per hour, so that 8760 packets are sent in (a), and 168 packets in (b). Handshake energy remains the same for both cases, but application data energy, which is proportional to the number of packets transmitted in a session, is much larger in (a). Session durations are typically determined by how often the DTLS authentication handshake is performed, which in turn depends on the security requirements of the IoT application. Fig. 2-5 shows the energy breakdown for these two use cases. Although absolute values of RF energy consumption may vary among different commercial transceivers, this analysis is sufficient to predict the energy trends in typical IoT applications. "Application Data RF" accounts for 87% of the total energy in (a), while "Handshake Compute" accounts for 84% of the total energy in (b). For both use cases, "Application Data Compute" and "Handshake RF" consume relatively negligible energy.

Therefore, it is important to reduce the energy costs of both “Application Data RF” and “Handshake Compute” in order to minimize the overall energy consumption.

## 2.4 Energy-Efficient DTLS

From the results discussed in the previous section, we conclude that different IoT applications will require different optimizations to achieve energy-efficiency. We follow a two-step approach to optimize the protocol – packet optimizations to reduce “Application Data RF” energy, and handshake optimizations to reduce “Handshake Compute” energy.

### 2.4.1 Packet Optimizations

The only way to reduce energy consumption of the RF transceiver, without modifying its circuitry or physical layer protocols, is to have smaller packets. We propose to optimize the following components of the packet structure:

- 48-byte UDP and IPv6 headers
- 13-byte DTLS header
- 16-byte AES-GCM tag

BLE headers are left untouched because we want the optimized protocol to be easily portable over different physical and link layer protocols. Also, we do not make any assumption about the application layer to allow further flexibility.

Header compression schemes have been proposed in [31], which can reduce the sizes of UDP and IPv6 headers. In this work, we exploit some properties of the network architecture to completely eliminate these headers. We assume that all sensor nodes (SN) are connected to a *gateway* device in star topology, which is the default architecture for BLE (Fig. 2-6). The gateway maintains an address translation table that maps the physical addresses of client nodes into their corresponding UDP ports. The gateway uses this table to fill in the UDP source port, and uses its own IP address

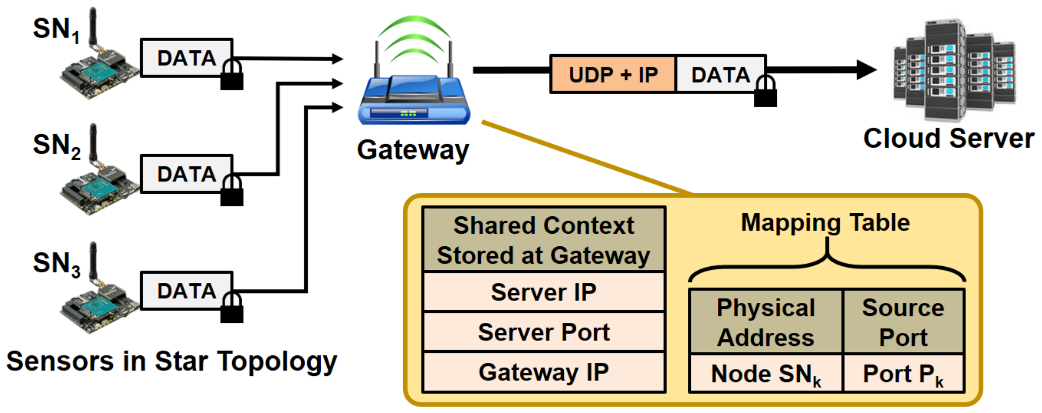


Figure 2-6: IoT network architecture, with sensor nodes and a gateway, for UDP/IPv6 packet optimizations.

as the source address. Additional shared connection context, such as the server's IP address and UDP port, is used by the gateway to fill in the UDP/IPv6 headers. This does not affect security since these headers are un-encrypted and not used in DTLS computations. This scheme requires minor modifications to the gateway software, therefore having minimal impact on existing network infrastructure.

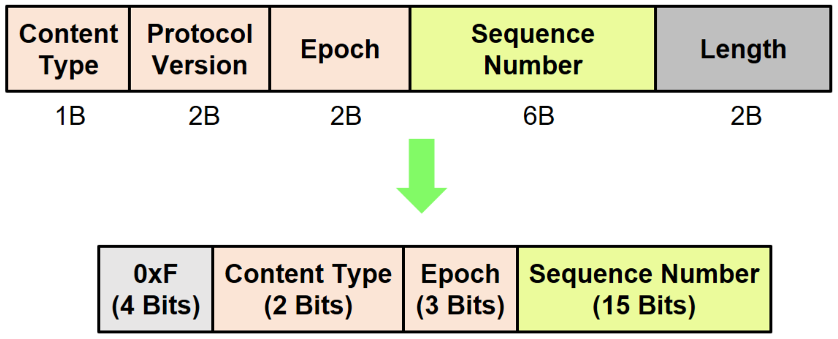


Figure 2-7: DTLS record headers – (top) Standard 13-byte header (all sizes in bytes), and (bottom) Optimized 3-byte header.

Next, we propose to reduce the 13-byte DTLS record header to a fixed size of 3 bytes (Fig. 2-7), unlike the variable-size header compression proposed in [32]. "Protocol Version" is constant, and "Length" can be inferred from the physical layer (PDU) header, so these fields are omitted. We use 2 bits for "Content Type", which has three possible values (21, 22 and 23); and 3 bits for "Epoch", which can vary from 0 to 5 (when non-forward-secret key updates are not allowed). The number of bits assigned to the sequence number is dictated by our final optimization – *truncated AES-GCM tags*. According to [13], AES-GCM tags can be truncated to 32 bits, provided the same encryption key is not used for more than $2^{15}$ packets, each up to 256 bytes in length. This sets the upper limit for the DTLS sequence number to $2^{15} - 1$, that is, 15 bits. Since the header size must be a multiple of 8 bits, the upper 4 bits are set to 0xF, which indicates that the energy-optimized variant DTLS is being used. The complete 13-byte record header is used as AAD for AES-GCM, therefore our DTLS header optimizations do not compromise security. Using truncated tags mandates performing a handshake after every $2^{15}$ transmitted packets, but this number is large enough for typical IoT applications.

Fig. 2-8 shows the optimized DTLS packet. Protocol overheads have been reduced by 91%, from 77 bytes to 7 bytes. By pushing the protocol overheads to their lower limit, we have also increased the maximum encrypted data size to 244 bytes, which further improves energy efficiency by allowing the client node to buffer sensor data, and reduces fragmentation of packets ($n = \lceil l_P/244 \rceil$ for the energy-optimized DTLS).

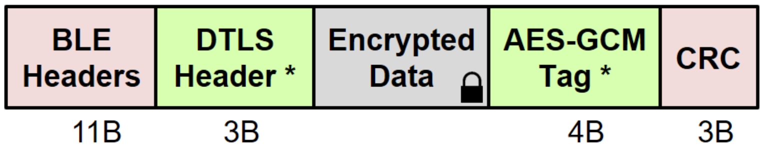


Figure 2-8: Optimized DTLS packet over BLE, with 91% reduction in protocol overheads (all sizes in bytes).

Using the BLE energy model discussed earlier, the energy required to transmit 32 bytes of encrypted application data using standard DTLS/UDP/IPv6 over BLE is calculated to be 145.4 $\mu$J, while it is 98.4 $\mu$J for our optimized DTLS scheme. Therefore, when using our optimized protocol, "Application Data RF" energy is reduced by 33%. An interesting point to note is that this percentage is smaller than the percentage reduction in number of bytes, and this is due to the additional energy consumed by the duty-cycled RF transceiver to wake up and power down.

### 2.4.2 Handshake Optimizations

ECDSA certificates constitute a bulk of the handshake energy consumption. In energy-constrained IoT applications, it is fair to assume that the client and the server can cache each other's public keys to authenticate the key exchange, and hence avoid exchanging the certificates. For this purpose, we refer to two seldom-used TLS extensions – "Client Certificate URL" [62] and "Cached Information" [63].

The "Client Certificate URL" extension allows the client to send a URL (Uniform Resource Locator) to its certificate along with the SHA-256 hash of its certificate, so that the server can look up the certificate and easily verify if it's the correct one. In this context, URL can even be used to refer to some database address on the server, where it stores all the client certificates.

The "Cached Information" extension allows the client to present the SHA-256 hash of a previously used server certificate, for which it has cached the corresponding public key, during the handshake, and the server may go ahead with that certificate if it wishes to, provided the hash is correct.

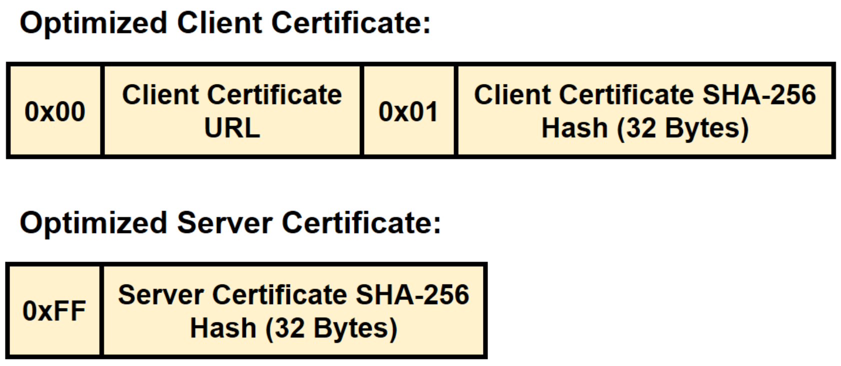

Optimized Client Certificate:
0x00
Client Certificate URL
0x01
Client Certificate SHA-256 Hash (32 Bytes)
Optimized Server Certificate:
0xFF
Server Certificate SHA-256 Hash (32 Bytes)

Figure 2-9: Optimized Certificate messages using the (top) "Client Certificate URL" [62] and (bottom) "Cached Information" [63] extensions.

These TLS extensions not only reduce the *Certificate* messages to few tens of bytes (Fig. 2-9), but also eliminate the need to verify CA signatures in the certificates, which is particularly helpful because CA signatures use higher security levels. In our case study, using these extensions can lower the handshake energy consumption by as much as 47%.

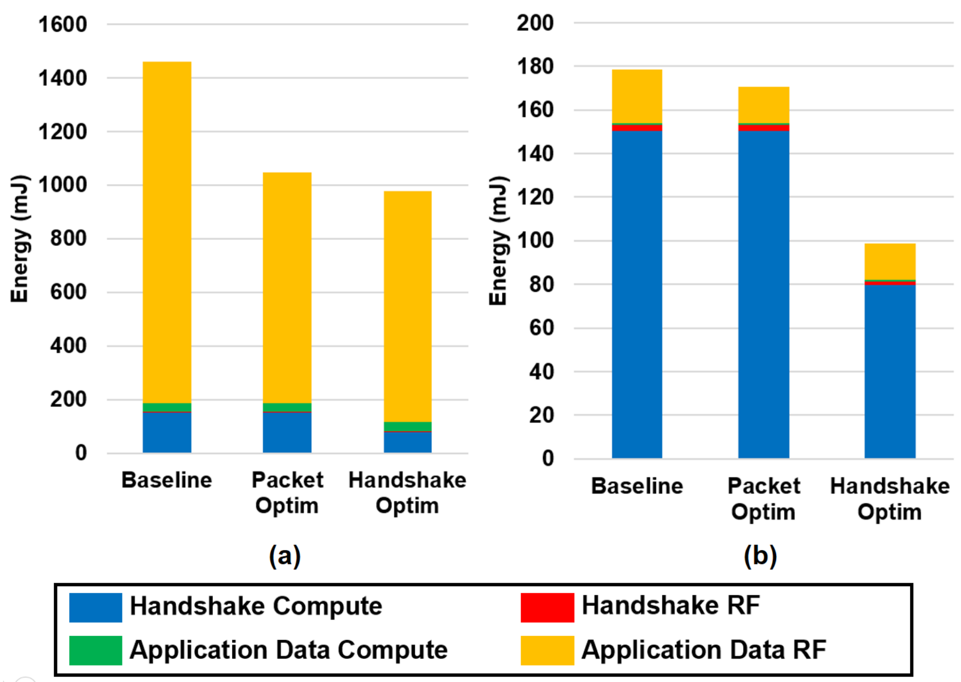


Figure 2-10: Energy benefits of the optimized DTLS, for session durations of (a) 1 year and (b) 1 week, with data rate of 32 bytes per hour.

Fig. 2-10 summarizes the energy benefits of our energy-optimized DTLS for the two test scenarios. For case (a) with 1 year session, packet optimizations provide 28% energy reduction, and handshake optimizations provide an additional 7%. For case (b) with 1 week session, packet optimizations provide only 4% energy reduction, while handshake optimizations provide 42%. Therefore, packet optimizations provide energy benefits for use cases with less frequent handshakes (a), while handshake optimizations help with applications where handshakes need to be performed more frequently (b). Overall, our optimized protocol provides 33% and 45% energy reduction respectively for (a) and (b). As mentioned earlier, these figures can vary with different IoT use-cases, micro-processors and RF transceivers, but the analysis presented in this work indicates the trends in energy consumption, along with methods to increase energy efficiency, for the typical IoT application.

For case (b), we observe that around 80% of the total energy is due to "Handshake Compute", even after our protocol optimizations. Most prospective IoT applications consider authenticating the device ("handshake") quite frequently, on the order of few days to a week. Therefore, we realize that the handshake computations will contribute largely to the energy consumption of such devices, and this motivates us to build energy-efficient dedicated cryptographic hardware for DTLS, which will be discussed in detail in Chapters 3 and 4.

# Chapter 3

# Design of Energy-Efficient Cryptographic Hardware

In Chapter 2, we have seen that cryptographic computations contribute largely to DTLS energy costs, especially when the communicating devices perform authentication handshakes frequently. This motivates the next contribution of this thesis – design of energy-efficient cryptographic hardware. This chapter presents implementation details of cryptographic primitives, and the proposed architectures are compared with previous work in literature.

## 3.1 AES-128-GCM

### 3.1.1 Encryption Algorithm

AES-128 uses 128-bit keys to encrypt 128-bit plain-text blocks, and involves 10 iteration rounds. Each round mixes the data with a *RoundKey*, which is derived from the original 128-bit encryption key. The cipher maintains an internal *State* of 128 bytes, in the form of a 4-by-4 matrix, on which the operations are performed. The initial value of the State is just the input plain-text block XOR-ed with the encryption key. The encryption rounds consist of four steps – *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*, except for the last round which does not perform MixColumns.

Fig. 3-1 summarizes the round operations involved in AES encryption.

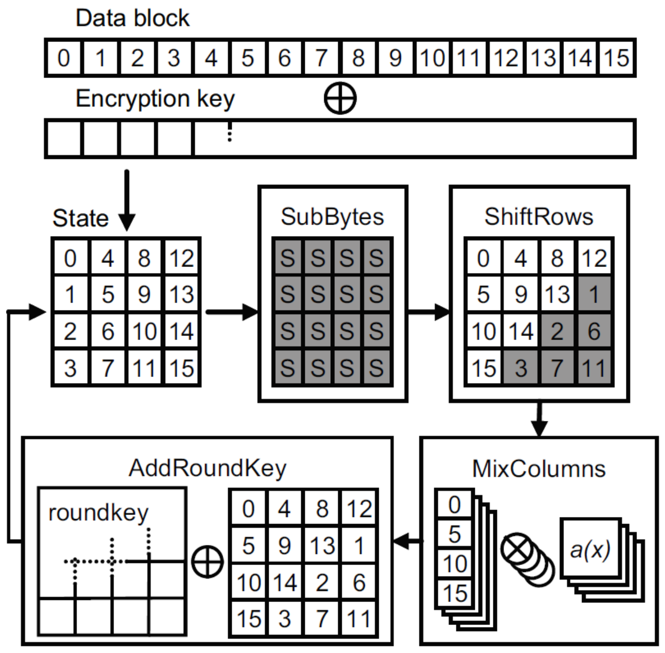


Figure 3-1: Overview of AES round operations [35].

- SubBytes is an invertible, nonlinear transformation, which uses a substitution function (S-Box) to independently map each byte of the State into a new byte. The S-Box output is generated by computing multiplicative inverse of the input in the Galois Field $GF(2^8)$ (with 0x00 being mapped to itself), and applying the following affine transformation [12]:

$$\begin{bmatrix} b'_7 \\ b'_6 \\ b'_5 \\ b'_4 \\ b'_3 \\ b'_2 \\ b'_1 \\ b'_0 \end{bmatrix} = \begin{bmatrix} 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \\ 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \end{bmatrix} \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

where $b_i$ and $b'_i$ are respectively the $i^{th}$ bits of a byte before and after the affine transformation.

- ShiftRows involves cyclic left shifts of the second, third and fourth row of the State by one, two and three bytes, respectively (Fig. 3-1).

- MixColumns transformation operates on the State column-by-column. Each column is considered as a four-term polynomial over $GF(2^8)$, and multiplied modulo $x^4+1$ with a constant polynomial $a(x) = \{03\}x^3+\{01\}x^2+\{01\}x+\{02\}$.

- AddRoundKey performs bitwise XOR of the State with the RoundKey generated using a *key expansion* algorithm (Fig. 3-2).

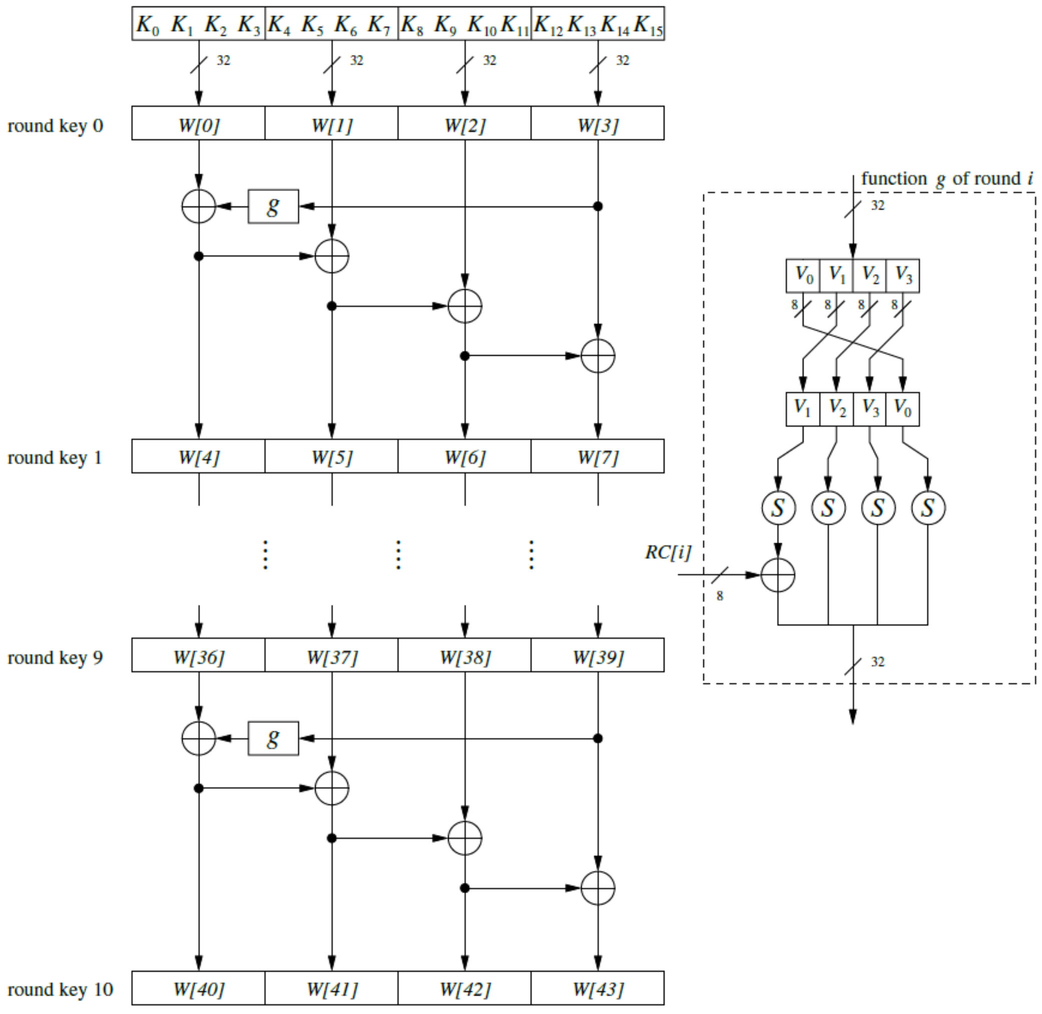


Figure 3-2: Key expansion for AES-128 [11].

The key expansion involves S-box substitutions, word rotations, and XOR operations on the encryption key. It can be performed "on-the-fly", that is, in parallel with the encryption, without any additional storage requirements for the intermediate round keys. After the final round, the State is copied to the output as cipher-text, completing the encryption process.

### 3.1.2 S-Box Design

From Figs. 3-1 and 3-2, it can be clearly seen that the S-Box is an important part of the AES algorithm – both encryption and key expansion, and each round of AES involves 20 invocations of the S-Box function. The security strength of AES is largely due to the non-linear part of the S-Box – inversion in Galois field $GF(2^8)$. The irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$ is used for all $GF(2^8)$ finite-field arithmetic in AES.

Vincent Rijmen had observed that any element in $GF(2^8)$ can be expressed as a degree-1 polynomial with coefficients in $GF(2^4)$, also known as the extension field representation $GF((2^4)^2)$. Multiplication of such elements is performed using a degree-2 irreducible polynomial, say $P(x) = x^2 + Ax + B$, where $A, B \in GF(2^4)$. Let $b_1x + b_0$ be an arbitrary element in $GF((2^4)^2)$ and let $d_1x + d_0$ be its inverse, where $b_0, b_1, d_0, d_1 \in GF(2^4)$. Then, by definition of multiplicative inverse in a finite field,

$$
\begin{aligned}
&(b_1x + b_0)(d_1x + d_0) = 1 \bmod P(x) \\
&\Rightarrow b_1d_1x^2 + (b_1d_0 + b_0d_1)x + b_0d_0 = 1 \bmod x^2 + Ax + B \\
&\Rightarrow b_1d_1(Ax + B) + (b_1d_0 + b_0d_1)x + b_0d_0 = 1 \\
&\Rightarrow (b_1d_0 + b_0d_1 + b_1d_1A)x + (b_0d_0 + b_1d_1B) = 1
\end{aligned}
$$

Therefore, $(b_1d_0 + b_0d_1 + b_1d_1A) = 0$ and $(b_0d_0 + b_1d_1B) = 1$.
Solving these two simultaneous equations, we get

$$
\begin{aligned}
&d_1 = b_1(b_1^2B + b_0^2 + b_0b_1A)^{-1} \text{ and } d_0 = (b_0 + b_1A)(b_1^2B + b_0^2 + b_0b_1A)^{-1} \\
&\Rightarrow (b_1x + b_0)^{-1} \bmod P(x) = b_1\delta x + (b_0 + b_1A)\delta, \text{ where } \delta = (b_1^2B + b_0(b_0 + b_1A))^{-1}.
\end{aligned}
$$

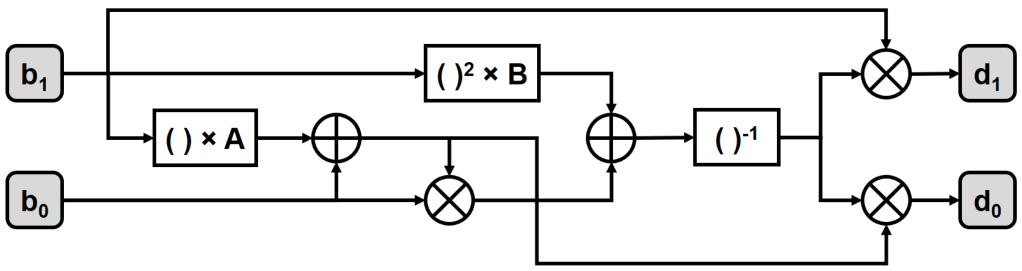


Figure 3-3: Decomposition of $GF(2^8)$ inversion into $GF(2^4)$ operations.

Hence, the inversion of an element in $GF(2^8)$ can be decomposed into 2 additions, 6 multiplications and 1 inversion in $GF(2^4)$ (Fig. 3-3). Similar decompositions can be used to compute the $GF(2^4)$ inverse in terms of $GF(2^2)$ operations, and so on.

Composite field-based S-Box designs have been studied extensively in existing literature. [34], [35], [36], [37] and [38] have all used such composite field structures. The most comprehensive analysis of S-Box architectures, using different polynomial and normal bases for $GF(2^8)$ and its isomorphic fields, has been published in [64]. The S-Box can also be implemented using look-up tables, but they are not as efficient as composite-field S-Box designs. In this work, we are going to use the low-power low-area S-Box design proposed in [64].

### 3.1.3 Energy-Efficient AES

Serial 8-bit data-path implementations of AES can provide significant area savings ([35], [37] and [38]). However, they are not the most energy-efficient. The AES algorithm operates on 128-bit blocks, but these serial designs process only 8 bits at a time, thus increasing energy consumption due to additional multiplexing and expensive register shifts. To verify this claim, we implemented three different AES architectures (Fig. 3-4):

- $A_1$, with an 8-bit data-path and one S-Box, processes the State and the RoundKey on separate cycles, 8 bit at a time. Therefore, it takes 16 cycles to generate the round key, and another 16 cycles to complete the encryption round, that is, total 336 cycles to encrypt a block.

- $A_2$, with a 128-bit data-path and 20 S-Boxes, can process the State and the RoundKey together in a single cycle. It takes 11 cycles to encrypt a block.

- $A_3$, with an unrolled pipelined 128-bit data-path containing 10 instances of AES rounds, effectively takes 1 cycle to encrypt a block.

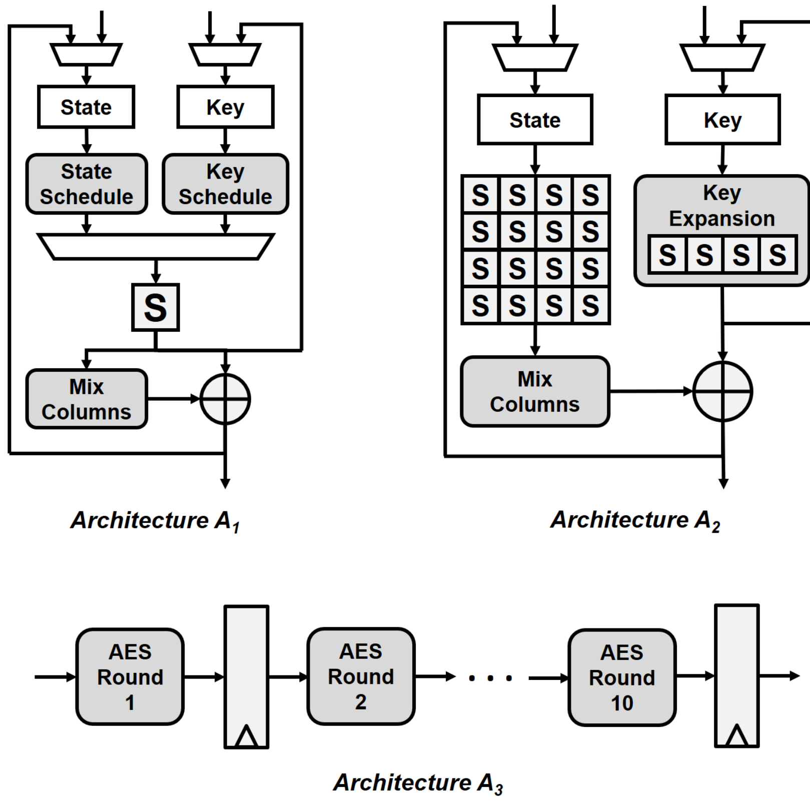


Figure 3-4: AES architectures – ($A_1$) Serial, ($A_2$) Parallel and ($A_3$) Pipelined.

Table 3.1: Comparison of our AES architectures at $V_{DD} = 1.0$ V.

| Architecture | Area | Power | Cycles | Throughput | Energy |
|---|---|---|---|---|---|
| | KGate | mW | | Gbps | pJ/bit |
| $A_1$ – Serial | 3 | 0.24 | 336 | 0.04 | 6 |
| $A_2$ – Parallel | 9.2 | 1.76 | 11 | 1.16 | 1.52 |
| $A_3$ – Pipelined | 78.8 | 16.56 | 1 | 12.8 | 1.29 |

These three designs were synthesized in the TSMC 40nm LP low-leakage technology node. They were clocked at $f_{CLK} = 100$ MHz, and verified with a set of 1000 test vectors. Table 3.1 shows the post-synthesis results. As conjectured, $A_1$ is the least energy-efficient, and the smallest in area. $A_3$ is the most energy-efficient, due to the absence of any multiplexing logic, but is also much larger than typical area budgets. $A_2$ is very close to $A_3$ in energy efficiency, while being much smaller in area, thus making it a perfect fit for our application. Fig. 3-5 shows the breakdown of area and power consumption of the different components of this design.

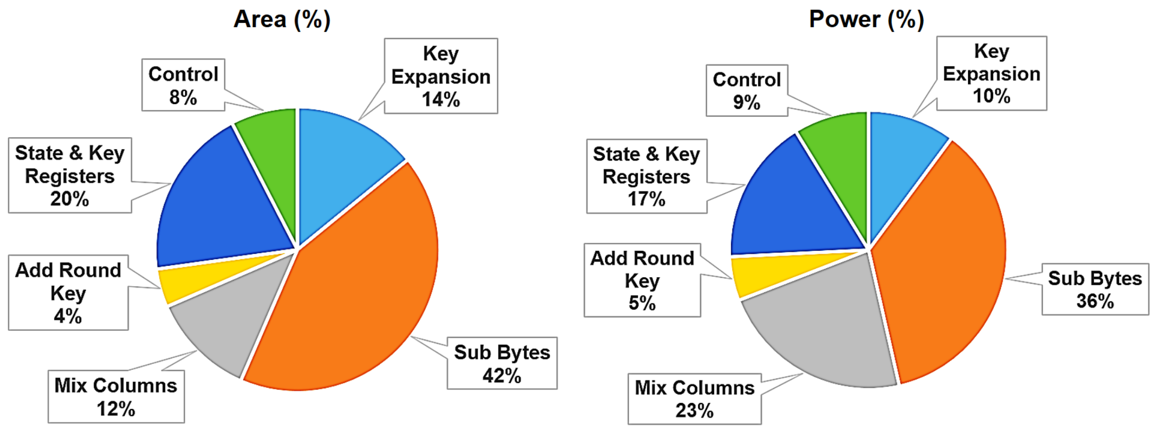


Figure 3-5: Area and power consumption distribution of the 11-cycle AES design.

Table 3.2 compares the energy consumption (pJ/bit) of our AES design ($A_2$) with previous work. All reported results are at the nominal supply voltages ($V_{DD}$) for the respective technology nodes.

Table 3.2: Comparison with previous work on AES-128 designs.

| **Design** | **Tech** | **$V_{DD}$** | **Area** | **Freq** | **Power** | **Throughput** | **Energy** |
|---|---|---|---|---|---|---|---|
| | **nm** | **V** | **KGate** | **MHz** | **mW** | **Gbps** | **pJ/bit** |
| [34]* | 110 | 1.2 | 12.5 | 145 | - | 1.7 | - |
| [35]* | 130 | 1.2 | 3.2 | 130 | 3.9 | 0.1 | 37.5 |
| [36]** | 45 | 1.1 | - | 2100 | 125 | 53.8 | 2.36 |
| [37]** | 22 | 0.9 | 1.9 | 1133 | 13 | 0.4 | 31 |
| [38]** | 40 | 0.9 | 2.3 | 1300 | 4.39 | 0.5 | 8.85 |
| **This work*** | 40 | 1.0 | 9.2 | 100 | 1.76 | 1.16 | 1.52 |

*Synthesis results **Chip results

### 3.1.4 Authenticated Encryption/Decryption

AES-GCM uses the AES forward cipher for encryption/decryption and a Galois multiplication-based special hash function, called *GHASH*, for authentication (Fig. 3-6). AES-GCM employs the counter mode of operation for encryption/decryption, which concatenates a counter value with the IV, and encrypts it with K using the AES forward cipher. The result of this encryption is then XOR-ed with the input $X$ to generate the output $Y$. For AES-GCM-Encrypt, $X = P$ and $Y = C$, while for AES-GCM-Decrypt, $X = C$ and $Y = P$, where $P$ is the plain-text and $C$ is the cipher-text. Like all counter modes, this essentially acts as a stream cipher, therefore it is important that a different $IV$ is used for each stream that is encrypted.

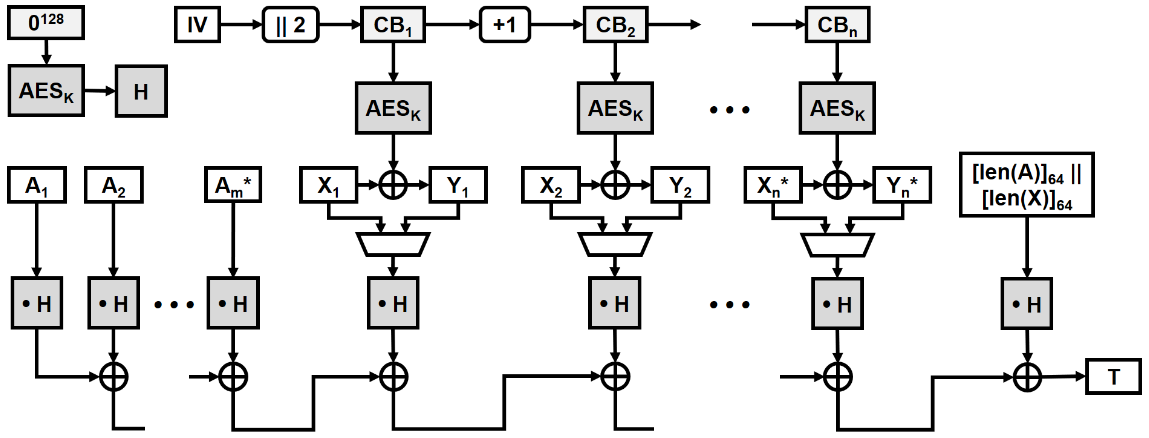


Figure 3-6: Authenticated encryption and decryption in AES-GCM.

The input $X$ is divided into 128-bit blocks as $X = X_1||X_2||\cdots||X_{n-1}||X_n^*$, where $n = \lceil len(X)/128 \rceil$ and the last block $X_n^*$ may be less than 128 bits. The initial counter block is $CB_1 = IV||0x00000002$, when $IV$ is 12 bytes long, and the *inc* function increments the counter by one for each subsequent block. These counter blocks $CB_1, CB_2, \cdots, CB_n$ are encrypted using AES, denoted as $AES_K$. The output is $Y = Y_1||Y_2||\cdots||Y_{n-1}||Y_n^*$, where the output blocks are computed as $Y_i = X_i \oplus AES_K(CB_i)$, and the final block is truncated accordingly.

The *GHASH* function uses Galois field multiplications ($\bullet$) to combine $C$ with $AAD = A_1||A_2||\cdots||Am - 1||A_m^*$, and produce an authentication tag $T$ that can be used to verify the integrity of the data. The input to *GHASH* is computed as $X =$

$AAD||0^v$ $||$ $C||0^u$ $||$ $[len(AAD)]_{64}||[len(C)]_{64}$, where $u = 128 \lceil len(C)/128 \rceil$ - $len(C)$, $v = 128 \lceil len(AAD)/128 \rceil$ - $len(AAD)$, and $[len(x)]_{64}$ denotes the length of $x$, in bits, as a 64-bit quantity. The *hash key* $H = AES_K(0^{128})$ is the output of encrypting the "zero" block with $K$ using the AES forward cipher.

### 3.1.5 Galois Multiplier Design

The Galois multiplier is an important component of GHASH computations. The algorithm below computes the Galois field product $Z = X \bullet Y$:

```
Algorithm: Galois multiplication for GHASH
Input: $X = (x_0 x_1 \cdots x_{127})_2$ and $Y = (y_0 y_1 \cdots y_{127})_2$
Output: $Z = X \bullet Y$
1. $Z_0 \leftarrow 0^{128}$, $V_0 \leftarrow Y$
2. for $i = 0$ to 127:
   2.1 if $x_i = 0$ then $Z_{i+1} \leftarrow Z_i$ else $Z_{i+1} \leftarrow Z_i \oplus V_i$
   2.2 if LSB$(V_i) = 0$ then $V_{i+1} \leftarrow V_i >> 1$ else $V_{i+1} \leftarrow (V_i >> 1) \oplus 11100001||0^{120}$
3. return $Z_{128}$
```

The Galois multiplier can be implemented in hardware using one or more copies of the basic function which we denote as $h$: $Z_{i+1} = Z_i \oplus x_i V_i$ and $V_{i+1} = (V_1 >> 1)$ $\oplus$ LSB$(V_i)(11100001||0^{120})$, that is, using loop unrolling (Fig. 3-7).

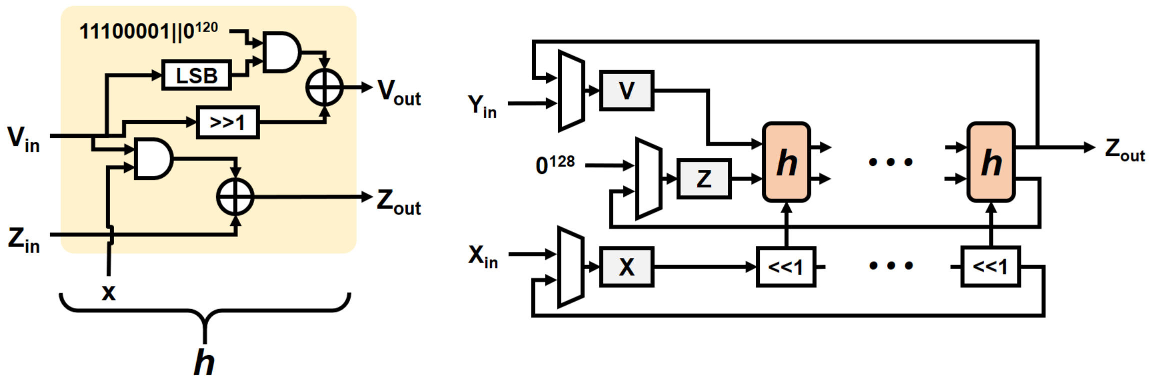


Figure 3-7: Implementation of Galois multiplier in hardware.

Multiple GHASH Galois multipliers were synthesized at $f_{CLK} = 100$ MHz, and a wide variety of area, latency and power figures were obtained (Table 3.3).

Table 3.3: Analysis of GHASH Galois multiplier architectures at $V_{DD} = 1.0$ V.

| **Architecture** | **Area** | **Power** | **Cycles** | **Throughput** | **Energy** |
|---|---|---|---|---|---|
| | **KGate** | **mW** | | **Gbps** | **pJ/Op** |
| 1-cycle | 60.1 | 5.77 | 1 | 12.8 | 57.7 |
| 2-cycle | 31.3 | 2.98 | 2 | 6.4 | 59.6 |
| 4-cycle | 16.9 | 1.74 | 4 | 3.2 | 69.6 |
| 8-cycle | 9.7 | 1.03 | 8 | 1.6 | 82.4 |
| 16-cycle | 6.1 | 0.64 | 16 | 0.8 | 102.4 |
| 32-cycle | 4.3 | 0.43 | 32 | 0.4 | 137.6 |
| 64-cycle | 3.6 | 0.33 | 64 | 0.2 | 211.2 |
| 128-cycle | 3.3 | 0.27 | 128 | 0.1 | 345.6 |

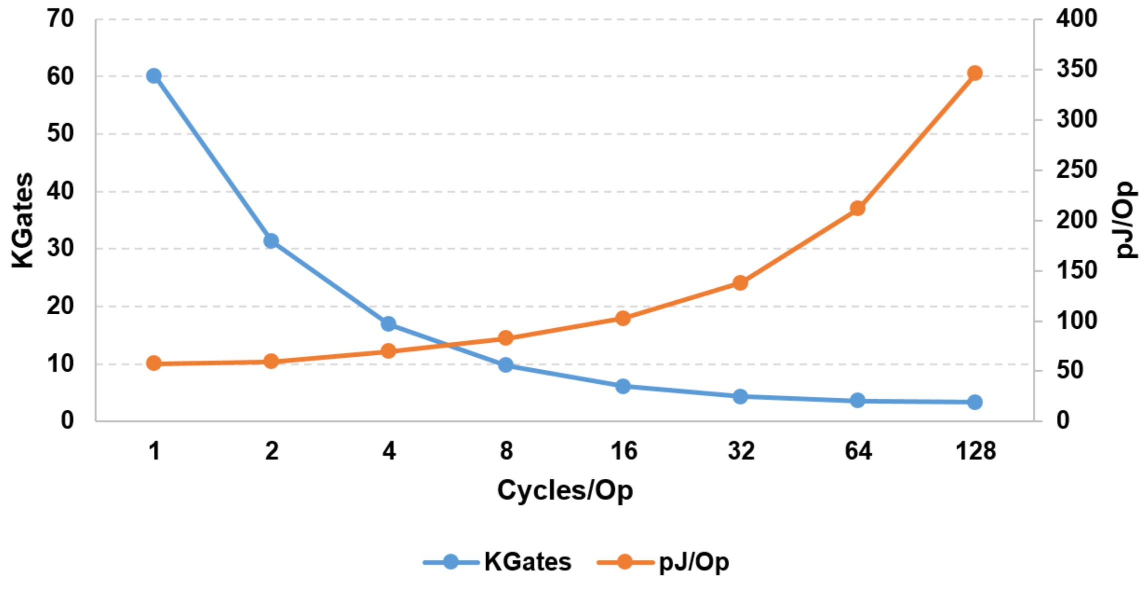


Figure 3-8: Analysis of different Galois multiplier architectures.

Fig. 3-8 shows the variations in area and energy consumption of the different Galois multiplier designs. The 8-cycle multiplier provides the perfect combination of area and energy efficiency. Faster designs are much larger, but do not offer significant energy benefits, while the slower ones are slightly smaller, but have much higher energy consumption. The increased energy efficiency in faster designs is achieved by reduction in the number of cycles required to perform the operation, along with reduced multiplexing and registers shifts.

### 3.1.6 Energy-Efficient AES-GCM

The design techniques discussed earlier have been combined to implement an energy-efficient AES-GCM unit. Table 3.4 compares synthesis results from our design with previous work.

Table 3.4: Comparison with previous work on AES-128-GCM designs.

| **Design** | **Tech** | **$V_{DD}$** | **Area** | **Freq** | **Power** | **Throughput** | **Energy** |
|---|---|---|---|---|---|---|---|
| | **nm** | **V** | **KGate** | **MHz** | **mW** | **Gbps** | **pJ/bit** |
| [39]* | 130 | 1.2 | 34 | 200 | - | 2.56 | - |
| [40]* | 65 | 1.2 | 630 | 613 | 101.2 | 19.6 | 5.16 |
| **This work*** | 40 | 1.0 | 27 | 100 | 1.63 | 0.53 | 3.06 |

*Synthesis results

## 3.2 SHA-256

### 3.2.1 Hash Algorithm

The SHA-256 hash algorithm can compress messages of arbitrary lengths ($< 2^{64}$ bits) and generate a unique 256-bit message digest. Fig. 3-9 provides an overview of the SHA-256 algorithm.

Since SHA-256 operates only on 512-bit blocks, it needs to pad the input message to a multiple of 512 bits. This is done by appending a '1' bit at the end of the message, followed by several '0' bits, and finally the 64-bit length of the message. The *State* of the hash function is initialized according to the specifications in [15]. The *Message Schedule* takes 512-bit blocks of the padded message and sends 32-bit words $W_t$ to the main SHA-256 *Round* function, along with a round constant $K_t$. Each 512-bit block is "digested" over 64 iterations of the round function, and the State is updated.

This process repeats till the entire message has been processed, and the final value of the State is the output message digest.

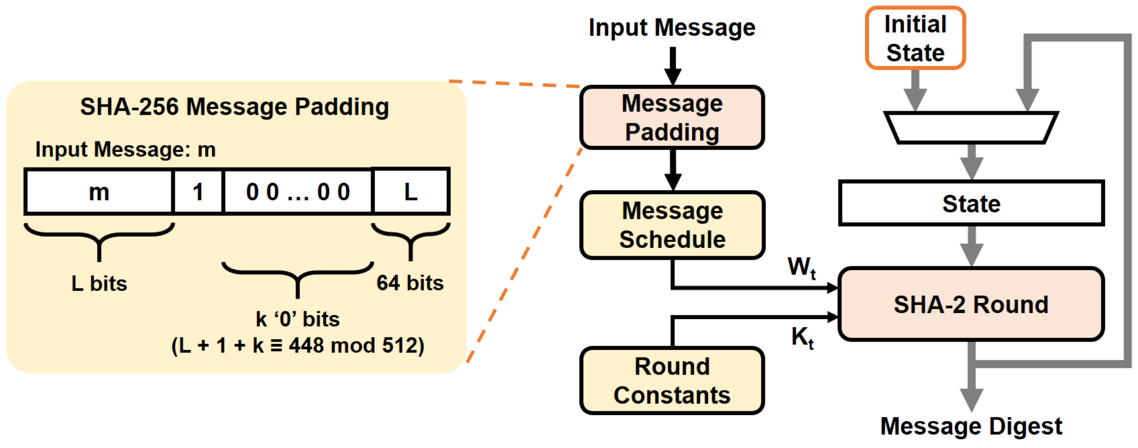


Figure 3-9: Overview of SHA-256 hash algorithm.

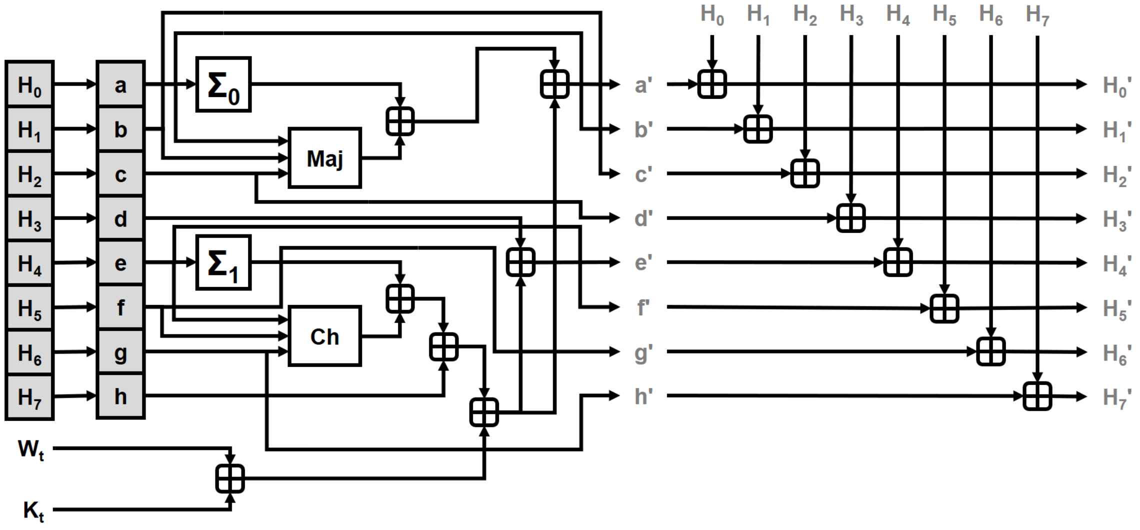


Figure 3-10: SHA-256 round function.

Fig. 3-10 shows details of the SHA-256 round function. The State consists of 16 32-bit variables $H_0 - H_7$ and $a - h$. The $Maj$, $Ch$ and $\Sigma$ functions are specified in [15]. The $\boxplus$ symbol denotes 32-bit modulo $2^{32}$ addition, that is, the final carry is ignored. Each iteration of the round function updates the state variables ($H_0' - H_7'$ and $a' - h'$ are the updated values). Although the state of the hash function is defined by $H_0 - H_7$, $a - h$ and the message schedule, it is important to note that $H_0 - H_7$

completely defines the SHA-256 state after every 64 iterations of the round, that is, after every 512-bit block has been processed. This observation can be exploited to implement efficient running hashes, as will be discussed in Chapter 4.

### 3.2.2 Energy-Efficient SHA-256

From Fig. 3-10, the critical path in the round function is: $e \rightarrow Ch \rightarrow \boxplus \rightarrow \boxplus \rightarrow \boxplus \rightarrow \boxplus \rightarrow \boxplus \rightarrow e'$. This path was implemented using a combination of "carry-save" and "ripple-carry" adders to reduce latency. The entire round function data-path was implemented, with $a' - h'$ being computed in parallel, to reduce energy consumption due to multiplexing, glitching and register shifts. Our final SHA-256 design takes 65 cycles to process a 512-bit input block, and a comparison with previous work is presented in Table 3.5. All reported results are at the nominal supply voltages ($V_{DD}$) for the respective technology nodes.

Table 3.5: Comparison with previous work on SHA-256 designs.

| **Design** | **Tech** | $\mathbf{V_{DD}}$ | **Area** | **Freq** | **Power** | **Throughput** | **Energy** |
|---|---|---|---|---|---|---|---|
| | **nm** | **V** | **KGate** | **MHz** | **mW** | **Gbps** | **pJ/bit** |
| [41]* | 180 | 1.3 | - | 819 | - | 6.4 | - |
| [42]* | 130 | 1.2 | 15.3 | 333 | - | 2.4 | - |
| [43]* | 130 | 1.2 | 22 | 794 | - | 6.0 | - |
| [44]** | 45 | 1.1 | - | 1400 | 50 | 23 | 2.17 |
| **This work*** | 40 | 1.0 | 13.3 | 100 | 1.6 | 0.79 | 2.03 |

*Synthesis results **Chip results

## 3.3 ECC

The basics of elliptic curve cryptography (ECC) were discussed in Chapter 1, and its applications, in context of the DTLS security protocol, were briefly presented in Chapter 2. The measurement results presented in Chapter 2 revealed that ECC implementations in software can be extremely power hungry. In this section, efficient hardware implementations of modular arithmetic operations will be discussed, along with point addition and doubling formulas for two types of elliptic curves – short Weierstrass and Montgomery. Different algorithms for elliptic curve scalar multiplication (ECSM) will be compared on the basis of number of computations and security implications. Finally, a highly energy-efficient reconfigurable ECC hardware design will be presented, and compared with previous work in the field of prime-field ECC implementations.

### 3.3.1 ECDH and ECDSA

As discussed in Chapter 1, the elliptic curve operations involved in ECDH are $Q_A = k_A G$, $Q_B = k_B G$ and $S = k_A Q_B = k_B Q_A = k_A k_B G$, that is, scalar multiplications. For ECDH, it is important to verify that $k \in [1, n-1]$, where $n$ is the order of the curve, before computing $Q = kG$. Also, it is important to verify that $Q$ is a valid point on the curve, before computing $S = kQ$.

A simplified explanation of ECDSA was provided in Chapter 1. Details of the ECDSA-Sign and ECDSA-Verify algorithms are discussed below [27]:

**ECDSA Parameters:** Elliptic curve $E(\mathbb{F}_p)$ with base point $G$ and order $n$, and cryptographic hash function $H$

```
Algorithm: ECDSA-Sign algorithm
Input: Private key $d \in [1, n-1]$, and message $m$
Output: Signature $(r, s)$
1. $z \leftarrow H(m)$
2. $k \in_R [1, n-1]$
3. $(x_1, y_1) \leftarrow kG$
4. $r \leftarrow x_1 \bmod n$
5. if $r = 0$ then go back to Step 2
6. $s \leftarrow k^{-1}(z + rd) \bmod n$
7. if $s = 0$ then go back to Step 2
8. return $(r, s)$
```

```
Algorithm: ECDSA-Verify algorithm
Input: Public key $Q = dG$, signature $(r, s)$, and message $m$
Output: VALID / INVALID
1. if $Q = \infty$ or $Q \notin E(\mathbb{F}_p)$ then return INVALID
2. if $r \notin [1, n-1]$ or $s \notin [1, n-1]$ then return INVALID
3. $z \leftarrow H(m)$
4. $w \leftarrow s^{-1} \bmod n$
5. $u_1 \leftarrow zw \bmod n$, $u_2 \leftarrow rw \bmod n$
6. $(x_1, y_1) \leftarrow u_1 G + u_2 Q$
7. if $r \equiv x_1 \pmod{n}$ then return VALID else return INVALID
```

The security of ECDSA relies on the fact that the same scalar $k$ is never used to sign more than one message with the same private key $d$, otherwise the private key $d$ can be easily computed from the signature values. For example, let us consider two messages $m$ and $m'$ with signatures $(r, s)$ and $(r', s')$ respectively, generated using the same scalar $k$. Then, $s - s' = k^{-1}(z - z')$, where $z = H(m)$ and $z' = H(m')$. Since $z$ and $z'$ can be easily computed by anybody, an attacker can recover the scalar $k = (z - z')(s - s')^{-1}$, and compute the private key $d = r^{-1}(sk - z)$. To prevent such attacks, it is recommended to use the "Deterministic ECDSA" algorithm specified in [66]. This modified algorithm generates the scalar $k$ from the input message hash $z$ and private key $d$. This binds the scalar to the message, and ensures that no two messages use the same scalar. The TLS specification recommends using "Deterministic ECDSA" for all ECDSA-based cipher suite implementations.

### 3.3.2 Modular Arithmetic

Modular arithmetic (addition, subtraction, multiplication and division) is an integral part of prime-field ECC computations. The first operation that we are going to discuss is prime-field addition:

```
**Algorithm:** 256-bit $GF(p)$ addition
**Input:** 256-bit integers $x$, $y$, and prime $p$
**Output:** $z = x + y \bmod p$
1. $(c_0, s_0) \leftarrow x + y$
2. $(b_1, d_1) \leftarrow s_0 - p$
3. if $c_0 = 1$ or $b_1 = 0$ then $z \leftarrow d_1$ else $z \leftarrow s_0$
4. return $z$
```

This addition algorithm adds the inputs $a$ and $b$ to get the sum $s_0$ with carry $c_0$, and subtracts $p$ from $s_0$ to get the difference $d_1$ with borrow $b_1$. If $c_0 = 1$ ($\Rightarrow a+b > 2^{256}-1$) or $b_1 = 0$ ($\Rightarrow a + b \geq p$), the correct result is $d_1$, else it is $s_0$. Prime field subtraction can also be performed using a very similar algorithm:

```
**Algorithm:** 256-bit $GF(p)$ subtraction
**Input:** 256-bit integers $x$, $y$, and prime $p$
**Output:** $z = x - y \bmod p$
1. $(b_0, d_0) \leftarrow x - y$
2. $(c_1, s_1) \leftarrow d_0 + p$
3. if $b_0 = 1$ then $z \leftarrow s_1$ else $z \leftarrow d_0$
4. return $z$
```

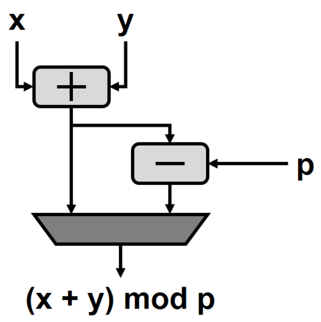


Figure 3-11: Hardware implementation of modular adder.

Fig. 3-11 shows the hardware implementation of a prime-field modular adder. It consists of one adder to add the inputs $x$ and $y$, and one subtracter to reduce the result modulo the prime $p$. Although a single adder can be used to perform the addition in one step and the reduction in the next step, it is not secure. This is because the reduction step is executed only when $x + y \geq p$, thus introducing a timing side channel.

Table 3.6: Analysis of 256-bit modular adder architectures at $V_{DD} = 1.0$ V.

| **Data-Path** | **Area** | **Power** | **Cycles** | **Energy** |
|---|---|---|---|---|
| | **KGate** | **$\mu$W** | | **pJ/Op** |
| 8-bit | 5.0 | 53.46 | 32 | 171.1 |
| 16-bit | 5.1 | 56.48 | 16 | 90.4 |
| 32-bit | 5.3 | 61.51 | 8 | 49.2 |
| 64-bit | 5.8 | 74.58 | 4 | 29.8 |
| 128-bit | 6.7 | 95.72 | 2 | 19.1 |
| 256-bit | 8.9 | 114.98 | 1 | 11.5 |

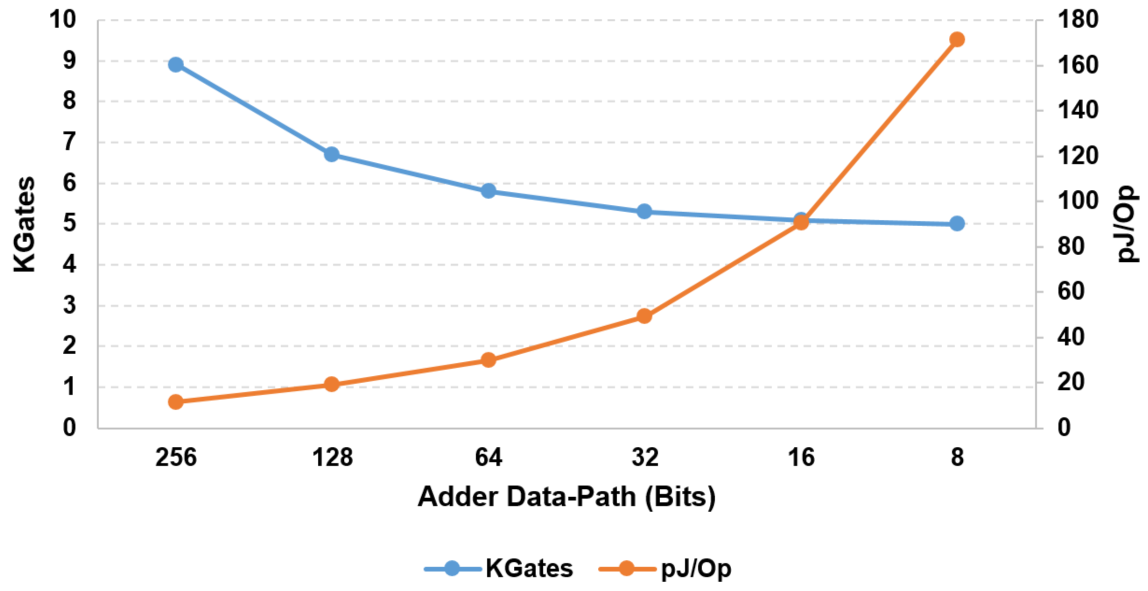


Figure 3-12: Analysis of different 256-bit modular adder architectures.

Although the inputs $x$, $y$ and $p$ to the modular adder are 256-bit quantities, the addition and subtraction operations can be performed with smaller data-paths. The modular adder shifts through appropriately sized words of the 256-bit inputs to compute the output words over multiple cycles. 256-bit modular adders of different data-path sizes were synthesized in the TSMC 40nm LP process at $f_{CLK} = 10$ MHz, and their areas and energy consumptions were compared (Table 3.6).

Fig. 3-12 shows the variations in area and energy consumption of the different 256-bit modular adders. Clearly, with reduction in data-path width, the energy consumption increases rapidly, while the area (dominated by the sequential elements, which are the same in all the adders) does not reduce significantly. Based on this analysis, we choose the 256-bit data-path adder. It is important to note that fixed-prime modular adders may use smaller data-paths to exploit special properties of the prime $p$. However, our objective is to design reconfigurable ECC hardware, hence the energy-efficient wide data-path modular adder architecture is ideal for supporting random primes.

Integer multiplications can be expressed as repeated integer additions. Similarly, modular additions can be used to perform modular multiplications. A straight-forward way to compute $a \cdot b \bmod p$ is to multiply the two 256-bit integers $a$ and $b$ to get a 512-bit product, and then reduce it modulo the 256-bit prime $p$. This method is particularly useful when the prime $p$ is fixed and has special structure that facilitates easy reduction modulo $p$. For example, the NIST primes have such special properties, as discussed in [27]. However, it is not efficient enough for hardware that can be reconfigured with random primes, and we refer to [65] for an interleaved modular multiplication algorithm that works for any prime $p$:

```
**Algorithm:** 256-bit $GF(p)$ interleaved multiplication
**Input:** 256-bit integers $x = (x_{127} \cdots x_1 x_0)_2$, $y = (y_{127} \cdots y_1 y_0)_2$, and prime $p$
**Output:** $z = x \cdot y \bmod p$
1. $z \leftarrow 0$
2. for $i = 127$ downto 0:
  2.1 $z \leftarrow 2 \cdot z + (x_i \cdot y)$
  2.2 if $z \geq p$ then $z \leftarrow z - p$
  2.3 if $z \geq p$ then $z \leftarrow z - p$
3. return $z$
```

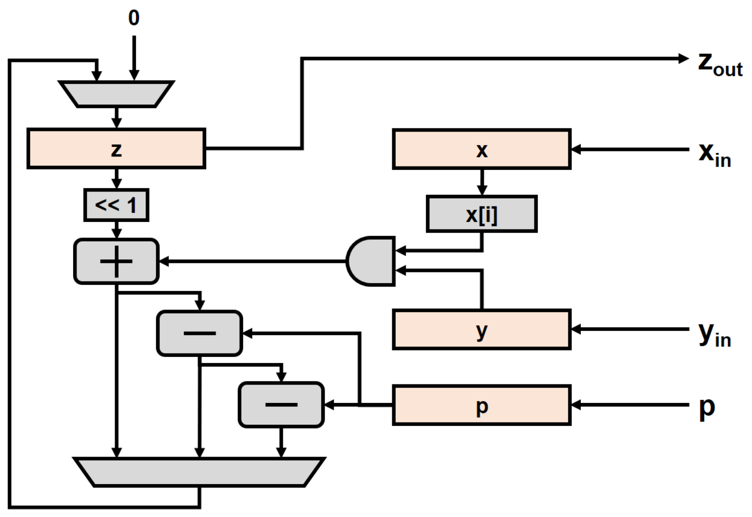


Figure 3-13: Hardware implementation of interleaved modular multiplication.

Fig. 3-13 shows the implementation of this algorithm in hardware. The input registers $x$, $y$ and $p$ are held constant throughout the multiplication process. The $i^{th}$ bit of $x$, denoted $x[i]$ in Fig. 3-13, is obtained using a 256-to-1 multiplexer. This method is used, instead of shifting the register $x$, in order to reduce the number of cycles required to perform multiplication modulo primes of smaller length, as well as to reduce energy consumption. The only register that gets updated every cycle is $z$, which also holds the final output $x \cdot y \bmod p$. The quantities $u = 2z + (x[i] \cdot y)$, $u - p$ and $u - 2p$ are computed. Since $y, z \in [0, p)$ and $x[i] \in \{0, 1\}$, we have $u \in [0, 3p)$, that is, $u \bmod p$ must be one of the three quantities $u$, $u - p$ and $u - 2p$. The updated value of $z$ is decided based on the carry of the adder and the borrows of the two subtracters. For a $t$-bit prime field, this design takes $t$ cycles to generate the result.

The final operation that will be discussed is modular division (and inversion). Integer division does not work in the finite field $\mathbb{F}_p$ because the result is a real number which may not be an element of $\mathbb{F}_p$. The equivalent operation in $\mathbb{F}_p$ is the modular division, which uses modular inversion and modular multiplication. Modular division $b/a \bmod p$, with $a, b \in \mathbb{F}_p$, is essentially $b \cdot a^{-1} \bmod p$, where $a^{-1}$ is the modular inverse of $a$. The inverse of an element $a \in \mathbb{F}_p$ is the quantity $a^{-1}$ such that $a \cdot a^{-1} = 1 \bmod p$. According to Fermat's Little Theorem, $a^{p-1} = 1 \bmod p$ for all $a \in \mathbb{F}_p$. Therefore, $a^{-1}$ can be easily computed as $a^{-1} = a^{p-2}$, that is, through repeated exponentiations.

```
Algorithm: 256-bit $GF(p)$ inversion using Fermat's Little Theorem
Input: 256-bit integer $x$, and prime $p = (p_{127} \cdots p_1 p_0)_2$
Output: $z = x^{-1} \bmod p$
1. $z \leftarrow 1$
2. for $i = 127$ down to $0$:
  2.1 $z \leftarrow z^2 \bmod p$
  2.2 if $p_i = 1$ then $z \leftarrow z \cdot x \bmod p$
3. return $z$
```

This algorithm uses modular multiplications, hence doesn't require any additional hardware. However, it involves 384 multiplications on average, which can be a significant overhead if a large number of inversions is required. An alternative method to compute $\mathbb{F}_p$ inversions is using the Extended Euclidean algorithm for integers [23]:

```
Algorithm: 256-bit $GF(p)$ inversion using Extended Euclidean algorithm
Input: 256-bit integer $x$, and prime $p$
Output: $z = x^{-1} \bmod p$
1. $u \leftarrow x$, $v \leftarrow p$
2. $t_1 \leftarrow 1$, $t_2 \leftarrow 0$
3. while $u \neq 0$:
  3.1 while $u$ even:
    3.1.1 $u \leftarrow u/2$
    3.1.2 if $t_1$ even then $t_1 \leftarrow t_1/2$ else $t_1 \leftarrow (t_1 + p)/2$
  3.2 while $v$ even:
    3.2.1 $v \leftarrow v/2$
    3.2.2 if $t_2$ even then $t_2 \leftarrow t_2/2$ else $t_2 \leftarrow (t_2 + p)/2$
  3.3 if $u \geq v$:
    then
      3.3.1 $u \leftarrow u - v$
      3.3.2 if $t_1 > t_2$ then $t_1 \leftarrow t_1 - t_2$ else $t_1 \leftarrow t_1 + p - t_2$
    else
      3.3.3 $v \leftarrow v - u$
      3.3.4 if $t_2 > t_1$ then $t_2 \leftarrow t_2 - t_1$ else $t_2 \leftarrow t_2 + p - t_1$
4. return $t_2$
```

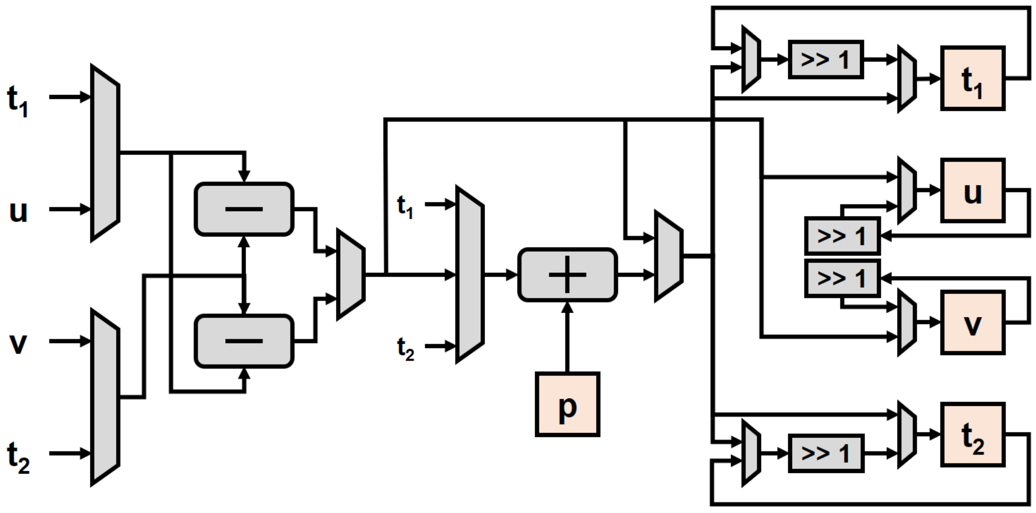


Figure 3-14: Hardware implementation of modular inversion using Extended Euclidean algorithm.

The steps 3.1 and 3.2 are never executed together, because the algorithm ensures that $u$ and $v$ are never both even. The above algorithm can also be used to compute $y \cdot x^{-1} \bmod p$ by setting $t_1 \leftarrow y$ in step 1. The algorithm has been modified slightly from [23] to prevent the intermediate variables $u$, $v$, $t_1$ and $t_2$ from becoming negative. Implementation of this algorithm definitely requires additional hardware, and our implementation is shown in Fig. 3-14. It performs steps 3.1 and 3.2 together, and then 3.3 in two cycles, using two subtracters and one adder (with one of the inputs $p$). Table 3.7 analyzes the post-synthesis energy consumption of various 256-bit modular arithmetic operations using the architectures discussed earlier.

Although Fermat's Theorem-based inversion requires minimal hardware apart from the modular multiplier, it is very slow and consumes a lot of energy. On the other hand, inversion using Extended Euclidean algorithm is faster and far more energy-efficient, but requires significant amounts of additional circuitry. Denoting $\mathbb{F}_p$ multiplication by $M$ and $\mathbb{F}_p$ inversion by $I$, we observe that $I_{Fermat} \approx 384M$, while $I_{Euclid} \approx 4M$. This

Table 3.7: Analysis of 256-bit modular arithmetic operations at $V_{DD} = 1.0$ V.

| **Operation** | **Area** | **Cycles** | **Energy** |
|---|---|---|---|
| | **KGate** | | **nJ/Op** |
| $\mathbb{F}_p$ Addition/Subtraction | 8.9 | 1 | 0.01 |
| $\mathbb{F}_p$ Interleaved Multiplication | 11.6 | 256 | 2.36 |
| $\mathbb{F}_p$ Inversion (Fermat's Theorem) | 12.1 | 98304 | 906 |
| $\mathbb{F}_p$ Inversion (Extended Euclid) | 21.5 | 720 | 9.5 |

observation will be very important in deciding the most efficient elliptic curve point formulas.

### 3.3.3 Point Addition and Doubling

In this research, we are going to focus on prime-field ECC using only two types of elliptic curves – short Weierstrass and Montgomery. The fundamental operations used in ECC are point addition ($R = P + Q$), and point doubling ($R = P + P$). The corresponding formulas for these curves are listed as follows:

- For the short Weierstrass curve $y^2 = x^3 + ax + b$,

  Point addition: $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$
  where $x_3 = \lambda^2 - x_1 - x_2$, $y_3 = \lambda(x_1 - x_3) - y_1$ and $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$

  Point doubling: $2(x_1, y_1) = (x_3, y_3)$
  where $x_3 = \lambda^2 - 2x_1$, $y_3 = \lambda(x_1 - x_3) - y_1$ and $\lambda = \frac{3x_1^2 + a}{2y_1}$

- For the Montgomery curve $by^2 = x^3 + ax^2 + x$,

  Point addition: $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$
  where $x_3 = b\lambda^2 - a - x_1 - x_2$, $y_3 = \lambda(x_1 - x_3) - y_1$ and $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$

  Point doubling: $2(x_1, y_1) = (x_3, y_3)$
  where $x_3 = b\lambda^2 - a - 2x_1$, $y_3 = \lambda(x_1 - x_3) - y_1$ and $\lambda = \frac{(3x_1 + 2a)x_1 + 1}{2by_1}$

where all operations are performed in the finite field $\mathbb{F}_p$.

These formulas use the affine representation of point coordinates. For short Weierstrass curves, point addition involves $2 \times M$ and $1 \times I$, while point doubling involves $3 \times M$ and $1 \times I$. Similarly, for Montgomery curves with $b = 1$, point addition involves $2 \times M$ and $1 \times I$, while point doubling involves $3 \times M$ and $1 \times I$. Modular additions and subtractions are ignored for this analysis, because they are much cheaper than multiplications ($M$) and inversions ($I$).

Projective coordinates represent the point $(x, y)$ as $(X, Y, Z)$, where $x = X/Z^2$ and $y = Y/Z^3$. This prevents expensive modular inversions because this representation keeps track of the denominator separately as the third coordinate $Z$. Projective coordinates avoid computing modular inversions at the cost of an increased number of modular multiplications. For short Weierstrass curves, the following Jacobian projective formulas are used:

- Point addition: $R(X_3, Y_3, Z_3) = P(X_1, Y_1, Z_1) + Q(x_2, y_2)$
  $X_3 = (y_2Z_1^3 - Y_1)^2 - (x_2Z_1^2 - X_1)^2(X_1 + x_2Z_1^2)$
  $Y_3 = (y_2Z_1^3 - Y_1)^2(X_1(x_2Z_1^2 - X_1)^2 - X_3) - Y_1(x_2Z_1^2 - X_1)^3$
  $Z_3 = (x_2Z_1^2 - X_1)Z_1$

- Point doubling: $R(X_3, Y_3, Z_3) = 2P(X_1, Y_1, Z_1)$
  $X_3 = 9(X_1^2 - Z_1^4)^2 - 8X_1Y_1^2$
  $Y_3 = 3(X_1^2 - Z_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4$
  $Z_3 = 2Y_1Z_1$

Therefore, point addition over mixed coordinates involves $11 \times M$, while point doubling over projective coordinates involves $8 \times M$. For Montgomery curves with $b = 1$, the Montgomery ladder formula is used to compute point multiplications using projective coordinates, and each double-and-add "ladder step" involves $10 \times M$ [57]. In the next section, we will determine which set of formulas is best suited for our hardware implementations of modular arithmetic.

### 3.3.4 Scalar Multiplication Formulas

The simplest algorithm that can be used to compute $Q = kP$ for any elliptic curve point $P$ is the "left-to-right double-and-add" method described below:

```
Algorithm: $E(\mathbb{F}_p)$ point multiplication using left-to-right double-and-add method
Input: $t$-bit scalar $k = (k_{t-1} \cdots k_1 k_0)_2$, and point $P \in E(\mathbb{F}_p)$
Output: $Q = kP$
1. $Q \leftarrow \infty$
2. for $i = t - 1$ downto 0:
  2.1 $Q \leftarrow 2Q$
  2.2 if $k_i = 1$ then $Q \leftarrow Q + P$
3. return $Q$
```

This algorithm uses only one additional variable $Q$, which also holds the final result. Therefore, no additional registers are required apart from the input point, the output point and the scalar.

An important observation here is that Step 2.2 is executed only when $k_i = 1$, that is, point doubling is always performed, while point addition is performed depending on the corresponding bit in the scalar $k$. We have seen earlier that point addition and point doubling are very different computations. The affine point doubling operation uses one extra multiplication, compared to affine point addition. Therefore, an attacker snooping on the target device can easily figure out the bits of $k$. This is called a *side-channel attack*, and is a major concern regarding secure ECC implementations. The execution time of the algorithm, in software or hardware, can be used as a side channel. Another common side-channel is the power supply ($V_{DD}$) pin of the device, and it is called a "Simple Power Analysis" (SPA) attack. We will discuss countermeasures to prevent such side-channel attacks in the next section.

The $t$-bit scalar $k = (k_{t-1} \cdots k_1 k_0)_2$ has $t/2$ ones on average. Therefore, point multiplication using the simple double-and-add algorithm involves $t$ doublings (DBL) and $t/2$ additions (ADD). There are different windowing methods that process multiple bits of $k$ at a time, but they only reduce the number of additions [23]. The "comb" method, discussed below, employs clever pre-computations to reduce the number of doublings and additions [23]:

```
Algorithm: $E(\mathbb{F}_p)$ point multiplication using fixed-base comb method
Input: $t$-bit scalar $k = (k_{t-1} \cdots k_1 k_0)_2$, window width $w$, $d = \lceil t/w \rceil$, and point $P \in E(\mathbb{F}_p)$
Output: $Q = kP$
1. Pre-computations – For all bit-strings $(a_{w-1}, \cdots, a_1, a_0)$, compute
     $[a_{w-1}, \cdots, a_1, a_0]P = a_{w-1} 2^{(w-1)d} P + \cdots + a_2 2^{2d} P + a_1 2^d P + a_0 P$
2. Pad $k$ to the left with zeros, if necessary, and write $k = K^{w-1} || \cdots || K^1 || K^0$, where each $K^j$ is a bit-string of length $d$, and let $K_i^j$ denote the ith bit of $K^j$
3. $Q \leftarrow \infty$
4. for $i = d - 1$ downto 0:
   4.1 $Q \leftarrow 2Q$
   4.2 if $k_i = 1$ then $Q \leftarrow Q + [K_i^{w-1}, \cdots, K_i^1, K_i^0]P$
5. return $Q$
```

Considering $t = 256$ and $w = 4$ ($\Rightarrow d = 64$), the pre-computations involve 192 DBL and 16 ADD, while the ECSM computation involves 64 DBL and 64 ADD. This is a significant improvement from the windowed double-and-add methods. The only additional overhead is the storage requirement for the $2^w$ pre-computed points.

While affine coordinate computations perform a modular inversion at the end of every DBL and ADD, the projective coordinate computations perform only one inversion at the end of the ECSM. Projective coordinates are preferred for software implementations because conventional Fermat's Theorem-based modular inversions are extremely expensive. However, we have proposed a highly efficient Extended Euclidean algorithm-based modular inversion architecture in a previous section, and we are going to use our analysis of $I_{Fermat} = 384M$ versus $I_{Euclid} = 4M$ to prove that affine coordinates are best suited for our implementation.

Table 3.8 compares the total amount of computations required for $w = 4$ comb-based 256-bit prime-field ECSM using both types of coordinates for the two types of elliptic curves. Using affine coordinates with modular inversion based on Extended Euclidean algorithm, we can reduce the ECSM computation costs by 48% for short

Table 3.8: Analysis of 256-bit prime-field ECSM using $w = 4$ comb.

| **Elliptic Curve** | **ECSM** | | | |
|---|---|---|---|---|
| | **Affine** | | **Projective** | |
| | $I_{Fermat}$ | $I_{Euclid}$ | $I_{Fermat}$ | $I_{Euclid}$ |
| Short Weierstrass | $49472 \times M$ | $832 \times M$ | $1600 \times M$ | $1220 \times M$ |
| Montgomery ($b = 1$) | $49472 \times M$ | $832 \times M$ | $1024 \times M$ | $644 \times M$ |

Weierstrass curves and 19% for Montgomery curves. Although projective coordinates with our efficient modular inverter can provide 37% savings for Montgomery curves, we choose to use only the affine form in order to avoid adding area overheads due to the projective coordinate state machine and related circuitry. Another important benefit of using affine coordinates is the reduction in the number of temporary registers. The projective coordinate algorithms in [23] use up to 4 temporary registers ($T_1$, $T_2$, $T_3$, $T_4$), which are no longer necessary in our hardware. This reduces area as well as energy consumption due to register multiplexing.

### 3.3.5 Side-Channel Security

Side-channel security is particularly essential for public-key cryptography algorithms like ECC, because of their long computation times. Although the comb method processes multiple bits of the scalar $k$ simultaneously, it is not free from side-channel attacks. A countermeasure has been proposed in [67] that can prevent SPA attacks on ECSM using the comb method. This countermeasure uses a modified representation of the scalar $k$, called “Zero-less Signed Digit” (ZSD) representation. The proposed algorithm replaces every zero bit of the scalar $k$ by 1 or -1, depending of its neighbor bits. Assuming that $k$ is odd, let $k_i$ be the first bit equal to 0. Then, $k_i$ is replaced by $k_i + 1 = 1$, and $k_{i-1}$ is replaced by $k_{i-1} - 2 = -k_{i-1} = -1$, which does not change the value of $k$. Iterating this process, every bit of the bit-string will be non-zero and the multiplication $Q = kP$ will be SPA-resistant. Since the scalar does not contain any zeros, this also reduces the number of pre-computed comb points from $2^w$ to $2^{w-1}$ [67]. It is to be noted that this process works only for odd integers $k$, that is, the least significant bit $k_0 = 1$. To prevent leaking any information about whether $k$ is even or odd, we initially compute $k' = k + 1$ if $k$ is even, and $k' = k + 2$ if $k$ is odd. Then, $Q' = k'P$ is computed, and finally, we obtain $Q = kP$ as $Q' - P$ if $k$ is even, and $Q' - 2P$ if $k$ is odd.

Side-channel attacks on ECC are not limited to simple SPA attacks. Another form of attack, called "Differential Power Analysis" (DPA), uses statistical techniques to extract the secret scalar $k$ from measurements taken over several point multiplications $Q = kP$. Such attacks can be prevented using some form of randomization [68], and we generate another random scalar $l \in_R [1, n-1]$. Then, we compute $Q_1 = k_1 P$ and $Q_2 = k_2 P$, where $k_1 = l$ and $k_2 = k - l \bmod n$. Finally, we compute $Q = Q_1 + Q_2 = kP$. An implementation is DPA-secure only when it is also SPA-secure. Therefore, the scalars $k_1$ and $k_2$ are converted to ZSD form to prevent SPA attacks on the point multiplications. Clearly, the DPA-secure algorithm uses two point multiplications to compute the result, therefore, it is twice as expensive as the SPA-secure algorithm.

### 3.3.6 Energy-Efficient ECC

The design techniques discussed earlier have been used to implement an energy-efficient ECC hardware. It supports ECDH, ECDSA-Sign and ECDSA-Verify, along with prime-field modular arithmetic. It can be reconfigured to support any short Weierstrass or Montgomery curve over any prime field up to 256 bits.

The ECC module contains two major execution units – the serial modular multiplier, which is also used to perform modular addition, and the modular inverter. Since all elliptic curve operations can eventually be expressed as a combination of different modular arithmetic operations, we have designed a hardware state machine along with the necessary storage elements to execute the ECC operations using the modular arithmetic units.

Fig. 3-15 shows the architecture of our reconfigurable prime-field ECC hardware. The registers $p$ and $n$ store the prime and the curve order respectively. The registers $x_1$ and $y_1$ holds the final output point or signature value. The modular inverter, based on the binary Extended Euclidean algorithm, contains the four registers $u$, $v$, $t_1$, $t_2$, and supports division and inversion modulo either $p$ or $n$. During modular multiplication,

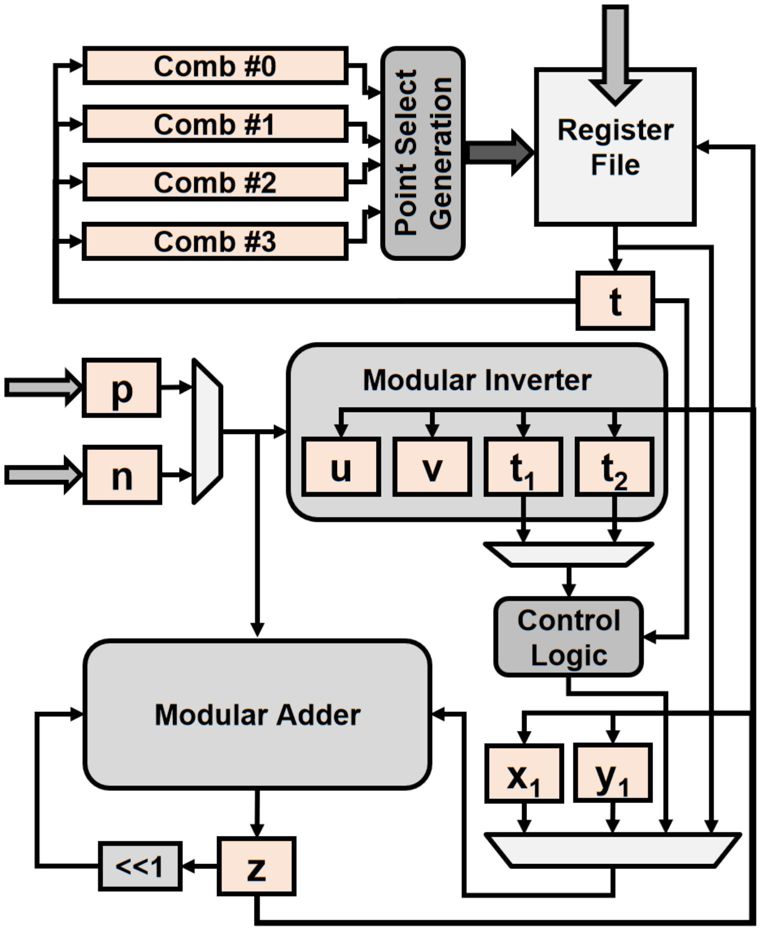


Figure 3-15: Architecture of the reconfigurable prime-field ECC hardware.

the multiplier value is stored in the register $t$ and the multiplicand is either $t_1$ or $t_2$. The modular multiplier performs interleaved modular multiplication, as discussed earlier, and stores the final value in the register $z$.

Point doubling (DBL) and point addition (ADD) are the basic operations used in ECC, and they are combined with modular arithmetic to define the ECC functions like ECDH and ECDSA. The state machines for the DBL and ADD operations are based on carefully optimized micro instructions to reduce register multiplexing and execution cycles, as described in Figs. 3-16 and 3-17. These micro-instructions have been specifically designed for our ECC architecture. All micro-instructions are register-based, and only the $z$ register is allowed to write back to the other registers (and the register file). This reduces control complexity, thus reducing energy consumption of the hardware. For a 256-bit short Weierstrass curve, DBL takes $\approx$ 1530 cycles and

```
1.  SKIP IF $P_1 = \infty$
2.  $(x_1, y_1) \leftarrow P_1$ ; $z \leftarrow 0$
3.  $(z = 0)\ z \leftarrow 2 \times z + y_1$
4.  $u \leftarrow z$ ; CHECK IF $u = 0$
5.  $z \leftarrow 2 \times z + 0$
6.  IF (CURVE = $Montgomery$ AND $b \neq 1$)
        $t \leftarrow z$ ; $z \leftarrow 0$
        $(z = 0)\ z \leftarrow 2 \times z + b$
        $t_2 \leftarrow z; z \leftarrow 0$
        $z \leftarrow t \times t_2$
7.  $u \leftarrow z$ ; $z \leftarrow 0$
8.  $(z = 0)\ z \leftarrow 2 \times z + x_1$
9.  $t_2 \leftarrow z$
10. IF (CURVE = $Montgomery$)
        $t_1 \leftarrow z$ ; $LSB \leftarrow z[0]$ ; $z \leftarrow 0$
        $t_1 \leftarrow t_1 >> 1$
        $(z = 0)\ z \leftarrow 2 \times z + t_1$
        $z \leftarrow 2 \times z + LSB + a$
11. $z \leftarrow 2 \times z + x_1$
12. $t \leftarrow z$ ; $z \leftarrow 0$
13. $z \leftarrow t \times t_2$
14. $t_1 \leftarrow z$ ; $LSB \leftarrow z[0]$ ; $z \leftarrow 0$
15. IF (CURVE = $Montgomery$)
        $(z = 0)\ z \leftarrow 2 \times z + 1 + t_1$
16. IF (CURVE = $Weierstrass$)
        $t_1 \leftarrow t_1 >> 1$
        $(z = 0)\ z \leftarrow 2 \times z + t_1$
        $z \leftarrow 2 \times z + LSB + a$
17. $t_1 \leftarrow z$; $z \leftarrow 0$; $v \leftarrow p$
18. $t_2 \leftarrow$ INV($p$, DIV = $True$) ; $z \leftarrow 0$
19. $(z = 0)\ z \leftarrow 2 \times z + t_2$
20. $t \leftarrow z$ ; $z \leftarrow 0$
21. $z \leftarrow t \times t_2$
22. $t_1 \leftarrow z$ ; $LSB \leftarrow z[0]$ ; $z \leftarrow 0$
23. IF (CURVE = $Montgomery$ AND $b \neq 1$)
        $(z = 0)\ z \leftarrow 2 \times z + b$
        $t \leftarrow z$ ; $z \leftarrow 0$
        $z \leftarrow t \times t_1$
        $t_1 \leftarrow z$ ; $LSB \leftarrow z[0]$ ; $z \leftarrow 0$
24. $t_1 \leftarrow t_1 >> 1$
25. $(z = 0)\ z \leftarrow 2 \times z + t_1$
26. $z \leftarrow 2 \times z + LSB - x_1$
27. $t_1 \leftarrow z$ ; $LSB \leftarrow z[0]$ ; $z \leftarrow 0$
28. $t_1 \leftarrow t_1 >> 1$
29. $(z = 0)\ z \leftarrow 2 \times z + t_1$
30. $z \leftarrow 2 \times z + LSB - x_1$
31. $t_1 \leftarrow z$ ; $LSB \leftarrow z[0]$ ; $z \leftarrow 0$
32. IF (CURVE = $Montgomery$)
        $t_1 \leftarrow t_1 >> 1$
        $(z = 0)\ z \leftarrow 2 \times z + t_1$
        $z \leftarrow 2 \times z + LSB - a$
        $t_1 \leftarrow z$ ; $LSB \leftarrow z[0]$ ; $z \leftarrow 0$
33. $t_1 \leftarrow t_1 >> 1$
34. $(z = 0)\ z \leftarrow 2 \times z + t_1$
35. $z \leftarrow 2 \times z + LSB - x_1$
36. $t \leftarrow z$ ; $z \leftarrow 0$
37. $(z = 0)\ z \leftarrow 2 \times z + t_1$
38. $z \leftarrow 2 \times z + LSB + 0$
39. $x_1 \leftarrow z$ ; $z \leftarrow 0$
40. $z \leftarrow t \times t_2$
41. $t_1 \leftarrow z$ ; $LSB \leftarrow z[0]$ ; $z \leftarrow 0$
42. $t_1 \leftarrow t_1 >> 1$
43. $(z = 0)\ z \leftarrow 2 \times z + t_1$
44. $z \leftarrow 2 \times z + LSB + y_1$
45. $t_1 \leftarrow z$ ; $z \leftarrow 0$
46. $(z = 0)\ z \leftarrow 2 \times z - t_1$
47. $y_1 \leftarrow z$ ; $z \leftarrow 0$
48. $P_3 \leftarrow (x_1, y_1)$
```

Figure 3-16: Micro-code used for point doubling $P_3 = 2P_1$ using the reconfigurable prime-field ECC module.

ADD takes $\approx 1280$ cycles. The point $P_1$ is loaded into registers $x_1$ and $y_1$, which also hold the output point $P_3$. The second point for ADD is read from the register file.

For ECSM, the point $(P_x, P_y)$ is one of the 8 pre-computed comb points. The ECSM secret scalar $k$ is loaded into register $t$ from the register file, and then shifted into the comb registers COMB$_i$ ($i \in [0, 3]$). For SPA security, the scalar $k$ is converted to ZSD form while being loaded into the COMB$_i$ registers. For easy "on-the-fly" conversion of $k$ from binary to ZSD, we use a special representation of ZSD where '1' represents '1' and '0' represents '-1'. As discussed earlier, the binary scalar $k = (k_{t-1}, k_{t-2}, \cdots, k_1, k_0)_2$

```
1.  SKIP IF $P_2 = \infty$
2.  $(x_1, y_1) \leftarrow P_1$ ; $z \leftarrow 0$
3.  IF $(P_1 = \infty)$
        $(z=0)\ z \leftarrow 2 \times z + P_x$
        $x_1 \leftarrow z$ ; $z \leftarrow 0$
        $(z=0)\ z \leftarrow 2 \times z \pm P_y$
        $y_1 \leftarrow z$ ; $z \leftarrow 0$
        SKIP
4.  $(z=0)\ z \leftarrow 2 \times z + y_1$
5.  $t_1 \leftarrow z$ ; $z \leftarrow 0$ ; $LSB \leftarrow z[0]$
6.  $t_1 \leftarrow t_1 >> 1$
7.  $(z=0)\ z \leftarrow 2 \times z + t_1$
8.  $z \leftarrow 2 \times z + LSB \pm P_y$
9.  $u \leftarrow z$ ; CHECK IF $u = 0$
10. $t_1 \leftarrow z$ ; $z \leftarrow 0$
11. $(z=0)\ z \leftarrow 2 \times z + x_1$
12. $t_2 \leftarrow z$ ; $z \leftarrow 0$ ; $LSB \leftarrow z[0]$
13. $t_2 \leftarrow t_2 >> 1$
14. $(z=0)\ z \leftarrow 2 \times z + t_2$
15. $z \leftarrow 2 \times z + LSB - P_x$
16. $u \leftarrow z$ ; CHECK IF $u = 0$ ; $v \leftarrow p$
17. $t_2 \leftarrow$ INV($p$, DIV $=$ $True$) ; $z \leftarrow 0$
18. $(z=0)\ z \leftarrow 2 \times z + t_2$
19. $t \leftarrow z$ ; $z \leftarrow 0$
20. $z \leftarrow t \times t_2$
21. $t_1 \leftarrow z$ ; $z \leftarrow 0$ ; $LSB \leftarrow z[0]$
22. IF (CURVE $=$ $Montgomery$ AND $b \neq 1$)
        $(z=0)\ z \leftarrow 2 \times z + b$
        $t \leftarrow z$ ; $z \leftarrow 0$
        $z \leftarrow t \times t_1$
        $t_1 \leftarrow z$ ; $z \leftarrow 0$ ; $LSB \leftarrow z[0]$
23. $t_1 \leftarrow t_1 >> 1$
24. $(z=0)\ z \leftarrow 2 \times z + t_1$
25. $z \leftarrow 2 \times z + LSB - P_x$
26. $t_1 \leftarrow z$ ; $z \leftarrow 0$ ; $LSB \leftarrow z[0]$
27. $t_1 \leftarrow t_1 >> 1$
28. $(z=0)\ z \leftarrow 2 \times z + t_1$
29. $z \leftarrow 2 \times z + LSB - x_1$
30. $t_1 \leftarrow z$ ; $z \leftarrow 0$ ; $LSB \leftarrow z[0]$
31. IF (CURVE $=$ $Montgomery$)
        $t_1 \leftarrow t_1 >> 1$
        $(z=0)\ z \leftarrow 2 \times z + t_1$
        $z \leftarrow 2 \times z + LSB - a$
        $t_1 \leftarrow z$ ; $z \leftarrow 0$ ; $LSB \leftarrow z[0]$
32. $t_1 \leftarrow t_1 >> 1$
33. $(z=0)\ z \leftarrow 2 \times z + t_1$
34. $z \leftarrow 2 \times z + LSB - P_x$
35. $t \leftarrow z$ ; $z \leftarrow 0$
36. $(z=0)\ z \leftarrow 2 \times z + t_1$
37. $z \leftarrow 2 \times z + LSB + 0$
38. $x_1 \leftarrow z$ ; $z \leftarrow 0$
39. $z \leftarrow t \times t_2$
40. $t_1 \leftarrow z$ ; $z \leftarrow 0$ ; $LSB \leftarrow z[0]$
41. $t_1 \leftarrow t_1 >> 1$
42. $(z=0)\ z \leftarrow 2 \times z + t_1$
43. $z \leftarrow 2 \times z + LSB \pm P_y$
44. $t_1 \leftarrow z$ ; $z \leftarrow 0$
45. $(z=0)\ z \leftarrow 2 \times z - t_1$
46. $y_1 \leftarrow z$ ; $z \leftarrow 0$
47. $P_3 \leftarrow (x_1, y_1)$
```

Figure 3-17: Micro-code used for point addition $P_3 = P_1 \pm P_2(P_x, P_y)$ using the reconfigurable prime-field ECC module.

needs to be odd to have a valid ZSD form, that is, the least significant bit $k_0 = 1$. We prove that $\text{ZSD}^*(k) = (1, k_{t-1}, \cdots, k_2, k_1)$, where ZSD* is our special representation:

$$(1, k_{t-1}, \cdots, k_2, k_1) = 2^{t-1} + \underbrace{\frac{k-1}{2}}_{\text{+1 bits of } (k_{t-1}, \cdots, k_1)} - \underbrace{(2^{t-1} - 1 - \frac{k-1}{2})}_{\text{-1 bits of } (k_{t-1}, \cdots, k_1)} = k$$

Therefore, no additional circuitry is required to convert $k$ to ZSD* form when shifting its bits into the comb registers. Each comb is of length $d = \lceil t/4 \rceil$ bits ($t \leq 256 \Rightarrow d \leq 64$), and the combs are generated as:

```
$r \leftarrow 0$
for $i = 0$ to $t - 1$:
  for $j = 0$ to $d - 1$:
    shift $i^{th}$ bit of ZSD$^*(k)$ into COMB$_r$
  $r \leftarrow r + 1$
```

The ECSM control logic shifts all four COMB$_i$ registers, one bit at a time, and selects the appropriate pre-computed point ($P_x$, $P_y$) from the register file, based on the 4 bits obtained from the comb registers. Each iteration of the double-and-add process is then executed as $P_1 \leftarrow 2P_1$ followed by $P_1 \leftarrow P_1 + (P_x, P_y)$. ECDSA-Verify always uses two ECSMs, and doesn't involve any notion of side-channel security since all its inputs are public values. ECDH and ECDSA-Sign use one ECSM when SPA-secure, and two ECSMs when DPA-secure.

The reconfigurable ECC hardware was synthesized in the low-leakage TSMC 40nm LP process, at a clock frequency of $f_{CLK} = 10$ MHz. The total logic area occupied by the ECC core is $\approx$ 50,000 gate equivalents, including the 256-bit modular multiplier and inverter, several 256-bit registers ($p$, $n$, $x_1$, $y_1$, $t$ and $z$), 64-bit COMB$_i$ registers, and multiplexing and control logic. A 4KByte SRAM is used to store curve constants, scalars, ECDSA inputs and comb pre-computed points (for up to 6 sets of base points), and accounts for a large portion of the synthesized area. The area occupied by the SRAM is TSMC proprietary information and cannot be reported in this thesis.

Table 3.9: Performance of our prime-field ECC hardware at $V_{DD} = 1.0$ V.

| **Curve** | **Pre-Computation** | | **SPA-Secure ECSM** | | **DPA-Secure ECSM** | |
|---|---|---|---|---|---|---|
| **Size (Bits)** | **Cycles** | **Energy ($\mu$J)** | **Cycles** | **Energy ($\mu$J)** | **Cycles** | **Energy ($\mu$J)** |
| 160 | $136 \times 10^3$ | 2.32 | $74 \times 10^3$ | 1.26 | $151 \times 10^3$ | 2.57 |
| 192 | $185 \times 10^3$ | 3.33 | $102 \times 10^3$ | 1.84 | $208 \times 10^3$ | 3.75 |
| 224 | $246 \times 10^3$ | 4.67 | $137 \times 10^3$ | 2.60 | $280 \times 10^3$ | 5.32 |
| 256 | $317 \times 10^3$ | 6.34 | $179 \times 10^3$ | 3.58 | $363 \times 10^3$ | 7.26 |

Table 3.9 provides details of the number of cycles and energy consumed in comput-

ing ECSMs over elliptic curves of different sizes. Since our serial modular multiplier takes $t$ cycles, and width-4 comb-based ECSM involves $t/4$ "double-and-add" iterations, for a $t$-bit prime field, the execution time of ECSM is approximately $O(t^2)$, as evident from our results in Table 3.9. Our design also supports two side-channel secure modes – SPA-secure and DPA-secure, which employ techniques that have been proven to be side-channel secure in theory as well as in experimental results reported in literature. As conjectured earlier, energy consumption of DPA-secure ECSM is double that of SPA-secure ECSM. Table 3.10 compares our implementation with previous work. Although our implementation is extremely energy-efficient, its area is much higher than the "Tiny ECC" implementations in existing literature [68]. The total logic and memory area for previous work is reported in Table 3.10. Our hardware has a logic area of 50 KGates, excluding the SRAM, which is much larger than the other implementations. This is primarily because of the energy-efficient modular inverter reconfigurability features offered by our ECC core. Since we support any random prime, without any special structure, it is necessary to have only 256-bit data-paths

Table 3.10: Comparison with previous work on prime-field ECC designs.

| **Design** | **Tech** | **V$_{DD}$** | **Curve Size** | **Area** | **Cycles/ ECSM** | **Energy/ ECSM** | **SPA/ DPA Secure** |
|---|---|---|---|---|---|---|---|
| | **nm** | **V** | **Bits** | **KGate** | | **$\mu$J/Op** | |
| [45]* | 130 | 1.2 | 192 | 20 | $4.2 \times 10^6$ | - | - |
| [46]* | 350 | 3.3 | 256 | 31 | $1.1 \times 10^6$ | 550 | - |
| [47]* | 350 | 3.3 | 192 | 24 | $0.5 \times 10^6$ | 846 | Both |
| [48]* | 350 | 3.3 | 192 | 19 | $0.86 \times 10^6$ | 1294 | SPA |
| [49]* | 350 | 3.3 | 160 | 18 | $0.51 \times 10^6$ | 439 | Both |
| [50]* | 180 | 1.8 | 192 | 12 | $1.3 \times 10^6$ | 148 | SPA |
| [51]* | 32 | 0.9 | 192 | 26 | $0.35 \times 10^6$ | 3.15 | - |
| [52]* | 130 | 1.2 | 160 | 12 | $0.1 \times 10^6$ | 4.40 | Both |
| [53]* | 130 | 1.2 | 256 | 12 | $6.2 \times 10^6$ | - | - |
| **This work*** | 40 | 1.0 | 256 | 50+ † | $0.18 \times 10^6$ | 3.58 | SPA |
| **This work*** | 40 | 1.0 | 256 | 50+ † | $0.36 \times 10^6$ | 7.26 | Both |

*Synthesis results

†Logic area only, excluding 4 KB SRAM

for energy efficiency. This adds to the area occupied by modular arithmetic units. Also, we have used the Extended Euclidean algorithm-based modular inverter, which is much larger than the other implementations described in Table 3.10. Therefore, our ECC hardware is best suited for energy-constrained applications where logic area is not a big concern. One such application is the DTLS co-processor presented in Chapter 4. Other applications will be discussed briefly in Chapter 5.

# Chapter 4

# DTLS Co-Processor – Design and Simulation Results

As discussed in Chapter 2, DTLS is one of the most suited protocols for securing the IoT. Several protocol optimizations were proposed to make DTLS energy efficient for energy-constrained IoT applications. In Chapter 3, energy-efficient hardware implementations of AES, SHA and ECC were described, along with post-synthesis simulation results. In this chapter, we are going to discuss the implementation of a DTLS 1.3 co-processor, which combines the techniques discussed in Chapters 2 and 3 with dedicated hardware state machines to accelerate the complete DTLS 1.3 protocol.

## 4.1 Design of DTLS Hardware Accelerator

At the core of the (D)TLS protocol is its state machine, which controls all handshaking protocols and related computations. Since (D)TLS supports a large number of cipher suites and other configurations, this state machine is extremely complicated to implement in software. As discussed in [70] and [71], all successful attacks on TLS in the past have been due to implementation errors, weak ciphers or lack of client authentication. In this design, we make sure to enable only a carefully chosen secure subset of all the configuration options supported by DTLS. In our DTLS core, we support only the following 2 cipher suites:

- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256

- TLS_PSK_WITH_AES_128_GCM_SHA256

Since these cipher suites are at 128-bit security level, we support NIST P-256 and Curve25519 for ECDHE and NIST P-256 for ECDSA. Both server and client authentication are made mandatory when using the ECDHE_ECDSA cipher suite. The "Client Certificate URL" [62] extension is used, and client certificates are not transmitted. The "Cached Information" [63] extension is optional, and the server decides whether to use it or not. CA public keys are cached by both parties, and CA certificates are never exchanged. The handshake messages have been structured based on the requirements of these cipher suites, while still complying with the TLS 1.3 standard [8]. Considering BLE 4.2 to be the RF protocol, physical layer payloads must be smaller than 251 bytes. All handshakes messages, except the server certificate, exchanged by our DTLS core are small enough to not get fragmented. Messages belonging to the same flight are packed into the same record, whenever possible.

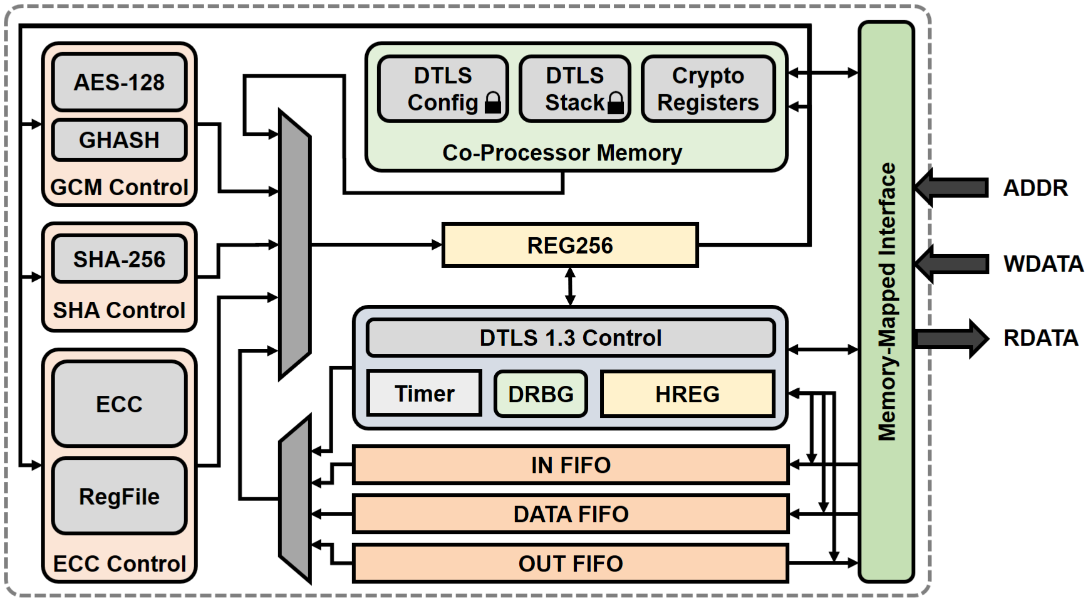


Figure 4-1: Architecture of the DTLS acceleration core.

Fig. 4-1 shows the overall architecture of our DTLS 1.3 hardware acceleration core.

The major components of the DTLS co-processor are:

- Cryptographic engines – AES, SHA, ECC
- DTLS 1.3 state machine
- Dedicated co-processor memory
- Packet FIFOs

The cryptographic engines are the energy-efficient implementations described in Chapter 3, and can also be accessed separately to accelerate standalone cryptographic computations. All data transfers between the cryptographic engines, the DTLS control and the co-processor memory occur through the 256-bit register REG256. This helps in minimizing multiplexing logic, thus saving area and power. An HMAC-based Deterministic Random Bit Generator (DRBG) [69] is used as a substitute for a True Random Number Generator (TRNG). The HMAC-DRBG is seeded with a 512-bit input, and it can be used to generate pseudo-random numbers with the same seed for up to $2^{48}$ invocations [69]. This DRBG is also used for deterministic ECDSA-Sign.

The DTLS core can be accessed through a memory-mapped interface, which exposes only a limited number of registers and memory locations to the external world. The co-processor memory is a 2 KByte SRAM ($64 \times 256$-bit words) which contains the "DTLS Config" configuration values, the "DTLS Stack" for handshake computations, and a set of "Crypto Registers" to allow standalone access to the cryptographic accelerators. The $15 \times 256$-bit "DTLS Config" is used to store important DTLS security parameters like the CA public key, server public key, client private key, CA and server identity details, pre-shared key and certificate hashes. These values are meant to be externally configured only when the device is first programmed, and are updated by the DTLS core automatically, as will be discussed in a later section. The $39 \times 256$-bit "DTLS Stack" is used to store temporary values used in handshake computations, DRBG states and DTLS session keys. To preserve security, these memory locations are not accessible through the memory-mapped interface. The "Crypto Registers" contain 10

$\times$ 256-bit dedicated memory locations for writing input operands and reading outputs from standalone cryptographic instructions.

The DTLS handshake requires 8 MD (including those required for ECDSA) and 19 HMAC computations using SHA-256, as discussed in Chapter 2, while our DTLS core has only one instance of the SHA-256 hardware. Therefore it is very important to efficiently schedule access to the SHA-256 engine. The handshake involves 6 "Session Hash" (also called "Transcript Hash") computations, that is, hash of the concatenation of all messages exchanged till that state. Software implementations of DTLS save all handshake messages, and compute the hash over all of them every time a "Session Hash" is required. As discussed in Chapter 2, the total handshake size can be more than 2000 Bytes. Therefore, storing them will require an additional 2 KByte SRAM, thus adding significantly to area and power overheads. The TLS specification [8] recommends using a "running hash" to avoid saving all messages, and we use the 64-Byte HREG FIFO to implement this concept.

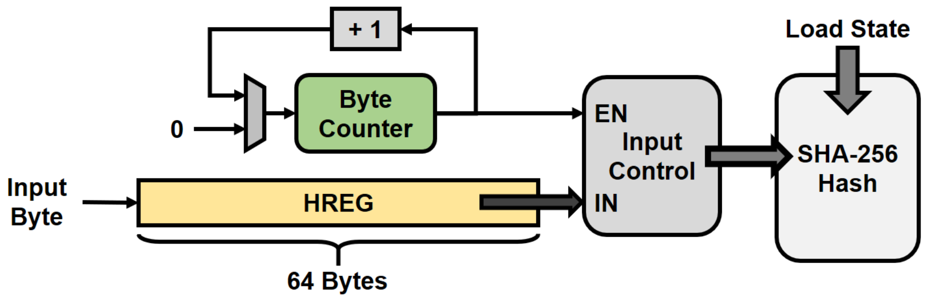


Figure 4-2: Efficient hardware implementation of "Session Hash" computations.

The SHA-256 input block size is 64 Bytes, and the SHA-256 internal registers $H_0 - H_7$ completely define the SHA-256 hash state every time a complete block has been digested. All handshake message bytes are pushed into HREG, and a byte counter is incremented. Whenever the counter reaches 64, that is, the FIFO is full, the block is sent to the SHA-256 engine. This ensures that "Session Hash" computations always digest data in blocks of 64 Bytes, except for the last block. After every "Session Hash", the state of HREG, the counter value and the hash state $H_0 - H_7$ are saved in the "DTLS Stack", so that the SHA-256 core can be used for other computations. For the next "Session Hash", the hash state is loaded back from the "DTLS Stack", and the above process is repeated. Since the $H_0 - H_7$ state is not accessible outside the DTLS core, there are no security concerns in implementing this scheme.

The DTLS core uses three FIFOs to fetch input messages (IN FIFO), send output messages (OUT FIFO) and read application data packets (DATA FIFO). Each FIFO is $256 \times 1$ Byte, implemented using 256 Byte SRAMs. The IN FIFO ensures that the DTLS core starts parsing input messages only when a fully formed packet is available, and sends out full output messages in to the OUT FIFO. The DTLS core also verifies that the data in DATA FIFO does not exceed BLE 4.2 PDU limitations. For encrypted application data, our DTLS core implements the packet optimizations proposed in Chapter 2, with the option to enable AES-GCM tag truncation.

As discussed earlier, UDP channels are prone to packet loss and packet re-ordering. The DTLS handshake has countermeasures to deal with both situations. For packet re-ordering, DTLS recommends optionally caching fragmented messages (which will require additional storage) based on their "Message Sequence" and "Fragment Offset" values. The handshake packets in our DTLS implementation can easily fit in the BLE 4.2 PDU, except for the *ServerCertificate*, thus making the probability of re-ordering negligible. Therefore, we do not cache fragmented messages, and recommend that the DTLS 1.3 server transmits the fragments of its certificate at time intervals sufficient to prevent re-ordering. For packet loss, DTLS requires the client to re-transmit the previous flight of messages if it does not receive the next flight from the server [6]. Our DTLS core uses a 64-bit counter `retx_timer` to implement the "DTLS Retransmission Timer". The time-out value can be configured externally, and the state machine retransmits the previous flight (*ClientHello*, *ClientHello* + *Cookie* or *ClientCertificateURL** + *ClientCertificateVerify** + *ClientFinished*) whenever the timer expires. When the DTLS state machine is waiting for the next flight from the server, and the timer is running, all other modules are clock-gated in order to reduce dynamic power consumption. The DTLS control logic retransmits a flight up to a fixed number of times, determined by the `retry_limit` register, which can be configured externally.

A 48-bit `seq_num` register is used to keep track of the sequence number of transmitted packets. The DTLS core stops accepting application data packets whenever the sequence number reaches its maximum value – $2^{15} - 1$ or $2^{48} - 1$ depending on whether truncated GCM tags are being used or not, respectively. To prevent replay attacks, a 48-bit `next_recv_seq_num` register is used to store the next expected sequence number of received packets. If the received packet has a sequence number smaller than `next_recv_seq_num`, it is discarded. As recommended in [6], packets with invalid GCM tags are discarded silently in order to prevent "error type oracle" attacks. For valid packets, the `next_recv_seq_num` register is updated appropriately.

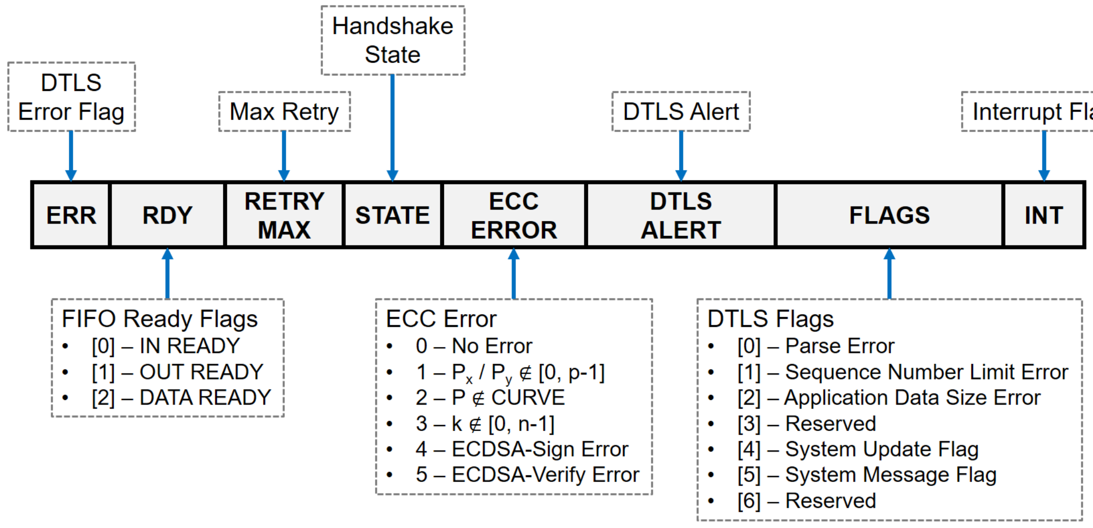


Figure 4-3: DTLS status register.

While most of the DTLS control logic is hidden from the external world, it is important to be able to access some state information for debug and software control. The memory-mapped 32-bit DTLS status register (Fig. 4-3) is used for this purpose. The "Error Flag" is set whenever an error occurs. The "FIFO Ready Flags" are set whenever the corresponding packet FIFO has a full packet. "Retry Max" is the maximum number of retries in a single session, and this value can be used to dynamically configure the retry limit. "State" indicates the phase of the handshake that the DTLS core is in. "ECC Error", "DTLS Alert" and "Flags" provide finer details about error conditions for easier debug. The "System Update" and "System Message" flags are related to the automatic security update feature of our DTLS core, which will be discussed in a later section. The interrupt flag "INT" is raised whenever the core has completed an instruction.

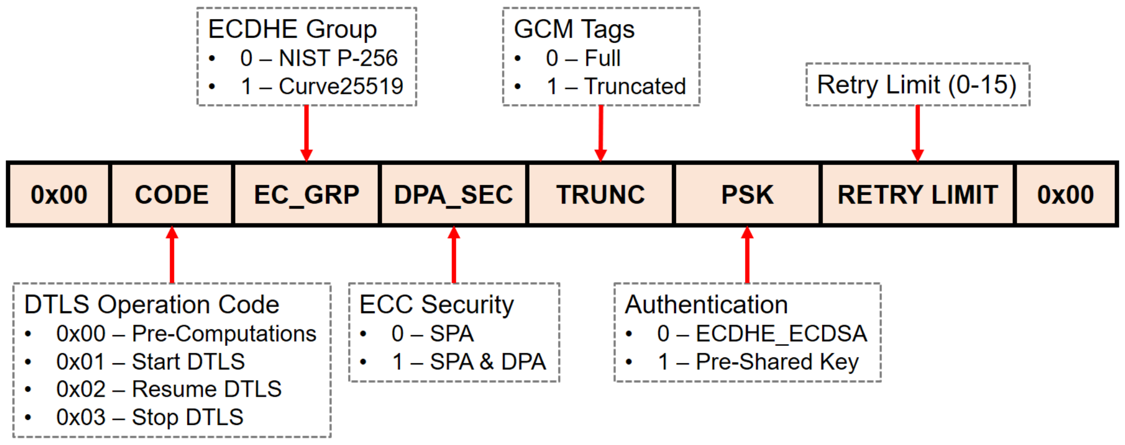


Figure 4-4: DTLS instruction.

Fig. 4-4 shows the 32-bit instruction used to perform DTLS operations. The operations are divided into four phases – ECC pre-computations, start connection, resume handshake or application data, and end connection. Various flags can be set in the instruction to indicate the elliptic curve to be used for ECDHE, ECC security level (SPA or DPA), GCM tag truncation and whether to use the non-forward-secret PSK cipher suite. The retry limit (between 0 and 15) can also be specified through this instruction. The pre-computations are meant to be performed once, and the points stored in a secure non-volatile memory. Before issuing the "Start DTLS" instruction, it is necessary to configure the memory-mapped "Date" (6 Bytes) and "Timeout" (8 Bytes) registers with the current date in YYMMDD format (for certificate validation) and the handshake retransmit timeout value respectively. When the "Stop DTLS" instruction is executed, the DTLS state machine generates an encrypted "Close Notify" alert in the OUT FIFO to ensure that the connection is terminated correctly.

The DTLS handshake may involve up to 7 ECSM computations, and we have seen in Chapter 3 that ECSM energy can be reduced by 65% if pre-computed comb points are available. Our energy-efficient ECC module supports storing up to 6 pre-computed base points, and we use this feature to minimize the energy consumption of the ECC operations which account for majority of the ECDHE_ECDSA handshake. In the "DTLS Pre-Computation" phase, comb points are generated for:

- NIST P-256 generator point $G_1$
- Curve25519 generator point $G_2$
- CA public key $Q_{CA}$
- Server public key $Q_{SRV}$

The pre-computations for $G_1$ and $G_2$ are useful in the first half of ECDHE, while no pre-computations can be done for the second half (since the server's key share is a random point on the curve). Pre-computation for $Q_{CA}$ helps with verifying the CA signature in *ServerCertificate*, while that for $Q_{SRV}$ helps with verifying the server signature in *ServerCertificateVerify* when the "Cached Information" extension is used. Therefore, our ECC pre-computation scheme reduces the total handshake ECC energy by 58% when the "Cached Information" extension is used, and 51% otherwise, at the cost of additional storage requirements. The two additional sets of available point storage can be used to perform ECDH and ECDSA with random points without corrupting the points required by DTLS. The pre-computation memory is internal to the ECC core, and not accessible through the memory-mapped interface.

## 4.2 Over-the-Air Security Update

Security updates are a major concern for IoT systems. Traditional solutions involve sending a signed security "patch" to the device, and having the on-device software verify the signature and update its security parameters. This method is prone to security issues because of software implementation bugs and the possibility of malicious software setting attacker-chosen security parameters on the device. Our DTLS hardware can seamlessly handle security updates through its secure system update state machine (Fig. 4-5).

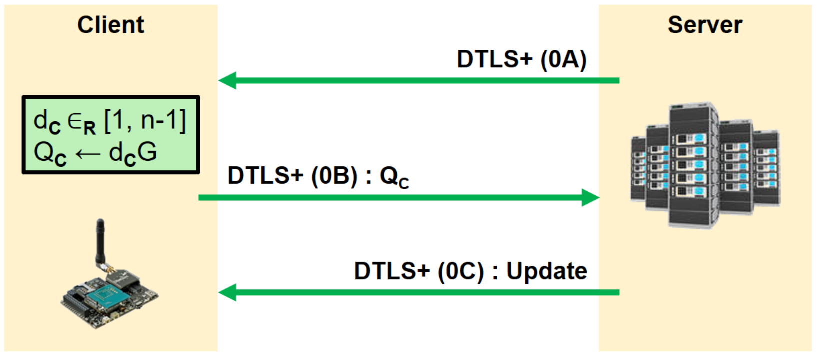


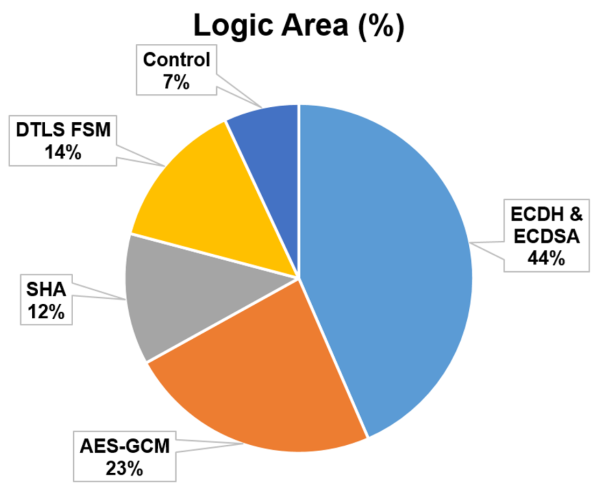
Figure 4-5: Over-the-air update of security parameters.

Once the handshake has been performed and a secure encrypted channel has been established, the server can initiate a security update by sending an encrypted system update message, which starts with the string "DTLS+", followed by the byte 0x0A. The DTLS core in the client device then generates a new ECDSA private key $d_C \in_R [1, n-1]$, caches it in the "DTLS Config" section of its memory, computes the corresponding public key $Q_C = d_C G$ and sends it in an encrypted packet to the server. The two parties continue exchanging application data as usual, while the server prepares the security update. Finally, the server sends another encrypted message containing the new CA public key, client certificate hash and URL, and certificate identity parameters. The DTLS core then updates its "DTLS Config", performs pre-computations for the newly obtained ECDSA public keys, and sets the "System Update" flag high in its status register to indicate that the connection must be restarted to begin using the updated security configurations. Both parties also update their pre-shared key (PSK) values by feeding them through a known DRBG. Since all these computations occur inside the DTLS core, it is free from software-based attacks. Also, no secret key values are transmitted through the encrypted channel, thus preserving forward secrecy. It is recommended to appropriately pad all server control messages to the full PDU limit to make them indistinguishable from each other based on packet length, thus reducing the chances of DoS attacks on the system update messages. This scheme only deals with DTLS-related updates, and device firmware updates are not currently handled. However, an IoT device may integrate the

DTLS core with a micro-processor in privileged mode to implement secure firmware updates.

## 4.3 Synthesis and Simulation Results

The DTLS 1.3 co-processor was synthesized at $f_{CLK} = 10$ MHz in the TSMC 40nm LP low-leakage process. The DTLS core has a total logic area of around 115 KGates and Fig. 4-6 shows the area occupied by the individual components. The DTLS core uses different SRAMs with a total size of 6.75 KBytes.



| Component | Logic Area (KGates) |
|---|---|
| AES-GCM | 27 |
| SHA | 14 |
| ECC | 50 |
| DTLS FSM | 16 |
| Control | 8 |
| **Total Logic** | 115 |

| SRAM | Size |
|---|---|
| ECC Comb | 4 KB |
| DTLS | 2 KB |
| Packet FIFO | 3 x 256 Bytes |
| **Total Memory** | 6.75 KB |

Figure 4-6: Area breakdown of DTLS core.

Table 4.1: Comparison of our energy-efficient reconfigurable DTLS core with integrated cryptographic accelerators.

| **Design** | **Tech** | **Area** | **AES** | **SHA** | **ECDH** | **ECDSA** | **DTLS** |
|---|---|---|---|---|---|---|---|
| | **nm** | **KGate** | | | | | |
| [56]* [1] | 350 | 22 | Y | Y | - | Y | - |
| [57]* [2] | 130 | 15 | Y | - | Y | - | - |
| **This work*** | 40 | 115 † | Y | Y | Y | Y | Y |

*Synthesis results

[1]Supports only NIST P-192 ECDSA, AES-128-ECB and SHA-1

[2]Supports only Curve25519 ECDH and Salsa20-Poly1305

†Logic area only, excluding SRAMs

Table 4.2: Energy consumption of DTLS handshake implementations.

| **Implementation** | **Crypto** | **Control** | **Handshake Energy ($\mu$J)** |
|---|---|---|---|
| Software Only | S/W | S/W | 90,000 |
| Software + Hardware | H/W | S/W | 5,000 |
| Hardware Only | H/W | H/W | 70 |

Table 4.1 compares our DTLS core with other integrated cryptographic accelerators in literature. Our design supports the largest number of operations, with the maximum reconfigurability in ECC. [56] and [57] report only the mean power values for their designs, hence it is very difficult to compare with our implementation since power for such integrated accelerators will be a strong function of the operation being performed.

For the ECDHE-ECDSA-based cipher suite, our DTLS core consumes $\approx 45$ $\mu$J to perform a handshake when the "Cached Information" extension is used, and $\approx 70$ $\mu$J otherwise. For the PSK-based cipher suite, the DTLS core consumes only $\approx 10\mu$J for the handshake. These numbers also include the energy consumption in SRAM accesses during handshake computations.

For a pure software-based implementation of the DTLS handshake, as much as 4-5 mJ of energy is used in executing the state machine, delegating data between cryptographic functions and most importantly, reading and writing data from the main memory. Our DTLS core reduces this energy by around 100 times by restricting memory accesses only to the dedicated "DTLS Stack", and scheduling SRAM accesses to minimize energy consumption. Table 4.2 compares the energy consumption of different implementations of the ECDHE-ECDSA-based DTLS handshake, using software only, using hardware and software, and using hardware only. Energy consumption of software-based DTLS control is based on analysis of the DTLS 1.2 handshake using the mbedTLS stack [59] implemented on ARM Cortex M0+. Clearly, the full hardware-based implementation provides maximum energy efficiency.

In Chapter 2, we had discussed that our proposed protocol optimizations lead to 45% reduction in total DTLS energy for IoT devices that authenticate once every week and transmit 32 Bytes of data every hour. We had also claimed that energy-efficient cryptographic hardware can provide further energy benefits for such frequently

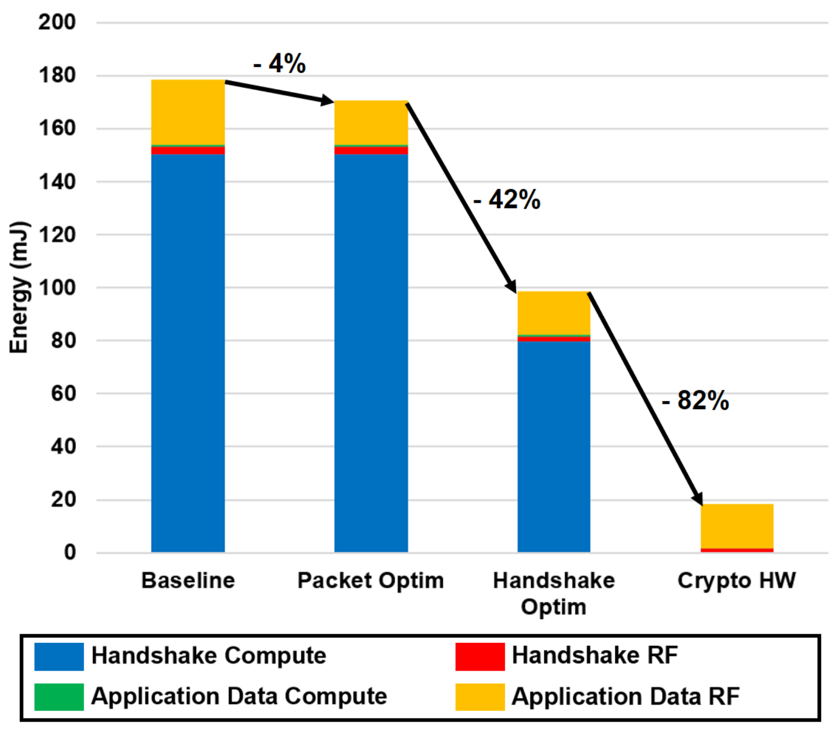


Figure 4-7: Reduction in energy consumption of frequently authenticating DTLS clients using energy-efficient DTLS hardware.

authenticating devices. Fig. 4-7 validates our claim, with further 82% reduction in total DTLS energy when using our DTLS core.

## 4.4 Integration with RISC-V

RISC-V is an open instruction set architecture based on the reduced instruction set computing (RISC) principles [72]. It is designed to be useful in computing applications ranging from huge cloud infrastructures to the smallest embedded systems. Our DTLS core was integrated with a 32-bit 3-stage pipeline RISC-V micro-processor, based on the "RISCY" family of open-source processors published by the Computation Structures Group from MIT CSAIL [73].

The processor can access the DTLS core through a memory-mapped interface, as shown in Fig. 4-8. The DTLS instruction register, status register, configuration registers, and cryptographic acceleration registers are assigned fixed 32-bit addresses on this memory bus. There is also an interrupt line from the DTLS core that connects to the interrupt controller inside the RISC-V core. Software can access the memory-mapped registers for read and write operations, and monitor the interrupt line to check if an instruction has completed execution.

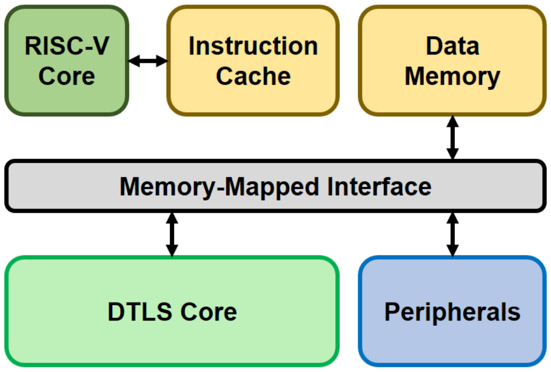


Figure 4-8: Block diagram of DTLS core integrated with RISC-V.

Apart from the DTLS instructions, the following cryptographic functions can be executed on the DTLS core:

- AES-128-ECB-Encrypt
- AES-128-GCM-Auth-Encrypt/Decrypt
- SHA-256 Message Digest
- SHA-256 HMAC
- SHA-256 HMAC-DRBG
- $\mathbb{F}_p$ Modular Add / Subtract / Multiply / Divide / Invert
- $E(\mathbb{F}_p)$ ECSM / ECDSA-Sign / ECDSA-Verify over Short Weierstrass and Montgomery curves

These standalone instructions can be used to implement custom protocols and novel cryptographic primitives in conjunction with software.

The RISC-V processor issues an instruction to the core and waits for an interrupt (WFI). While waiting for the interrupt, most of the processor logic is clock-gated to save power. Similarly, the DTLS core can be clock gated, by writing to a memory-mapped register, to save power when it is not being used. The DTLS core also has a clock divider which can be configured, again by writing to a memory-mapped register, to run at the system clock or logarithmically divided clocks. Appendix A provides some details about programming the DTLS co-processor using the memory-mapped register space.

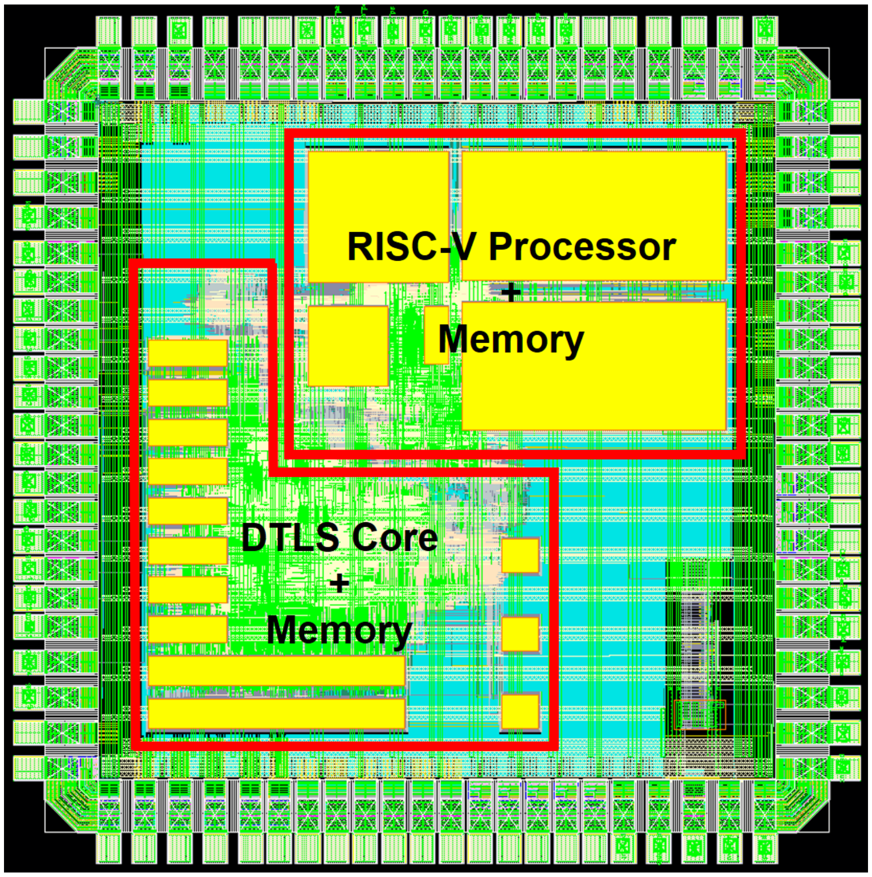


Figure 4-9: Chip layout of RISC-V processor with DTLS security core.

The chip, with the RISC-V processor, data memory, instruction cache, the DTLS core and peripherals, was taped out in the TSMC 65nm LP process. Fig. 4-9 shows the layout of the chip.

# Chapter 5

# Conclusion

This research work presents several hardware and protocol architectures that can make security affordable for low-power wireless sensor networks. The proposed techniques can be used to secure the Internet of Things, while minimizing energy consumption on the sensor node devices. This chapter summarizes the contributions of this thesis, along with comparison with the state-of-the-art, and future work.

## 5.1 Thesis Contributions

This thesis presents energy-efficient protocols and cryptographic hardware architectures for implementing Datagram Transport Layer Security (DTLS) in energy-constrained IoT devices.

- Protocol optimizations are proposed that can reduce the overall energy costs of DTLS. Packet optimizations reduce communication overheads by 91%, and handshake optimizations are used to reduce the number of expensive ECDSA-Verify operations. Overall, the optimized protocol provides as much as 45% savings in the total energy consumption, both RF and computation, of a typical duty-cycled wireless sensor node.

- Energy-efficient architectures for the standard cryptographic primitives AES, SHA and ECC are discussed, along with post-synthesis results and comparison with previous work. For all three primitives, our proposed designs provide around 2,000-2,500 times reduction in energy consumption when compared with software implementations on embedded micro-processors.

- A dedicated co-processor for the DTLS 1.3 protocol is also presented, which can accelerate the complete DTLS state machine in hardware. This dedicated DTLS core reduces energy consumption involved in memory accesses and delegation of data between cryptographic accelerators, and reduces the total computation energy of the DTLS handshake by several orders of magnitude. It also uses clever pre-computations to reduce ECC computations, and hash architectures to allow computing running hashes for the DTLS handshake.

- The DTLS core is integrated with a RISC-V processor through a memory-mapped interface to allow accelerating a wide range of cryptographic functions. This provides a flexible platform for implementing and evaluating custom security protocols and cryptographic algorithms.

## 5.2 Comparison with State-of-the-Art

As discussed in Chapter 1, there is substantial amount of literature in the field of cryptographic hardware design, and some existing work on security protocol optimizations. There is no prior work on a completely hardware-based DTLS state machine, which is first presented in this thesis. Also, existing literature on cryptographic hardware mostly focus on low area implementations, while our proposed techniques reduce energy consumption, often at the cost of area. Comparison of our hardware implementations with previous work, based on power, performance, area and other metrics, has been presented in detail in Chapter 3. Table 5.1 summarizes the comparison with the state-of-the-art.

Table 5.1: Comparison with the state-of-the-art.

| **References** | **Key Ideas Published** | **Our Contributions** |
|---|---|---|
| AES | | |
| [34], [35], [37], [38] | Low area and low power using highly serialized designs and optimized S-Box | High energy efficiency using parallel data-paths |
| [36] | Energy efficiency through pipelining | Energy efficiency through parallelism |
| GCM | | |
| [39], [40] | High throughput architectures with large area | Energy efficient architectures with small area |
| SHA | | |
| [41], [42], [43], [44] | High throughput implementations | Low energy using round data-path optimizations |
| ECC | | |
| [45], [46] | Low-area dual-field modular arithmetic for ECC | Energy-efficient prime-field ECC |
| [47], [48], [49], [50], [52], [56] | Low area and low power ECDSA for RFID | Energy efficient ECDSA using optimized hardware state machines |
| [51], [53], [57] | Low area prime-field modular arithmetic for ECDH | Low energy hardware state machine for ECDH |
| Integrated Accelerators | | |
| [56], [57] | Low area integration of AES and SHA with ECDH or ECDSA | Highly reconfigurable co-processor that integrates AES, SHA and ECC along with dedicated DTLS 1.3 state machine |

## 5.3 Future Work

The test chip, with the DTLS core and RISC-V processor, will be used to evaluate DTLS and related cryptographic computations. Energy consumption of software-based implementations as well as the hardware-based primitives will be measured. This will be especially useful because both the micro-processor and the cryptographic hardware have been designed in the same technology node. The ECC implementations will be tested for side-channel security, by recording multiple power traces and running standard side-channel attack algorithms. Finally, a test system for DTLS connected sensor nodes will be prepared to demonstrate the energy efficiency of the proposed techniques.

Through a combination of hardware and software, the chip can be used to perform several other investigations. Some directions of future research are mentioned below:

- Compare the energy consumption of memory accesses with AES or SHA operations to determine the limits of hardware optimizations.
- Implement elliptic-curve based protocols like bilinear pairings.
- Implement quantum-secure cryptographic schemes like Ring-Learning with Errors using the modular arithmetic instructions.
- Profile software implementations of custom security protocols that utilize the on-chip cryptographic accelerators.

# Appendix A

# Programming the DTLS Co-Processor

The 32-bit instructions used by the DTLS co-processor contain 8 bits of opcode in their most significant byte. Table A.1 provides a list of opcodes for the different instructions supported.

Table A.1: Instruction opcodes.

| **Opcode** | **Instruction** |
|---|---|
| 00 | No operation |
| 01 | DTLS |
| 02 | $\mathbb{F}_p$ Arithmetic |
| 03 | $\mathbb{F}_p$ ECC |
| 04 | AES-ECB |
| 05 | AES-GCM |
| 06 | SHA Message Digest |
| 07 | SHA HMAC |
| 08 | SHA HMAC-DRBG |
| FF | Clear Interrupt |

As discussed in Chapter 5, the DTLS co-processor uses a memory-mapped interface for all communications with the master processor. Table A.2 provides a list of these memory-mapped locations, along with their sizes and functions. The registers can be accessed through software using the following C construct:

```
volatile unsigned long * const <variable_name> =
        (unsigned long *) <memory_adress>;
```

Table A.2: Memory-mapped registers of DTLS co-processor.

| **Address** | **Size (Bits)** | **Function** |
|---|---|---|
| AAAA01C0 | 32 | Instruction Register |
| AAAA01C4 | 32 | Status Register |
| AAAA01C8 | 32 | DTLS Packet Input FIFO |
| AAAA01CC | 32 | DTLS Packet Output FIFO |
| AAAA01D0 | 32 | Application Data FIFO |
| AAAA01E0 | 256 | ECC Prime $p$ |
| AAAA0200 | 256 | ECC Order $n$ |
| AAAA0220 | 256 | ECC Curve Parameter $a$ / AES Key and IV |
| AAAA0240 | 256 | ECC Curve Parameter $b$ / AES AAD / Input Text / Input Tag |
| AAAA0260 | 256 | ECC Secret Scalar $k$ / ECC Input $P_x$ |
| AAAA0280 | 256 | ECDSA Message Hash $H_m$ / ECC Input $P_y$ |
| AAAA02A0 | 256 | ECDSA Input $r$ / HMAC Key Upper |
| AAAA02C0 | 256 | ECDSA Input $s$ / HMAC Key Lower |
| AAAA02E0 | 256 | ECC Output $Q_x$ or $r$ / AES Output Text / Output Tag |
| AAAA0300 | 256 | ECC Output $Q_y$ or $s$ / SHA Output |
| AAAAA000 | 32 | Clock Divider Config |
| AAAAA004 | 32 | Clock Gate Config |

All the instructions need to write the input operands to the specified memory locations, issue the instruction, wait for interrupt, and then read the output from the corresponding memory address after an interrupt is received from the co-processor.

# Bibliography

[1] D. Evans, "The Internet of Things - How the Next Evolution of the Internet Is Changing Everything," *CISCO White Paper*, April 2011.

[2] Symantec Corporation, "Internet Security Threat Report," vol. 21, April 2016.

[3] E. Ronen, C. O'Flynn, A. Shamir and A. Weingarten, "IoT Goes Nuclear: Creating a ZigBee Chain Reaction," *Cryptology ePrint Archive*, Report 2016/1047, November 2016.

[4] S. Gollakota, H. Hassanieh, B. Ransford, D. Katabi and K. Fu, "They Can Hear Your Heartbeats: Non-Invasive Security for Implantable Medical Devices," in *Proceedings of the ACM SIGCOMM*, August 2011.

[5] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," *IETF RFC*, vol. 5246, August 2008.

[6] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," *IETF RFC*, vol. 6347, January 2012.

[7] S. L. Keoh, S. S. Kumar and H. Tschofenig, "Securing the Internet of Things: A Standardization Perspective," in *IEEE Internet of Things Journal*, vol. 1, no. 3, pp. 265-275, June 2014.

[8] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," *IETF Internet-Draft*, March 2017. [Online]. Available: https://tlswg.github.io/tls13-spec/

[9] E. Rescorla and H. Tschofenig, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3," *IETF Internet-Draft*, October 2016. [Online]. Available: https://tools.ietf.org/html/draft-rescorla-tls-dtls13-00

[10] A. Menezes, P. van Oorschot and S. Vanstone, "Handbook of Applied Cryptography," CRC Press, 1996.

[11] C. Paar and J. Pelzl, "Understanding Cryptography – A Textbook for Students and Practitioners," Springer-Verlag, 2010.

[12] NIST, "Advanced Encryption Standard (AES)," *NIST Technical Report*, FIPS PUB 197, November 2001.

[13] NIST, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," *NIST Special Publication*, vol. 800-38D, November 2007.

[14] A. Bogdanov, D. Khovratovich and C. Rechberger, "Biclique Cryptanalysis of the Full AES," in *Advances in Cryptology Ũ ASIACRYPT 2011*, *Lecture Notes in Computer Science*, vol. 7073, pp. 344-371, December 2011.

[15] NIST, "Secure Hash Standard (SHS)," *NIST Technical Report*, FIPS PUB 180-4, March 2012.

[16] NIST, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," *NIST Technical Report*, FIPS PUB 202, August 2015.

[17] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, A. Petit Bianco and C. Baisse, "Announcing the First SHA1 Collision," *Google Security Blog*, February 2017.

[18] D. Khovratovich, C. Rechberger and A. Savelieva, "Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family," in *Fast Software Encryption*, *Lecture Notes in Computer Science*, vol. 7549, pp. 244-263, 2012.

[19] M. Lamberger and F. Mendel, "Higher-Order Differential Attack on Reduced SHA-256," emphCryptology ePrint Archive, Report 2011/037, January 2011.

[20] NIST, "The Keyed-Hash Message Authentication Code (HMAC)," *NIST Technical Report*, FIPS PUB 198-1, July 2008.

[21] N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203Ũ209, January 1987.

[22] V. Miller, "Use of Elliptic Curves in Cryptography," in *Proc. Advances in Cryptology – CRYPTO 1985*, *Lecture Notes in Computer Science*, vol. 218, pp. 417Ũ426, August 1985.

[23] D. Hankerson, A. Menezes, and S. Vanstone, "Guide to Elliptic Curve Cryptography," Springer-Verlag, 2004.

[24] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records" in *Public Key Cryptography – PKC 2006*, *Lecture Notes in Computer Science*, vol. 3958, pp. 207-228, April 2006.

[25] R. C. Merkle, "Secure Communications over Insecure Channels." *Communications of the ACM*, vol. 21, no. 4, pp. 294-299, April 1978.

[26] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644-654, November 1976.

[27] NIST, "Digital Signature Standard (DSS)," *NIST Technical Report*, FIPS PUB 186-4, July 2013.

[28] IEEE Computer Society, "IEEE 802.15.4 – Low-Rate Wireless Personal Area Networks (LR-WPANs)." [Online]. Available: http://standards.ieee.org/about/get/802/802.15.html

[29] Bluetooth SIG, "Bluetooth Specification 4.2." [Online]. Available: https://www.bluetooth.com/specifications/bluetooth-core-specification

[30] J. Hui and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks," *IETF RFC*, vol. 6282, September 2011.

[31] J. Nieminen, T. Savolainen, M. Isomaki, B. Patil, Z. Shelby and C. Gomez, "IPv6 over BLUETOOTH (R) Low Energy," *IETF RFC*, vol. 7668, October 2015.

[32] S. Raza, D. Trabalza and T. Voigt, "6LoWPAN Compressed DTLS for CoAP," *IEEE International Conference on Distributed Computing in Sensor Systems*, pp. 287-289, May 2012.

[33] A. Capossele, V. Cervo, G. De Cicco and C. Petrioli, "Security as a CoAP Resource: An Optimized DTLS Implementation for the IoT," *2015 IEEE International Conference on Communications (ICC)*, pp. 549-554, September 2015.

[34] A. Satoh, S. Morioka, K. Takano and S. Munetoh, "A Compact Rijndael Hardware Architecture with S-Box Optimization," in *Proc. Advances in Cryptology – ASIACRYPT 2001*, *Lecture Notes in Computer Science*, vol. 2248. pp. 239-254, November 2001.

[35] P. Hamalainen, T. Alho, M. Hannikainen and T. D. Hamalainen, "Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core," *EUROMICRO Conference on Digital System Design (DSD '06)*, pp. 577-583, September 2006.

[36] S. K. Mathew, F. Sheikh, M. Kounavis, S. Gueron, A. Agarwal, S. K. Hsu, H. Kaul, M. A. Anders and R. K. Krishnamurthy, "53 Gbps Native $GF(2^4)^2$ Composite-Field AES-Encrypt/Decrypt Accelerator for Content-Protection in 45 nm High-Performance Microprocessors," in *IEEE Journal of Solid-State Circuits*, vol. 46, no. 4, pp. 767-776, April 2011.

[37] S. Mathew, S. Satpathy, V. Suresh, M. Anders, H. Kaul, A. Agarwal, S. Hsu, G. Chen and R. Krishnamurthy, "340 mVŰ1.1 V, 289 Gbps/W, 2090-Gate NanoAES Hardware Accelerator With Area-Optimized Encrypt/Decrypt $GF(2^4)^2$ Polynomials in 22 nm Tri-Gate CMOS," in *IEEE Journal of Solid-State Circuits*, vol. 50, no. 4, pp. 1048-1058, April 2015.

[38] Y. Zhang, K. Yang, M. Saligane, D. Blaauw and D. Sylvester, "A Compact 446 Gbps/W AES Accelerator for Mobile SoC and IoT in 40nm," *2016 IEEE Symposium on VLSI Circuits*, pp. 1-2, June 2016.

[39] A. Satoh, "High-Speed Parallel Hardware Architecture for Galois Counter Mode," *2007 IEEE International Symposium on Circuits and Systems*, pp. 1863-1866, May 2007.

[40] M. Mozaffari-Kermani and A. Reyhani-Masoleh, "Efficient and High-Performance Parallel Hardware Architectures for the AES-GCM," in *IEEE Transactions on Computers*, vol. 61, no. 8, pp. 1165-1178, August 2012.

[41] L. Dadda, M. Macchetti and J. Owen, "The Design of a High Speed ASIC Unit for the Hash Function SHA-256 (384, 512)," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, vol. 3, pp. 70-75, February 2004.

[42] A. Satoh and T. Inoue, "ASIC Hardware Focused Comparison for Hash Functions MD5, RIPEMD-160, and SHS," *International Conference on Information Technology: Coding and Computing (ITCC '05)*, pp. 532-537, April 2005.

[43] Y. K. Lee, H. Chan, I. Verbauwhede, "Iteration Bound Analysis and Throughput Optimum Architecture of SHA-256 (384,512) for Hardware Implementations," in *Information Security Applications – WISA 2007*, *Lecture Notes in Computer Science*, vol. 4867, pp. 102-114, August 2007.

[44] R. Ramanarayanan, S. Mathew, F. Sheikh, S. Srinivasan, A. Agarwal, S. Hsu, H. Kaul, M. Anders, V. Erraguntla and R. Krishnamurthy, "18Gbps, 50mW Reconfigurable Multi-Mode SHA Hashing Accelerator in 45nm CMOS," in Proc. European Solid-State Circuits Conference – ESSCIRC 2010, pp. 210-213, September 2010.

[45] A. Satoh and K. Takano, "A Scalable Dual-Field Elliptic Curve Cryptographic Processor,"" in *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 449-460, April 2003.

[46] J. Wolkerstorfer, "Scaling ECC Hardware to a Minimum," *Cryptographic Advances in Secure Hardware – CRASH 2005*, September 2005.

[47] F. Furbass and J. Wolkerstorfer, "ECC Processor with Low Die Size for RFID Applications," *2007 IEEE International Symposium on Circuits and Systems*, pp. 1835-1838, May 2007.

[48] M. Hutter, M. Feldhofer and T. Plos, "An ECDSA Processor for RFID Authentication," in *Radio Frequency Identification: Security and Privacy Issues – RFIDSec 2010*, *Lecture Notes in Computer Science*, vol. 6370, pp. 189-202, June 2010.

[49] E. Wenger, M. Feldhofer and N. Felber, "Low-Resource Hardware Design of an Elliptic Curve Processor for Contactless Devices," in *Information Security Applications – WISA 2010*, *Lecture Notes in Computer Science*, vol. 6513, pp. 92-106, August 2010.

[50] T. Kern and M. Feldhofer, "Low-Resource ECDSA Implementation for Passive RFID Tags," *2010 IEEE International Conference on Electronics, Circuits and Systems*, pp. 1236-1239, December 2010.

[51] S. S. Roy, B. Yang, V. Rozic, N. Mentens, J. Fan and I. Verbauwhede, "Designing Tiny ECCProcessor," *Workshop on Elliptic Curve Cryptography – ECC 2013*, September 2013.

[52] P. Pessl and M. Hutter, "Curved Tags – A Low-Resource ECDSA Implementation Tailored for RFID," in *Radio Frequency Identification: Security and Privacy Issues – RFIDSec 2014*, *Lecture Notes in Computer Science*, vol. 8651, pp. 156-172, July 2014.

[53] J. Bosmans, S. S. Roy, K. Jarvinen and I. Verbauwhede, "A Tiny Coprocessor for Elliptic Curve Cryptography over the 256-bit NIST Prime Field," *International Conference on VLSI Design – VLSID 2016*, pp. 523-528, January 2016.

[54] D. Carlson, D. Brasili, A. Hughes, A. Jain, T. Kiszely, P. Kodandapani, A. Vardharajan, T. Xanthopoulos and V. Yalal, "A High Performance SSL IPSEC Protocol Aware Security Processor," *2003 IEEE International Solid-State Circuits Conference*, *Digest of Technical Papers – ISSCC 2003*, vol. 1, pp. 142-483, February 2003.

[55] J. Goodman and A. P. Chandrakasan, "An Energy-Efficient Reconfigurable Public-Key Cryptography Processor," in *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1808-1820, November 2001.

[56] M. Hutter, M. Feldhofer and J. Wolkerstorfer, "A Cryptographic Processor for Low-Resource Devices: Canning ECDSA and AES Like Sardines," in *Information Security Theory and Practice: Security and Privacy of Mobile Devices in Wireless Communication – WISTP 2011*, *Lecture Notes in Computer Science*, vol. 6633, pp. 144-159, June 2011.

[57] M. Hutter, J. Schilling, P. Schwabe and W. Wieser, "NaCl's `crypto_box` in Hardware," in *Cryptographic Hardware and Embedded Systems – CHES 2015*, *Lecture Notes in Computer Science*, vol. 9293, pp. 81-101, September 2015.

[58] NXP Semiconductors, "Kinetis KL25 Sub-Family: 48 MHz Cortex-M0+ Based Microcontroller with USB," *Data Sheet*, Rev. 5, August 2014. [Online]. Available: http://www.nxp.com/assets/documents/data/en/data-sheets/KL25P80M48SF0.pdf

[59] ARM Holdings, *ARM mbedTLS*. [Online]. Available: https://tls.mbed.org

[60] M. Siekkinen, M. Hiienkari, J. K. Nurminen and J. Nieminen, "How Low Energy is Bluetooth Low Energy? Comparative Measurements with ZigBee/802.15.4," *2012 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pp. 232-237, April 2012.

[61] Texas Instruments Inc., “CC2540: 2.4GHz Bluetooth Low Energy System-on-Chip,” *Data Sheet*, June 2013. [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2540.pdf

[62] D. Eastlake, “Transport Layer Security (TLS) Extensions: Extension Definitions,” *IETF RFC*, vol. 6066, January 2011.

[63] S. Santesson and H. Tschofenig, “Transport Layer Security (TLS) Cached Information Extension,” *IETF RFC*, vol. 7924, July 2016.

[64] D. Canright, “A Very Compact Rijndael S-Box,” *Naval Postgraduate School Technical Report*, NPS-MA-04-001, 2004.

[65] J. Guajardo, T. Guneysu, S. S. Kumar, C. Paar and J. Pelzl, “Efficient Hardware Implementation of Finite Fields with Applications to Cryptography,” *Acta Applicandae Mathematica*, vol. 93, no. 1, pp. 75Ű118, September 2006.

[66] T. Pornin, “Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA),” *IETF RFC*, vol. 6979, August 2013.

[67] M. Hedabou, P. Pinel and L. Beneteau, “Countermeasures for Preventing Comb Method Against SCA Attacks,” in *Information Security Practice and Experience – ISPEC 2005*, *Lecture Notes in Computer Science*, vol. 3439, pp. 85-96, April 2005.

[68] J. Fan, X. Guo, E. De Mulder, P. Schaumont, B. Preneel and I. Verbauwhede, “State-of-the-Art of Secure ECC Implementations: A Survey on Known Side-Channel Attacks and Countermeasures,” *IEEE International Symposium on Hardware-Oriented Security and Trust – HOST 2010*, pp. 76-87, June 2010.

[69] NIST, “Recommendation for Random Number Generation Using Deterministic Random Bit Generators,” *NIST Special Publication*, vol. 800-90A, rev. 1, June 2015.

[70] C. Meyer and J. Schwenk, “Lessons Learned From Previous SSL/TLS Attacks – A Brief Chronology Of Attacks And Weaknesses,” *Cryptology ePrint Archive*, Report 2013/049, January 2013.

[71] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti and P. Y. Strub, “Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS,” *IEEE Symposium on Security and Privacy*, pp. 98-113, May 2014.

[72] RISC-V, “The RISC-V Instruction Set Manual,” May 2016. [Online]. Available: https://riscv.org/specifications/

[73] MIT CSAIL Computation Structures Group, “Riscy Processors - Open-Sourced RISC-V Processors,” April 2017. [Online]. Available: https://github.com/csail-csg/riscy