An Energy-Efficient Reconfigurable DTLS Cryptographic Engine for Securing Internet-of-Things Applications

Utsav Banerjee, Andrew Wright, Chiraag Juvekar, Madeleine Waller, Arvind, and Anantha P. Chandrakasan
Dept. of EECS, Massachusetts Institute of Technology, Cambridge, MA, USA

***Abstract*—This paper presents the first hardware implementation of the Datagram Transport Layer Security (DTLS) protocol to enable end-to-end security for the Internet of Things (IoT). A key component of this design is a reconfigurable prime field elliptic curve cryptography (ECC) accelerator, which is 238× and 9× more energy-efficient compared to software and state-of-the-art hardware respectively. Our full hardware implementation of the DTLS 1.3 protocol provides 438× improvement in energy-efficiency over software, along with code size and data memory usage as low as 8 KB and 3 KB respectively. The cryptographic accelerators are coupled with an on-chip low-power RISC-V processor to benchmark applications beyond DTLS with up to two orders of magnitude energy savings. The test chip, fabricated in 65 nm CMOS, demonstrates hardware-accelerated DTLS sessions while consuming 44.08 μJ per handshake, and 0.89 nJ per byte of encrypted data at 16 MHz and 0.8 V.**

***Index Terms*—Cryptographic accelerator, Elliptic Curve Cryptography (ECC), Advanced Encryption Standard (AES), AES-GCM, Secure Hash Algorithm (SHA), Transport Layer Security (TLS), DTLS, Internet of Things (IoT), RISC-V, micro-processor, low-power, side-channel, hardware security.**

I. Introduction

THE Internet of Things (IoT) is an ever-growing network of wireless electronic devices always connected to the Internet - collecting, processing and communicating data. While the IoT promises to enable fundamentally new applications, it is important to guarantee that the communication channel between each sensor node and the cloud server is secure, even in the presence of untrusted and potentially malicious network infrastructure [1]. This is called end-to-end security, and protocols such as Datagram Transport Layer Security (DTLS) [2], [3] enable the establishment of mutually authenticated confidential channels between IoT sensor nodes and the cloud. DTLS employs elliptic curve-based public key cryptographic techniques to authenticate the two end points and establish shared secret keys, which are then used to encrypt application data. TLS version 1.3 has recently been standardized by the Internet Engineering Task Force (IETF), and is considered to be one of the most suited protocols for securing the IoT [1]. While this makes DTLS an ideal solution for IoT, the associated computational cost makes software-only implementations prohibitively expensive for resource-constrained embedded devices [4]. IoT devices are usually powered by batteries, which are expected to last several years, or through energy harvesting. Moreover, commercially available IoT platforms use micro-controllers with limited instruction and data memory. Therefore, it is essential to have a DTLS implementation which not only has minimal energy consumption but also comes with a small memory footprint. To address these challenges, we present the first hardware implementation [5] of DTLS 1.3, based on version 18 of the protocol draft [3]. Our reconfigurable elliptic curve cryptography (ECC) accelerator enables two orders of magnitude energy savings, while a dedicated DTLS engine offloads protocol control flow to hardware reducing program code and memory usage by an order of magnitude. An on-chip RISC-V processor exercises the flexibility of the cryptographic accelerators to demonstrate security applications beyond DTLS.

An overview of the DTLS protocol is presented in Section II, along with our high-level system architecture. Section III describes the RISC-V processor, Section IV provides architectural details of the energy-efficient cryptographic primitives and Section V describes the design of the DTLS engine. Measurement results from the test chip are presented in Section VI, and Section VII provides concluding remarks.

II. System Architecture

A. Transport Layer Security

The DTLS protocol can be divided into two major phases - *handshake* and *application data* (Fig. 1). The handshake starts with the client (sensor node) and the server agreeing upon protocol parameters such as the cryptographic algorithms to be used. Next, a Diffie-Hellman key exchange [6] is performed to establish a shared secret over the untrusted channel. The subsequent handshake messages are completely encrypted using keys derived from this shared secret. Following this, the client and the server authenticate each other through digital certificate verification. Finally, the two parties verify the integrity of the information exchanged in the above steps, to prevent man-in-the-middle attacks. At this point, a mutually authenticated confidential channel has been established between the client and the server. This channel can then be used, in the application data phase, to exchange data encrypted under a new set of keys derived from the handshake parameters.

A revised version of this paper was published in the IEEE Journal of Solid-State Circuits (JSSC) - DOI: 10.1109/JSSC.2019.2915203

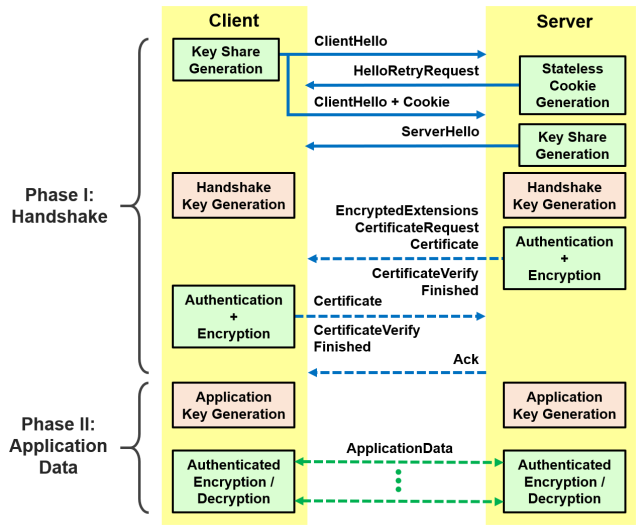


Fig. 1. Overview of the DTLS handshake protocol with digital certificate-based mutual authentication and key exchange (dashed arrows indicate that the messages are encrypted).

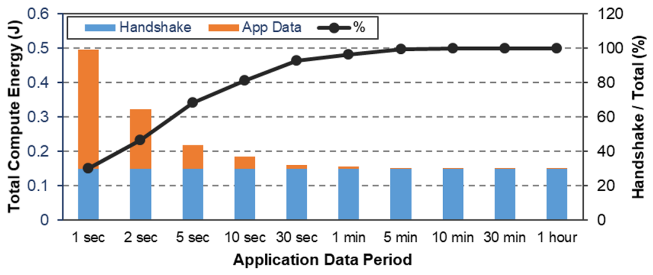


Fig. 2. DTLS computation energy breakdown and percentage of total compute energy spent in handshake, for $N = 32$ bytes of application payload, session duration $t_{session} = 1$ day and varying application data period $t_{appdata}$.

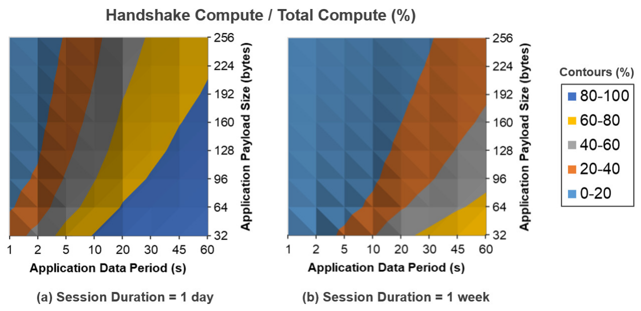


Fig. 3. Contour plots showing the percentage of total compute energy spent in handshake, for varying application payload size N and varying application data period $t_{appdata}$, for session duration of (a) 1 day and (b) 1 week.

The DTLS specification lists a set of recommended cryptographic algorithms, also known as *cipher suites*, to be used for performing the handshake and encrypting data. In this work, we consider DTLS connections implementing the TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 cipher suite, where elliptic curve cryptography [7] is used for endpoint authentication and key exchange, AES-128-GCM (Advanced Encryption Standard in Galois/Counter Mode) [8], [9] is used for authenticated encryption, and SHA2-256 (Secure Hash Algorithm 2) [10] is used for message hashing, key derivation and pseudo-random number generation. The handshake phase involves $\approx$ 100 invocations each of the AES-GCM and SHA primitives, which operate in blocks of 128 bits and 512 bits respectively; one ECDHE (Elliptic Curve Diffie-Hellman Key Exchange), and at least two ECDSA (Elliptic Curve Digital Signature Algorithm) operations (one ECDSA-Sign and at least one ECDSA-Verify). Once the handshake is complete, encryption or decryption of application data requires one invocation of AES-GCM per 128-bit block of data.

While the computation energy spent during each DTLS handshake is constant for a given cipher suite, the energy required during the application data phase is a direct function of the application payload size. Let us denote the handshake energy and the encrypted application data energy per byte of payload as $E_{handshake}$ and $E_{appdata}$ respectively, the session duration (time interval between two consecutive handshakes) as $t_{session}$ and the application data period (time interval between two consecutive application data transmissions) as $t_{appdata}$. Then, for N bytes of application payload, the total computation energy during a session is given by

$$E_{total} = E_{handshake} + (N \times \frac{t_{session}}{t_{appdata}} \times E_{appdata})$$

since the total number of data transmissions during a session is $t_{session}/t_{appdata}$. The fraction of energy spent in handshake computations is $E_{handshake}/E_{total}$. The session duration $t_{session}$ is dictated by security requirements of the application – more frequent handshakes (to establish new session keys), that is, smaller $t_{session}$, imply stronger security guarantees, e.g., medical devices authenticate more often than industrial sensors. The application data rate is calculated as $N/t_{appdata}$, which also depends on the application, e.g., industrial sensors typically send small packets of data every hour while medical devices send large amounts of data every minute or every second.

To understand the effect of application data rate on compute energy, we consider $E_{handshake} = 150$ mJ and $E_{appdata} = 125$ nJ as measured from an embedded software implementation of DTLS [4]. For devices handshaking once every day and payload size of $N = 32$, the breakdown of computation energy is shown in Fig. 2. We observe that the percentage of energy spent in DTLS handshake is around 30% when data is transmitted every second, and more than 99% when data is transmitted every hour. To further analyze the effects of these parameters, contour plots are shown in Fig. 3 for $t_{session} = 1$ day and $t_{session} = 1$ week. As expected, the handshake energy becomes a larger fraction of total energy for smaller N, larger $t_{appdata}$ and smaller $t_{session}$. We observe that the total computation energy for a software implementation of DTLS is of the order of 0.1-0.5 J, which is dominated by either handshake computations or application data encryption depending on the application parameters. Therefore, it is essential to design energy-efficient hardware to accelerate both handshake and application data computations for low-power IoT devices secured by DTLS.

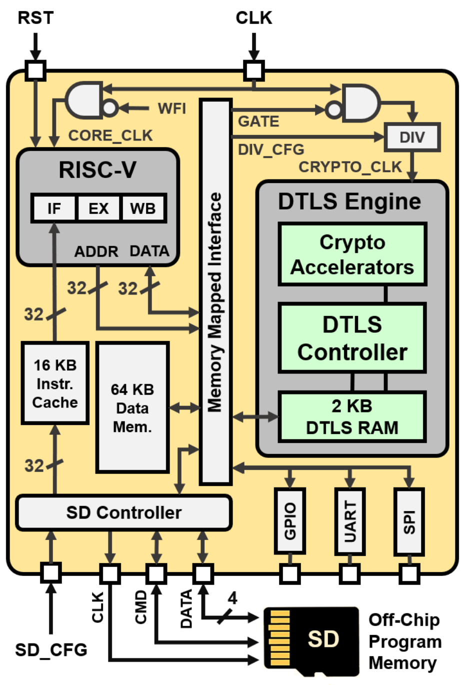


Fig. 4. System block diagram with an overview of the hardware modules implemented on the test chip.

B. Chip Overview

Fig. 4 shows the system block diagram. It consists of a 3-stage (IF: instruction fetch, EX: execute, WB: write back) RISC-V processor [11] supporting the RV32I instruction set, with 16 KB instruction cache and 64 KB data memory, and an SD (Secure Digital) card used as backing store for larger programs. A DTLS engine (DE), comprised of a protocol controller, a dedicated 2 KB RAM, and AES-128-GCM, SHA2-256 and prime field ECC primitives, accelerates both the handshake and application data phases of the DTLS protocol. Sleep mode is implemented on the RISC-V, to save power, by gating its clock when cryptographic tasks are delegated to the DE. The DE uses a dedicated hardware interrupt to wake the processor on completion of these tasks. The DE is clocked by a software-controlled divider to decouple the processor operating frequency from the long critical paths in the ECC accelerator. A memory-mapped interface provides access to the DTLS engine, through the DTLS RAM, not only for executing DTLS protocol workloads but also for standalone computations in the cryptographic accelerators. The same interface is also used to communicate with peripherals such as GPIO (General Purpose Input / Output), UART (Universal Asynchronous Receiver / Transmitter) and SPI (Serial Peripheral Interface) through RISC-V software. This memory-mapped interface, along with the accelerator interrupts, behaves very similarly to the Rocket Custom Coprocessor (RoCC) interface used by the Rocket RISC-V core [12] to interface with accelerators. However, the memory-mapped approach does not require building custom instructions, thus simplifying the software tool-chain.

III. RISC-V Microprocessor

The RISC-V processor on the test chip is a 32-bit core, designed in Bluespec System Verilog, supporting the integer subset of instructions (RV32I) with user and machine privilege modes. The RISC-V core was designed to not only seamlessly interface with the DTLS engine but also efficiently implement the DTLS protocol in software. In order to support the large instruction storage required by the DTLS software (detailed in Section VI), an SD card is used as off-chip program memory with the processor instruction cache reading program blocks from the card through an on-chip SD controller.

The instruction cache, being backed by an SD card, has to deal with a larger memory access granularity (512 bytes vs 64 bytes) and a longer memory access latency compared to typical microprocessor caches backed by DRAM modules. To match the access granularity of the SD card, the instruction cache was designed with a block size of 512 bytes, and the cache was made 4-way set associative with a tree-based pseudo-LRU (least recently used) replacement policy to reduce the number of cache misses. Apart from the 16 KB SRAM for storing instruction words, the cache also contains an 84-byte register array for storing tags. The tag array accesses only a single way's tag at a time, instead of all four tags, and the 16 KB SRAM only accesses one 32-bit instruction at a time, thus reducing access power. To reduce the overhead of accessing one way at a time, the cache has an MRU (most recently used) way predictor to estimate which way will be used for each instruction fetch. This way predictor reuses the meta-data from the replacement policy to determine the most recently used way. To further reduce the cache access penalty, a register is used to cache the last tag read from the tag array so that the tag array is accessed only when switching cache lines.

The SD controller is designed to reduce miss latency by using the SD bus protocol [13] where card data is accessed 4 bits at a time, instead of the bit-serial SPI mode. Both SDHC and SDXC cards are supported, with clock frequencies up to 25 MHz. The SD clock is generated from the system clock through a clock divider configured externally (through SD_CFG). Memory-mapped SPI and GPIO peripherals on the chip are used to interface with off-chip components. The UART peripheral, driven by a custom software library, is used to send debug messages to the host computer during testing.

The RISC-V core is also equipped with an interrupt controller to handle interrupts from the cryptographic accelerators, the peripherals as well as off-chip. The interrupts can be

TABLE I
Comparison of our RISC-V core with state of the art

Design	Arch	Tech (nm)	Voltage (V)	Energy (pJ / cyc)
Duran *et al.*, LASCAS 2017 [15]	RISC-V RV32IM	130	1.2	167
Uytterhoeven *et al.*, ESSCIRC 2018 [16]	RISC-V RV32IM	28	0.38	8.81
This work	RISC-V RV32I	65	0.8	40.36

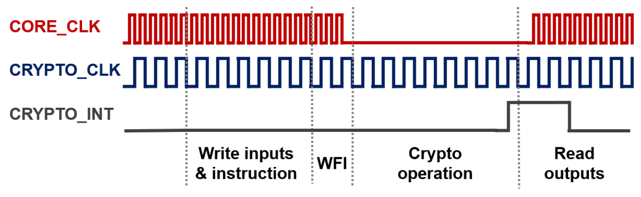


Fig. 5. Processor clock gating during WFI (wait for interrupt).

individually enabled and programmed to be edge or level triggered through software. Executing the *wait for interrupt* (WFI) instruction gates the clock feeding the RISC-V core and its instruction cache and data memory, as shown in Fig. 5, which enables power savings when the DTLS engine is accelerating cryptographic computations. The interrupt controller wakes them up when the appropriate interrupt is received.

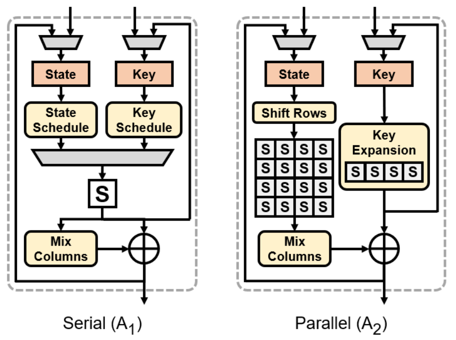
When executing the Dhrystone benchmark, our RISC-V processor consumes 40.36 μW/MHz at 0.8 V, and achieves 0.96 DMIPS/MHz which is comparable to the ARM Cortex-M0 processor [14]. Table I compares the energy-efficiency of our design with some recent embedded-scale processor implementations.

IV. Cryptographic Primitives

As discussed in Section II, DTLS requires not only symmetric cryptography primitives such as AES and SHA, but also public key protocols using ECC. In this section, we provide details of the energy-efficient implementations of these primitives, including architectural optimizations, design space exploration and on-chip characterization results.

A. AES in Galois/Counter Mode (AES-GCM)

The DTLS protocol uses AES-128 in the GCM mode for authenticated encryption with associated data (AEAD), that is, it simultaneously guarantees confidentiality, integrity, and authenticity of the data. The AES-128 cipher uses 128-bit keys to encrypt 128-bit plain-text blocks over 10 iteration rounds, with each round performing a set of linear and non-linear transformations on the cipher's internal state. The S-Box is the most important non-linear component of AES, used both in encryption and key expansion. In this work, we have used the low-power low-area S-Box design proposed in [18].



Architecture	Area (kGE)	Cycles	Energy (pJ / bit)
Serial (A_1)	2.8	336	14.2
Parallel (A_2)	8.6	11	3.1

Fig. 6. Comparison of AES architectures - A_1: serial and A_2: parallel.

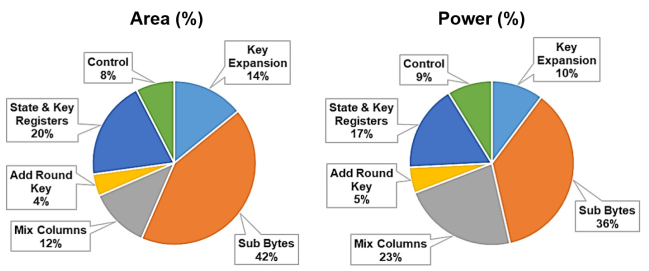


Fig. 7. Simulated area and power breakdown of the 128-bit data-path 11-cycle AES design.

To explore the effects of AES data-path size on area and energy-efficiency, we implemented two different AES architectures, as shown in Fig. 6:

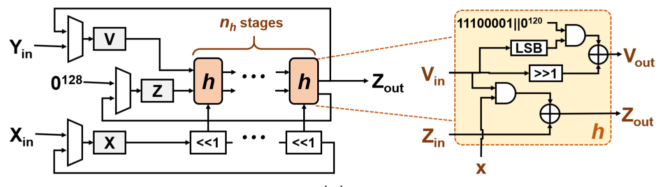
- A_1, with 8-bit data-path and one S-Box, processes the state and the round key on separate cycles 8 bits at a time, and takes 336 cycles to encrypt a block.
- A_2, with 128-bit data-path and 20 S-Boxes, processes the state and the round key together in a single cycle, and takes 11 cycles to encrypt a block.

The 8-bit architecture A_1 replicates the optimizations proposed in [21] and [22] to reduce the number of temporary registers. Fig. 6 compares the area, performance and energy-efficiency of the two designs, as determined from post-synthesis simulations in 65 nm LP process at 1.2 V. The 128-bit parallel design A_2

TABLE II
Comparison of our AES-128 with State of the Art

Design	Tech	Area		Cycles	Voltage	Energy
	(nm)	(mm^2)	(kGE)	/ Block	(V)	(pJ / bit)
Hamalainen *et al.*, EUROMICRO 2006 [19] [a]	130	-	3.2	160	1.2	37.5
Mathew *et al.*, JSSC 2011 [20]	45	0.15	-	5	1.1	2.3
					0.32	0.5
Mathew *et al.*, JSSC 2015 [21]	22	0.0022	1.9	336	0.9	30.1
					0.34	5.9
Zhang *et al.*, VLSIC 2016 [22]	40	0.0043	2.3	336	0.9	8.9
					0.47	2.2
This work	65	0.015 [b]	10.6 [b]	11	0.8	4.08 [c]

[a] Post-synthesis area and power reported in [19] [b] Area of final placed-and-routed design [c] Measured energy



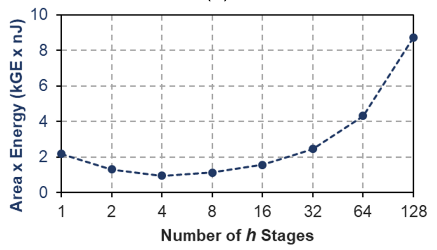


Fig. 8. (a) Implementation of GHASH Galois multiplier in hardware and (b) effect of number of multiplier stages (n_h) on area and energy.

is 4.6× more energy-efficient and 30× faster, at the cost of 3× increase in logic area. Fig. 7 shows the breakdown of area and power consumption of different components of this design. The S-Boxes account for a large fraction of area and power, which reaffirms our choice of composite-field S-Box.

Table II compares our AES-128 design with state of the art, both in terms of area and energy. Our design is smaller than the 128-bit data-path 2-stage pipelined AES design in [20], while having comparable energy consumption, after accounting for voltage and technology scaling. In comparison to [19], [21], [22], which are all 8-bit data-path serial implementations, our design is more energy-efficient, when accounting for voltage and technology scaling, but at the cost of larger area. Note that our AES could not be characterized at voltages smaller than 0.8 V because all logic and SRAMs on our chip are powered by a single supply rail. Also, our measured AES energy includes leakage from the entire chip, since other components were clock-gated but not power-gated.

AES-GCM uses the AES forward cipher for both encryption and decryption, and a Galois multiplication-based special hash function called *GHASH* for authentication [9]. AES-GCM employs the counter mode of operation, which concatenates a counter value with the initialization vector IV, and encrypts it with the secret key using AES. The result of this encryption is then XOR-ed with the plain-text to generate the cipher-text. Like all counter modes, this essentially acts as a stream cipher, therefore it is important to ensure that a different IV is used for each stream that is encrypted.

The Galois multiplier in GHASH can be implemented in hardware using one or more copies of the basic function which we denote as h: $Z_{i+1} = Z_i \oplus x_i \cdot V_i$ and $V_{i+1} = (V_i >> 1) \oplus \{\text{LSB}(V_i)\} \cdot (11100001||0^{120})$, as shown in Fig. 8a. A Galois multiplier with n_h stages requires $128/n_h$ cycles per multiplication, and the number of h-stages directly affects area, cycles per operation and energy consumption. Multiple Galois multipliers were synthesized to determine a suitable architecture, and their area-energy products were plotted as a function of the number of h-stages, as shown in Fig. 8b. We observed that a 32-cycle design, with $n_h = 4$, has the lowest area-energy product, hence this version was used in our AES-GCM implementation. Since AES-GCM involves computing GHASH on the cipher-text, our design performs encryption and Galois multiplications in parallel, at 32 cycles per 128-bit data block. For m blocks of associated data and n blocks of plain-text (cipher-text), it takes $54 + 32 \cdot (m + n)$ cycles to encrypt (decrypt) and generate (verify) the GCM tag, where the fixed 54-cycle overhead accounts for computing the hash key, hashing the data length, computing the tag as well as configuring the key, IV and other encryption parameters. The final placed-and-routed design occupies 29.9 kGE area, including the 10.6 kGE AES, of which about 25% is attributed to registers used to store input/output data, keys, intermediate states and configuration values. Energy consumption of our design is 11.88 pJ/bit at 0.8 V.

B. Secure Hash Algorithm (SHA2)

The SHA2-256 hash algorithm compresses messages of arbitrary lengths ($< 2^{64}$ bits) and generates a unique 256-bit message digest. Since SHA2-256 operates on 512-bit blocks, the input message is padded to a multiple of 512 bits. The internal state of the hash function is initialized according to the SHA2 specification [10]. The *Message Schedule* takes 512-bit blocks of the padded message and sends 32-bit words W_t to the main SHA2-256 *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 continues till the entire message has been processed, and the final value of the state is the message digest.

Fig. 9 shows details of the round function. The internal state consists of 16 32-bit registers $H_0 - H_7$ and $a - h$. The Σ_0, Σ_1, Maj and Ch functions are specified in [10], while $\boxplus$ denotes 32-bit addition modulo 2^{32}, that is, the final carry is ignored. $H'_0 - H'_7$ and $a' - h'$ denote the updated state values after one iteration. Although the state of the hash function is defined by $H_0 - H_7$, $a - h$ and the message schedule, we note that $H_0 - H_7$ completely define the SHA2-256 state after every 64 iterations of the round, that is, after every 512-bit block has been processed. This property can be exploited to implement efficient running hashes, as will be discussed in Section V.

The critical paths in the round function were implemented using a combination of carry-save and ripple-carry adders to reduce latency. Messages are sent to the SHA2 core one byte

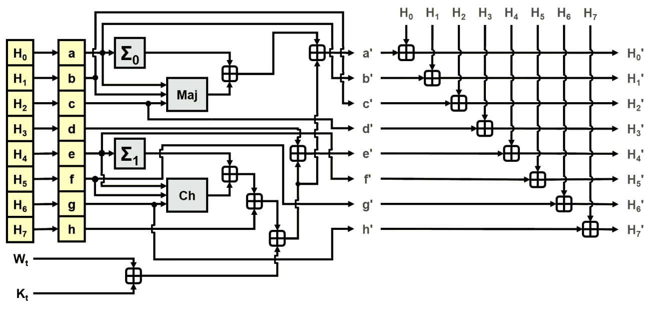


Fig. 9. Implementation of SHA2-256 round function in hardware.

at a time, and a counter is used to track the input data length, which is used by the SHA2 core to perform message padding. The SHA2-256 core computes $a' - h'$ in parallel to achieve increased energy-efficiency. Our final design occupies 18.2 kGE, and takes 65 cycles to process a 512-bit input block, while consuming 4.43 pJ/bit at 0.8 V.

C. Reconfigurable Prime Field ECC

Elliptic curve cryptography (ECC) is used in DTLS for both key exchange and digital signature protocols. We consider two types of elliptic curves over finite fields $\mathbb{F}_p$ of large prime characteristic p – short Weierstrass curves ($y^2 = x^3 + ax + b$) and Montgomery curves ($by^2 = x^3 + ax^2 + x$). All other prime curves (for $p \neq 2, 3$) can be transformed into the short Weierstrass form with a simple change of variables [7]. ECC-based protocols can choose from a large set of standard curves, e.g., NIST, Curve25519, Brainpool, SEC and ANSSI. While existing literature in ECC hardware mostly focus on implementing a single family of curves [23], [24], [25], [26], a similar approach is not suitable for DTLS because the standard allows a much wider choice of curves. This provides the motivation for our reconfigurable prime field ECC design, and we support curves over any prime up to 256 bits, which correspond to at most 128 bits of security.

The fundamental operations used in ECC are *point addition* ($R = P + Q$), and *point doubling* ($R = P + P$). Repeated additions of a point P with itself is called “elliptic curve scalar multiplication” (ECSM). For any scalar k, the multiple kP is computed as a series of point doubling (DBL) and point addition (ADD) operations, which can be decomposed into arithmetic in the finite field $\mathbb{F}_p$. This makes efficient modular arithmetic integral to both software and hardware implementations of ECC. Fig. 10 describes our energy-efficient ECSM hardware, which can be configured with prime p of variable length t (up to 256 bits) and curve parameters a and b. Given scalar k and point $P(x, y)$, it generates $Q = kP$.

One of the key components of our design is an efficient modular multiplier, shown in Fig. 10. In order to support arbitrary prime fields, it performs multiplication with interleaved modular reduction [27]. Three adders are used for this computation, one for addition and two for reduction. The reduction uses conditional subtractions, all performed in the same cycle so that the modular multiplication is constant time and there is no potential timing side-channel. The same circuitry can be re-used for modular addition.

While most ECC designs choose 16-bit or 32-bit data-paths for modular arithmetic, we have used full 256-bit adders for energy-efficiency, with higher bits of the data-path gated when working with smaller primes. Design space exploration was performed for 256-bit modular adders with different data-path sizes, as shown in Fig. 11. Clearly, the total area doesn’t scale linearly with the data-path width due to the fixed overhead of the 256-bit registers required to store the inputs and output. When scaling up from 16-bit to 256-bit data-path, total area increases by 1.8×, while energy per operation decreases by 6.8×, primarily due to reduced control circuitry and muxing logic. Table III shows the simulated energy consumption of our modular adder and multiplier at 1.2 V and 20 MHz.

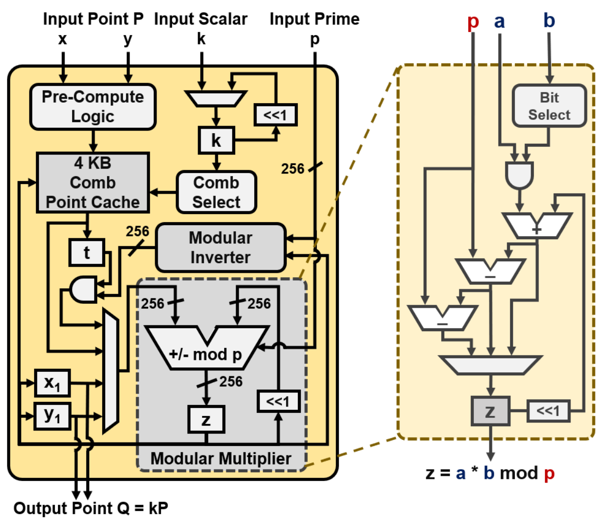


Fig. 10. Block diagram of the reconfigurable prime-field elliptic curve cryptography accelerator, along with detailed architecture of the modular multiplier implementing interleaved modular reduction.

Prior work on hardware implementations of ECC re-use the modular multiplier to perform modular inversion using Fermat’s theorem: $x^{-1} = x^{p-2} \bmod p$ [7]. This method uses repeated modular multiplications (384 on average for 256-bit primes) for exponentiation. Therefore, inversion using Fermat’s theorem (I_{Fermat}) is slow, but doesn’t require any additional logic area. In this design, we make an energy-area trade-off and implement dedicated hardware [17] to perform modular inversion using the extended Euclidean algorithm (I_{Euclid}) [7], which involves modular additions, subtractions and bit-shifts. Similar to the multiplier, our inverter also consists of 256-bit adders for energy-efficiency. From Table III, energy consumption of the two types of inversions are found to be related to multiplication (M) as: $I_{Fermat} \approx 384M$ and $I_{Euclid} \approx 3M$, indicating that I_{Euclid} is 128× more efficient, albeit at the cost of increased logic area.

Having optimized the modular arithmetic implementations, the next step is to select an efficient ECSM algorithm. Traditional window-based ECSM [7] requires 256 DBL and 64 ADD operations for window-size $w = 4$. Instead, a pre-computation-based comb algorithm [7], [28] is implemented, which involves 64 DBL and 64 ADD operations, thus reducing ECSM energy by 2.5×. A 4 KB cache stores pre-computed comb data for up to six points, including generator points and

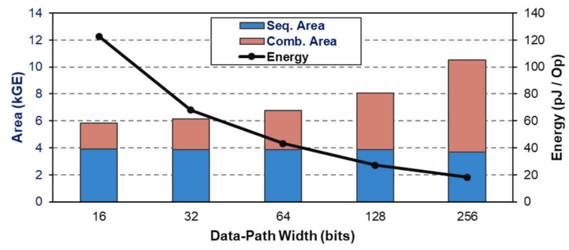


Fig. 11. Comparison of modular adder architectures, with different data-path widths, in terms of area (sequential and combinational) and energy.

TABLE III
SYNTHESIS RESULTS FOR 256-BIT MODULAR ARITHMETIC

Operation	Cycles / Op	Energy (nJ / Op)
Add / Sub	1	0.02
Mul	256	4.04
Inv (Fermat)	98304	1552
Inv (Euclid)	$\approx$ 720	12.9

public keys, which is specifically used to speed up the DTLS handshake, as will be explained in Section V.

The final optimization step in our design is the appropriate choice of coordinates for elliptic curve points. Resource-constrained ECC implementations [23], [24], [25], [26] typically use projective coordinates to avoid modular inversions in the ECSM inner loop, at the cost of extra multiplications and a final expensive Fermat inversion. In projective coordinates, the costs of point opertions are ADD = $8M$ and DBL = $11M$. Since we have an efficient dedicated modular inverter, we use affine coordinates where ADD = $2M + I$ and DBL = $3M + I$. The total ECSM costs of the projective and affine coordinate representations are calculated as $E_{proj} = 64 \times (8M + 11M) + 4M + I_{Fermat} = 1604M$ and $E_{aff} = 64 \times (5M + 2I_{Euclid}) = 704M$. Therefore, the use of affine coordinates saves $\approx 2\times$ in energy by trading off the extra multiplications for cheaper Euclid inversions.

Public-key algorithms are prone to side-channel attacks due to their expensive computations and long execution times. One such attack is *simple power analysis* (SPA). Simple double-and-add ECSM algorithms perform conditional point additions in the outer loop depending on whether the corresponding bit in the secret scalar is a 1. Since DBL and ADD involve distinct arithmetic, the power consumption of the chip can leak this information. For reference, we demonstrate an SPA attack on a software implementation of this algorithm, as shown in Fig. 12. The slower operations – multiplication and inversion, can be clearly inferred from a single power trace, and the bits of the secret scalar can be successfully determined. In order to prevent SPA attacks, we use a zero-less signed digit (ZSD) representation of the scalar [28] in conjunction with the comb technique, which transforms the scalar to have no zero bits, thus avoiding conditional point additions. This also reduces the number of pre-computed comb points per ECSM from 16 to 8. Fig. 13 shows power traces of our SPA-secure implementation for 10 random scalars overlaid together, where both DBL and ADD are computed at each iteration of the outer loop, irrespective of the bits of the scalar.

The binary scalar $k = (k_{t-1}, k_{t-2}, \cdots, k_1, k_0)_2$ needs to be odd to have a valid ZSD form, that is, the least significant bit $k_0 = 1$ [28]. 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. We use a compact scalar encoding, which we denote as ZSD*, of the ZSD scalar where the 1-bit represents '1' and the 0-bit represents '-1', similar to [29]. We prove that this compact form of scalar k can be computed "on-the-fly" as $\text{ZSD}^*(k) = (1, k_{t-1}, \cdots, k_2, k_1)_2$

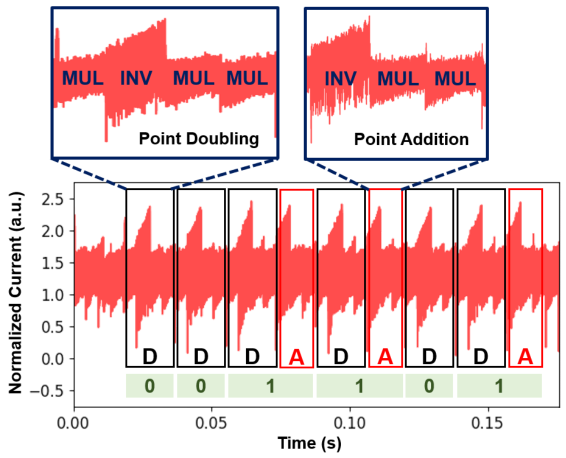


Fig. 12. Measured power trace demonstrating SPA attack on the simple double-and-add ECSM algorithm implemented in software on the RISC-V processor. The double (D) and add (A) steps are marked, along with their key constituent modular arithmetic operations - multiplication (MUL) and inversion (INV). Also shown are bits of the secret scalar successfully recovered from this trace.

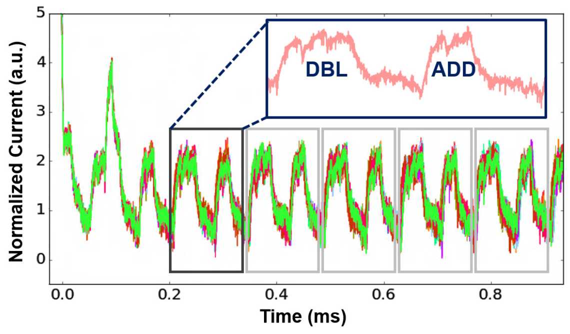


Fig. 13. Measured power traces of the SPA-secure hardware ECSM, for 10 random scalars, overlaid together for comparison. The sets of point doubling (DBL) and point addition (ADD) operations are shown in boxes, indicating that the double-and-add patterns are constant irrespective of the secret scalar.

since the following equation holds:
$(1, k_{t-1}, \cdots, k_2, k_1)_2 =$

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

Therefore, no additional circuitry is required to convert k to the ZSD* form. The SPA countermeasure introduces 5 extra point additions, on average, for 256-bit scalars [7], which translates to $\approx 4\%$ energy and performance overhead.

More sophisticated side-channel attacks on ECC exist [30], which involve statistical metrics, e.g., correlation, and therefore require several power traces for a single scalar. Since the same scalar is never used twice for any of the ECSM computations during the DTLS handshake, these attacks are not particularly relevant to our main application. For other ECC-based protocols, appropriate countermeasures, usually requiring some form of input randomization, can be easily implemented using software.

For a 256-bit short Weierstrass curve, our design takes $\approx$ 320k cycles for comb pre-computations and $\approx$ 180k

TABLE IV
COMPARISON OF OUR RECONFIGURABLE ECC DESIGN WITH STATE OF THE ART

Design	**Tech (nm)**	**Voltage (V)**	**Logic Area (kGE)**	**Supported Curve(s)**	**Cycles / ECSM**	**Energy [a] / ECSM (μJ)**
Hutter *et al.*, WISTP 2011 [23] [b]	350	3.3	9.5	NIST P-192	753k	1423.6
Roy *et al.*, ECC 2013 [24] [b]	32	1.0	26	SEC P-160	250k	2.25
				NIST P-192	350k	3.15
Pessl *et al.*, RFIDSec 2014 [25] [b]	130	1.2	8.6	NIST P-160	100k	4.4
Hutter *et al.*, CHES 2015 [26] [b]	130	1.2	32.6	Curve25519	811k	56.8
This work (Reconfigurable ECC)	65	0.8	65.5 [c] (49.1 [d])	**All prime curves up to 256 bits**		
				160-bit	74k	2.22 [e]
				192-bit	102k	3.11 [e]
				256-bit	180k	6.47 [e]

[a] Base point ECSM energy [b] Post-synthesis area and power reported in [23], [24], [25], [26]
[c] Area of final placed-and-routed design [d] Synthesized area for comparison [e] Measured energy

cycles for SPA-secure ECSM. Table V compares the measured execution time and energy consumption of our hardware with an equivalent software implementation on the RISC-V at 0.8 V and 16 MHz. As described earlier, our reconfigurable ECC supports all short Weierstrass and Montgomery curves over prime fields up to 256 bits. Fig. 14 shows measured base point ECSM performance over different curve sizes, generated using the NIST curves for 160, 192, 224 and 256 bit primes. ECSM performance and energy scale approximately cubically with size of the prime. Since the prime size is directly related to security, our reconfigurable ECC can be used to scale security and efficiency, depending on the application requirements.

Our configurable ECC architecture can be easily scaled to larger prime fields (such as 384-bit or 521-bit primes), in order to support security levels higher than 128-bit, by using even wider data-path adders and small changes to the control logic. Table IV compares our design with recent work in ECC hardware. Our design is the most energy-efficient and flexible, but has larger area owing to the dedicated modular inverter (31k GE) and full data-path modular multiplier (11.8k GE). Reconfigurability of our ECC core is also responsible for some

TABLE V
MEASURED 256-BIT PRIME CURVE ECC PERFORMANCE

Computation	**Time (s)**	**Energy (μJ)**
Software Comb + ECSM	8.5	4180
Hardware Comb	0.02	11.1
Hardware ECSM	0.012	6.47

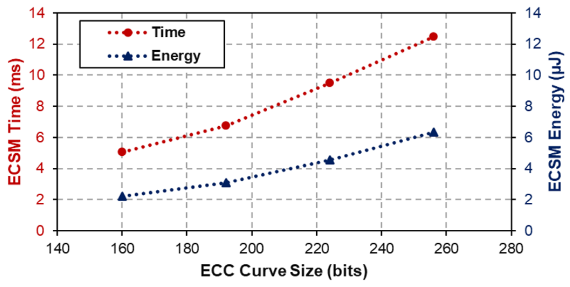


Fig. 14. Base point ECSM performance over different curve sizes.

of the area overheads, since fixed prime field arithmetic (such as NIST primes) can be implemented with smaller logic with hard-wired parameters.

V. DTLS ENGINE

At the core of DTLS is its state machine, which controls all handshaking protocols and related computations. Since the DTLS state machine supports a variety of configurations [2], [3], software implementations can be error-prone and has lead to attacks in the past [31]. To avoid such issues, we enable only a carefully chosen secure subset of all the configurations supported by DTLS. In this work, we have implemented the cipher suite with ECDHE, ECDSA, AES-128-GCM and SHA2-256, requiring mandatory server/client authentication. The *Client Certificate URL* extension is used, that is, client certificates are not transmitted. The *Cached Information* extension is made optional, and the server decides whether to use it during the handshake. Certificate Authority (CA) public keys are cached by both parties, and CA certificates are never exchanged (still maintaining compliance with the TLS specification).

Fig. 15 shows the architecture of our DTLS engine (DE), with its key components – (1) energy-efficient cryptographic accelerators, (2) DTLS controller and (3) DTLS RAM. The efficient cryptographic primitives, described in Section IV, not only accelerate DTLS computations but can also be accessed individually through RISC-V software to implement standalone protocols. The DTLS RAM and DTLS controller are discussed in detail in the following subsections.

A. DTLS RAM

The 2 KB DTLS RAM can be divided into three sections - DTLS micro stack, DTLS Config memory and Accelerator Config memory. The 1.25 KB DTLS micro stack acts as scratch-pad for temporary variables computed during the DTLS handshake, including DRBG states and DTLS session keys. The DTLS stack is not accessible through the memory-mapped interface so that secret session information, including encryption keys, cannot be read by software. The 0.45 KB DTLS Config memory is used to store public keys, secret keys and certificate details, which can be programmed through the RISC-V processor, while the remaining 0.3 KB Accelerator

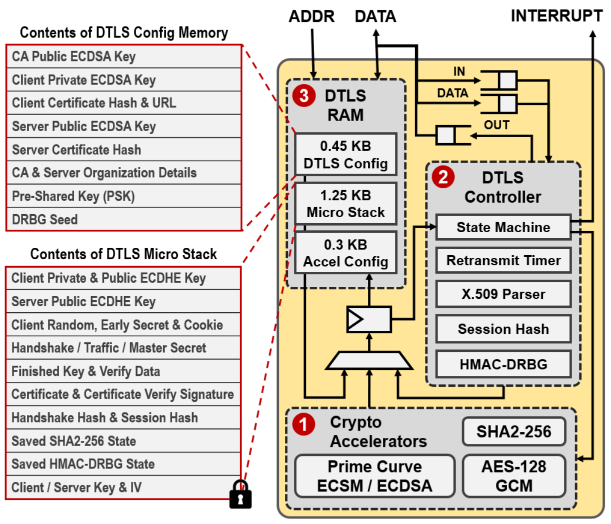
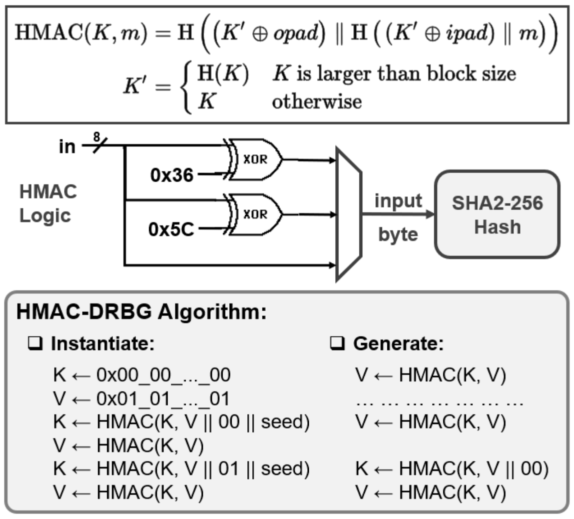


Fig. 15. Architecture of DTLS engine along with contents of DTLS RAM.


$\mathrm{HMAC}(K,m) = \mathrm{H}\left((K' \oplus opad) \parallel \mathrm{H}\left((K' \oplus ipad) \parallel m\right)\right)$
$K' = \begin{cases} \mathrm{H}(K) & K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$
in
8
HMAC Logic
0x36
0x5C
XOR
XOR
input byte
SHA2-256 Hash

```
HMAC-DRBG Algorithm:
❑ Instantiate:                         ❑ Generate:
    K ← 0x00_00_..._00                     V ← HMAC(K, V)
    V ← 0x01_01_..._01                     … … … … … … …
    K ← HMAC(K, V || 00 || seed)           V ← HMAC(K, V)
    V ← HMAC(K, V)
    K ← HMAC(K, V || 01 || seed)           K ← HMAC(K, V || 00)
    V ← HMAC(K, V)                         V ← HMAC(K, V)
```

Fig. 16. HMAC logic along with details of DRBG computations.

Config memory stores accelerator configuration values for standalone cryptographic operations. Contents of the config memory and the micro stack are detailed in Fig. 15.

B. DTLS Controller

The DTLS controller implements a micro-coded DTLS 1.3 state machine for pseudo-random number generation, key schedule, session transcripts, encrypted packet framing, parsing and validation of X.509 digital certificates and re-transmission timeouts, as shown in Fig. 15. Details of some key components of the DTLS controller are discussed next.

1) HMAC-DRBG and HKDF: An HMAC-based Deterministic Random Bit Generator (HMAC-DRBG) [34] is used to generate cryptographically secure pseudo-random numbers, while an HMAC-based Key Derivation Function (HKDF) [33] is used to compute DTLS handshake and session keys. Both HMAC-DRBG and HKDF use the SHA2-256 cryptographic accelerator to efficiently compute HMACs (keyed-hash message authentication codes) [32], as shown in Fig. 16. HMAC uses two passes of the SHA2-256 hash function, with the key XOR-ed with the appropriate *pad* – repeated bytes valued `0x36` and `0x5C` for the inner and outer passes respectively. The HMAC inputs are loaded from the DTLS RAM into a temporary register and bytes are shifted out to the hash module, with or without padding. The micro stack is used to store any intermediate values computed during HMAC.

The HMAC-DRBG algorithm involves two 256-bit values K and V, and operates in two phases. The DRBG is first initialized using the input seed material (as obtained from the DTLS Config memory), also known as the *Instantiate* phase. During the *Generate* phase, pseudo-random numbers are generated in V, 256 bits at a time, by repeatedly computing HMAC(K,V), followed by an update of K and V, as shown in Fig. 16, which are then stored in the micro stack. The DRBG is initialized (seeded) only once at the time of device setup, and it can be used for up to 2^{48} invocations of the *Generate* step, with up to 2^{19} bits generated in each invocation, as per the NIST DRBG specification [34]. Since the DRBG *Generate* function

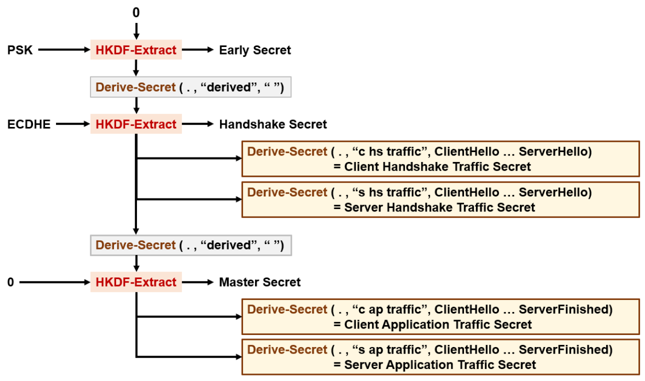

0
PSK
HKDF-Extract
Early Secret
Derive-Secret (. , "derived", " ")
ECDHE
HKDF-Extract
Handshake Secret
Derive-Secret (. , "c hs traffic", ClientHello … ServerHello) = Client Handshake Traffic Secret
Derive-Secret (. , "s hs traffic", ClientHello … ServerHello) = Server Handshake Traffic Secret
Derive-Secret (. , "derived", " ")
0
HKDF-Extract
Master Secret
Derive-Secret (. , "c ap traffic", ClientHello … ServerFinished) = Client Application Traffic Secret
Derive-Secret (. , "s ap traffic", ClientHello … ServerFinished) = Server Application Traffic Secret

Fig. 17. TLS 1.3 key schedule [2].

is used 3 times (for generating client random and scalars for ECDHE and ECDSA-Sign) during a DTLS handshake (DRBG not required during application data exchange), it need not be re-seeded for $\approx 9.4 \times 10^{13}$ handshakes, which exceeds the life of the IoT device.

The SHA2-256-based HKDF algorithm also works in two steps – *Extract* and *Expand*. In the *Extract* phase, a non-secret *salt* and input keying material *IKM* are used to calculate the pseudo-random key *PRK* = HMAC(*salt*, *IKM*). In the *Expand* phase, output keying material is generated in 256-bit blocks $T[k]$ using *PRK* and some application specific *info* as:
HKDF-Expand (*PRK*, *info*, L) = $T[1] \parallel T[2] \parallel \cdots \parallel T[L/32]$
where $T[k]$ = HMAC (*PRK*, $T[k-1] \parallel$ *info* $\parallel k$) for $k > 0$, $T[0]$ = *null*, and L is length of output keying material in bytes.

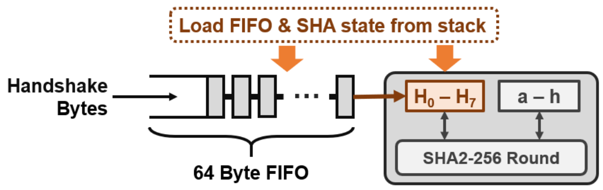
Along with HKDF-Extract, the TLS 1.3 *Key Schedule* [2] uses the following function for key derivation:
Derive-Secret (Secret, Label, Messages) =
HKDF-Expand (Secret, `0x0020` ‖ `0x746C733133` ‖ Label ‖ SHA2-256 (Messages), 32)
The detailed key schedule is shown in Fig. 17, where PSK refers to the pre-shared key (if PSK is not in use, it is replaced with a string of zero bits) and ECDHE refers to the shared secret computed during the Diffie-Hellman key exchange. The

handshake and application traffic secrets are used to generate AES-GCM key and IV pairs (using further invocations of HHKDF-Expand, as specified in [2]) during the handshake and application data phases respectively.

While our prototype chip uses SRAMs and flip-flops as the only on-chip storage elements, a commercial product replicating this design would replace the DTLS config and micro stack with non-volatile memory so that the IoT device can enable power gating while still retaining the configuration values, DRBG state and session keys.

2) Session Transcript: The DTLS handshake involves 6 session hash (transcript) computations, that is, hash of the concatenation of all messages exchanged till that point in the handshake. Software implementations of DTLS typically save all handshake messages, and compute the hash over all of them every time a transcript is required. Handshakes can be as large as 2-3 KB and repeatedly reading them from SRAMs can be very expensive. To eliminate the need to store the entire handshake, we implement a *running hash* by exploiting the property of SHA2-256 that the internal registers $H_0 - H_7$ completely define the hash state every time a 512-bit block has been digested, as discussed in Section IV. Fig. 18 provides an overview of our session hash architecture. Handshake bytes are pushed into a 64-byte FIFO, and a 512-bit block is sent to the SHA2-256 core whenever the FIFO is full. This ensures that session hash computations always digest data in blocks of 64 bytes, except for the last block, and when computing the hash $H(m)$ of an N-byte message m, the *intermediate hash* of $\lfloor N/64 \rfloor$ blocks of m is stored in $H_0 - H_7$.

In Fig. 18, we illustrate this technique using a simple example where the DTLS controller needs to compute SHA2-256 (*ClientHello* || *ServerHello*). After every session hash, the FIFO state (containing any un-hashed bytes) and the registers $H_0 - H_7$ are copied to the DTLS stack, so that the SHA2 core can be used for other computations. This is particularly useful for later phases of the handshake which involve hashing large digital certificates. Our proposed approach reduces the total session transcript memory usage from several kilobytes down to only 96 bytes – 64 bytes for the SHA2 state and up to 32 bytes for the un-hashed portions of the messages.



Session Hash Example

- Generate ClientHello (206 bytes):
  - Compute intermediate hash of first 192 bytes (3 blocks)
  - Save hash state $H_0 - H_7$
  - Save remaining un-hashed 14 bytes of message
- Release SHA2-256 core for other computations
- Receive ServerHello (125 bytes):
  - Restore hash state $H_0 - H_7$
  - Append previously un-hashed 14 bytes to new message
  - Compute hash of 139 bytes

Fig. 18. Efficient session hash computation for DTLS handshake.

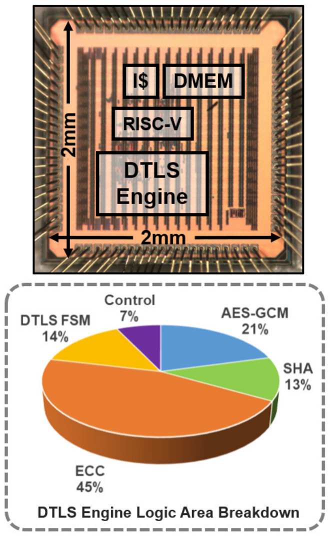
3) ECC Computations in DTLS: The reconfigurable ECC core is used to perform both ECDH and ECDSA-Sign/Verify computations, where the deterministic ECDSA scheme [35] is used to securely generate signatures. The DTLS handshake involves up to 7 ECSM computations, and we have seen in Section IV that ECSM energy can be reduced by 2.5× if pre-computed comb points are available. Our ECC comb point cache supports up to 6 pre-computed base points, which are used to minimize the energy consumption of the ECDHE_ECDSA handshake. Comb points are computed and stored for the curve generator point G, the CA public key Q_{CA} and the server public key Q_{SRV}. This one-time pre-computation requires around 33 μJ of energy, which gets amortized over all the subsequent handshakes, but provides up to 2.2× reduction in the energy consumption of each DTLS handshake. The pre-computation for G is essential for both ECDH and ECDSA, while pre-computations for Q_{CA} and Q_{SRV} are used to verify signatures from the CA and the server respectively. The rest of the point cache is used for ECDH and ECDSA with random points without corrupting the stored points required by DTLS.

4) DTLS State Machine: The DTLS state machine is used to generate and process messages at different steps of the handshake as well as exchange encrypted data after the handshake. A 64-bit counter is used to implement the DTLS *Retransmission Timer* [3] which handles dropped packets. The time-out value can be configured externally, and the state machine re-transmits the previous flight whenever the timer expires. When the DTLS state machine waits for the next flight, all cryptographic accelerators are clock-gated in order to reduce power consumption. Three 256-byte FIFOs are used to fetch input messages (IN FIFO), send output messages (OUT FIFO) and read application data packets (DATA FIFO). The IN FIFO ensures that the DTLS controller starts parsing input messages only when a fully formed packet is available, and sends out complete output messages to the OUT FIFO. For encrypted application data, the state machine also implements the packet optimizations proposed in [4], with the option to enable AES-GCM tag truncation.

VI. Measurement Results

The test chip, shown in Fig. 19, was fabricated in a 65 nm CMOS process, with a core size of 1.54 × 1.54 mm^2. The RISC-V processor occupies 0.0489 mm^2 (34 kGE) area and interfaces with 16 KB instruction cache and 64 KB data cache. The DTLS engine requires 0.214 mm^2 (149k GE) logic area, and uses 6.75 KB of SRAM for the comb point cache, DTLS RAM and packet FIFOs. The chip supports voltage scaling from 1.2 V down to 0.8 V. The RISC-V core achieves a maximum frequency of 78 MHz at 1.2 V and 20 MHz at 0.8 V. The DTLS engine can operate in a frequency range of 16 MHz (at 0.8 V) to 20 MHz (at 1.2 V). All measurements for the RISC-V processor and the DTLS engine are reported at 16 MHz and 0.8 V.

Fig. 20 shows our test board and measurement setup. The test chip is housed in a QFN64 socket soldered to the board,



Chip Specifications	
Technology	65 nm LP CMOS
Supply voltage	0.8 – 1.2 V
Die size	2 mm x 2 mm
DTLS Engine	
Logic gates	149k (NAND2 equiv.)
SRAM	6.75 KB
Max. frequency	16 MHz at 0.8 V 20 MHz at 1.2 V
DTLS Energy	44.08 μJ (Handshake) 0.89 nJ/B (App. Data)
RISC-V Processor	
Logic gates	34k (NAND2 equiv.)
SRAM	16 KB Instr. Cache 64 KB Data Mem.
Max. frequency	20 MHz at 0.8 V 78 MHz at 1.2 V
Dhrystone	0.96 DMIPS/MHz
Dhrystone Energy	40.36 μW/MHz

Fig. 19. Chip micrograph, logic area breakdown of the DTLS engine and summary of chip specifications.

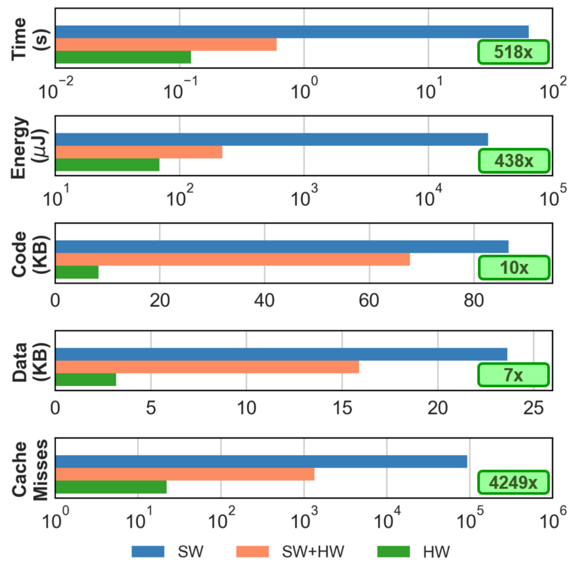


Fig. 21. Processor resource utilization in three different DTLS handshake implementations – SW, SW+HW and HW. Improvements of HW versus SW are indicated in the boxes.

and an Opal Kelly XEM7001 FPGA is used to interface with the chip. A Keithley 2602A source meter is used to supply power to the chip. Both the FPGA and the source meter are controlled from a host computer through USB and GPIB interfaces respectively. While our chip has an SD interface which can communicate with standard SD cards, we use the FPGA to emulate the SD card program memory so that we can eliminate the overhead otherwise imposed by real SD card access times and thus allow fair software benchmarking.

A. Protocol Benchmarks and Energy Measurements

The DTLS engine supports handshake in two modes:

- Full – with verification of server certificate
- Cached – with caching of server certificate information to speed up future handshakes

The cached mode requires one less ECDSA-Verify operation, thus achieving 36% reduction in handshake time and energy. Energy consumption of the hardware-accelerated DTLS handshake is 68.94 μJ and 44.08 μJ in the full and cached modes respectively. In the application data phase, the chip consumes 0.89 nJ per byte of data.

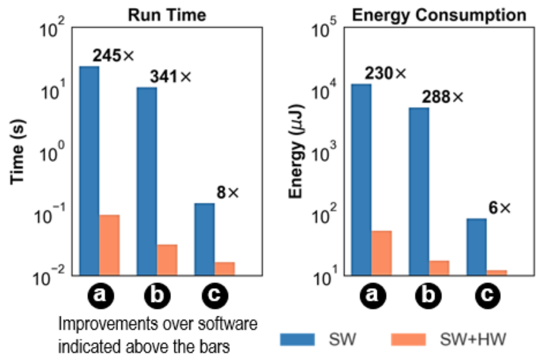


Fig. 22. Benchmarks for security protocols implemented in SW and SW+HW – (a) ECMQV, (b) Schnorr Prover and (c) Merkle Hashing. Improvements over software are indicated above the bars.

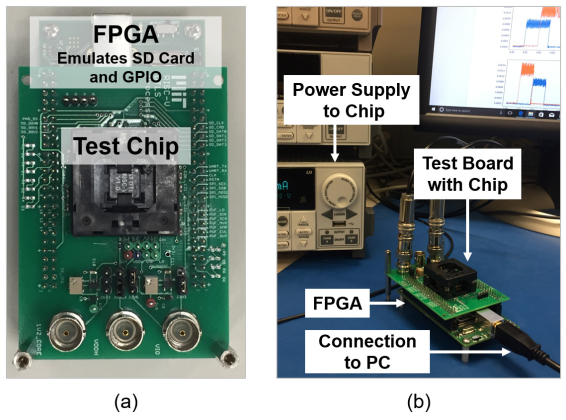


Fig. 20. (a) Test board with FPGA and (b) power measurement setup.

In order to analyze the efficiency of our DTLS hardware accelerator, we compared resource utilization in three scenarios: DTLS fully implemented as RISC-V software (SW), the cryptographic kernels accelerated in hardware and only the DTLS controller implemented in software (SW+HW), and DTLS fully implemented in hardware (HW). Test software was implemented using the cryptographic libraries provided by ARM mbedTLS [36]. Since mbedTLS does not support cached server certificates, all analyses were performed with the DE in non-cached mode. Detailed comparisons are shown in Fig. 21. The use of cryptographic accelerators alone results in over 2 orders of magnitude improvement in run time and energy efficiency (SW vs. SW+HW). The hardware DTLS controller reduces code size by 60 KB, while the DTLS micro stack results in 13 KB reduction in data memory usage (SW+HW vs. HW). When DTLS is accelerated in hardware, code size goes down to only 8 KB, including system functions. We also note that the area occupied by the DTLS state machine and control logic is 5× smaller than the area of SRAM otherwise required to accommodate the DTLS program in software.

Security applications beyond DTLS can also be implemented on the RISC-V, using the cryptographic accelerators in standalone mode. We illustrate this flexibility using three benchmark applications – (a) ECMQV, an alternative to ECDHE/ECDSA-based authenticated key exchange, (b) Schnorr Prover, an interactive zero-knowledge prover of identity, and (c) Merkle Hashing, used to ensure data integrity in peer-to-peer network protocols. The reduction in resource utilization for all three applications is shown in Fig. 22. The ECC-based applications achieve over 200× increase in energy-efficiency, while Merkle hashing sees 6× energy savings.

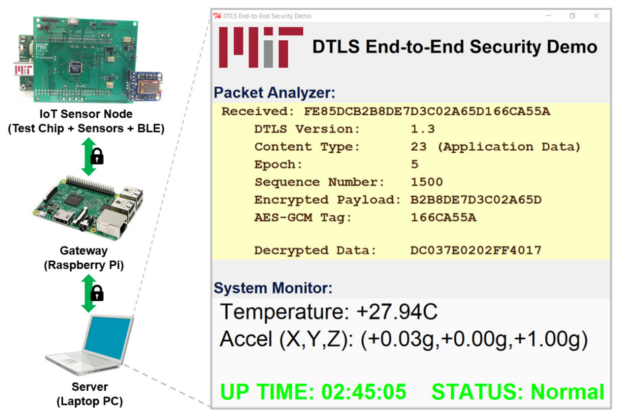
Since the cryptographic hardware is active only for a fraction of the total processing time of the IoT node, it is important to analyze the effect of leakage power of the DE on overall energy consumption. Overall leakage power of the chip was measured around 30 μW at 0.8 V, out of which the DE accounts for $\approx$ 60% (based on layout area). Power consumption of the chip, with only the RISC-V processor active and the DE clock-gated, ranges between 300-650 μW at 16 MHz and 0.8 V depending on the application software being executed. Therefore, DE leakage power is at most 6% of the total power consumption of the IoT application, thus justifying some of the architectural optimizations described in Section IV which increase leakage power due to larger logic area. The peak current drawn by the chip at 16 MHz and 0.8 V was measured to be around 800-900 μA, which is well below the maximum current supplied by standard batteries, e.g., 15/30 mA for coin cells.

B. System Demonstration

To demonstrate the functionality of our chip in a complete system, a secure IoT node was designed with the test chip collecting data from a temperature sensor and an accelerometer, encrypting it and then transmitting it through a Bluetooth Low Energy (BLE) transceiver, where all data communications with our test chip are through SPI. A Raspberry Pi module is used as a gateway which forwards these encrypted packets to the application software running on a PC. The system setup is shown in Fig. 23, along with a screenshot of the server application which displays packet details along with decrypted sensor data.

C. Comparison with Previous Work

Fig. 24 compares this work with embedded systems that integrate multiple cryptographic accelerators. This work implements a flexible ECC accelerator which supports arbitrary primes up to 256 bits, in contrast with [23] and [26] which only support fixed 192 and 255-bit curves respectively. [37] only supports binary field modular arithmetic in hardware. Our ECC accelerator is 458× and 9× more energy-efficient than [23] and [26] respectively at comparable security levels. In addition to the resource savings enabled by the individual cryptographic accelerators, offloading DTLS control flow to the DE realizes a further 3× reduction in energy and 5× reduction in run time compared to a SW+HW implementation.



Packet Analyzer:

```
Received: FE85DCB2B8DE7D3C02A65D166CA55A
    DTLS Version:       1.3
    Content Type:       23 (Application Data)
    Epoch:              5
    Sequence Number:    1500
    Encrypted Payload:  B2B8DE7D3C02A65D
    AES-GCM Tag:        166CA55A

    Decrypted Data:     DC037E0202FF4017
```

System Monitor:

Temperature: +27.94C
Accel (X,Y,Z): (+0.03g,+0.00g,+1.00g)

UP TIME: 02:45:05 STATUS: Normal

Fig. 23. System demonstration of a secure IoT node with our test chip collecting data from sensors and transmitting them to a server application over a DTLS-encrypted channel.

Specifications	WISTP'11 [23] [a]	CHES'15 [26] [a]	VLSIC'17 [37]	**This work**
Technology	350 nm	130 nm	40 nm	65 nm
Supply voltage (V)	3.3	1.2	0.7	0.8
Frequency (MHz)	0.847	1	28.8	16
App. Processor	–	–	ARM Cortex M0	RISC-V RV32I
Cryptographic Accelerator				
Logic gates	12.8k	32.6k	–	149k
SRAM	0.25 KB	0.28 KB	8 KB	6.75 KB
Hardware ECSM support	Only NIST P-192	Only Curve25519	–[b]	**All prime curves up to 256 bits**
Base point ECSM energy (μJ)	1423.6 (192 bit)	–	–[b]	**3.11** (192 bit)
	–	56.8 (255 bit)		**6.34** (256 bit)
AES energy (nJ)	8558.04	521.01 [c]	7.05	6.21
SHA energy (nJ)	6876.3 [d]	–	48.7 [d]	24.3 [d]
DTLS in hardware	No	No	No	**Yes**
DTLS energy	–	–	–	**44.08 μJ** (Handshake) **0.89 nJ/B** (App. Data)

[a] Post-synthesis data reported for [23] and [26]
[b] [37] implements only modular multiplication in binary fields in hardware
[c] [26] implements Salsa20 instead of AES for encryption
[d] [23] implements SHA-1; [37] implements SHA-3; This work implements SHA-2

Fig. 24. Comparison of our design with integrated cryptographic accelerators for embedded systems.

VII. Conclusion

In this work, we have presented an energy-efficient reconfigurable cryptographic engine which makes DTLS a practical solution for implementing end-to-end security on resource-constrained IoT devices. Energy-efficient accelerators for ECC, AES and SHA provide more than two orders of magnitude improvement in performance and energy-efficiency compared to software implementations of DTLS. This allows IoT sensor nodes to re-authenticate more frequently for applications that demand stronger security guarantees. A dedicated DTLS 1.3 protocol controller enables 78 KB and 20 KB reduction in code and memory usage respectively. This allows IoT platforms to implement application programs without having to worry about the overheads otherwise imposed by the security protocol. Protocols beyond DTLS can also be implemented using the RISC-V processor working in conjunction with the cryptographic accelerators, while still getting the benefits of energy-efficiency and performance.

Acknowledgment

The authors would like to thank the Qualcomm Innovation Fellowship and Texas Instruments for funding this work, the TSMC University Shuttle Program for chip fabrication support, and Bluespec, Cadence and Mentor Graphics for providing CAD tools. They would also like to thank Mehul Tikekar, Priyanka Raina and Phillip Nadeau for their support during chip tape-out and testing.

References

[1] 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.

[2] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," *IETF RFC*, vol. 8446, August 2018.

[3] E. Rescorla, H. Tschofenig and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3," *IETF Internet-Draft*, Draft 18, November 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-dtls13-18

[4] U. Banerjee et al., "eeDTLS: Energy-Efficient Datagram Transport Layer Security for the Internet of Things," *IEEE Global Communications Conference (GLOBECOM)*, pp. 1-6, December 2017.

[5] U. Banerjee et al., "An Energy-Efficient Reconfigurable DTLS Cryptographic Engine for End-to-End Security in IoT Applications," *IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 42-44, February 2018.

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

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

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

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

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

[11] A. Waterman et al., "The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.0," *Technical Report, EECS Department, University of California, Berkeley*, UCB/EECS-2014-54, May 2014.

[12] K. Asanovic et al., "The Rocket Chip Generator", *Technical Report, EECS Department, University of California, Berkeley*, UCB/EECS-2016-17, April 2016.

[13] SD Card Association , "Physical Layer Simplified Specification," *SD Simplified Specifications*, Part 1, Version 6.00, April 2017.

[14] ARM Holdings, *Arm Cortex-M Series Processors*. [Online]. Available: https://developer.arm.com/products/processors/cortex-m

[15] C. Duran et al., "A System-on-Chip Platform for the Internet of Things featuring a 32-bit RISC-V based Microcontroller," *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pp. 1-4, February 2017.

[16] R. Uytterhoeven and W. Dehaene, "A sub 10 pJ/Cycle Over a 2 to 200 MHz Performance Range RISC-V Microprocessor in 28 nm FDSOI," *IEEE European Solid State Circuits Conference (ESSCIRC)*, pp. 236-239, September 2018.

[17] U. Banerjee, "Energy-Efficient Protocols and Hardware Architectures for Transport Layer Security," *S.M. Thesis, Massachusetts Institute of Technology*, June 2017.

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

[19] 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)*, pp. 577-583, September 2006.

[20] S. K. Mathew et al., "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.

[21] S. Mathew et al., "340 mV1.1 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.

[22] Y. Zhang et al., "A Compact 446 Gbps/W AES Accelerator for Mobile SoC and IoT in 40nm," *IEEE Symposium on VLSI Circuits*, pp. 1-2, June 2016.

[23] 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.

[24] S. S. Roy et al., "Designing Tiny ECCProcessor," *Workshop on Elliptic Curve Cryptography – ECC 2013*, September 2013.

[25] 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.

[26] M. Hutter et al., "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.

[27] J. Guajardo et al., "Efficient Hardware Implementation of Finite Fields with Applications to Cryptography," *Acta Applicandae Mathematica*, vol. 93, no. 1, pp. 75118, September 2006.

[28] 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.

[29] M. Joye and C. Tymen, "Compact Encoding of Non-adjacent Forms with Applications to Elliptic Curve Cryptography," in *Public Key Cryptography – PKC 2001*, *Lecture Notes in Computer Science*, vol. 1992, pp. 353-364, February 2001.

[30] J. Fan et al., "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.

[31] 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.

[32] H. Krawczyk, M. Bellare and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," *IETF RFC*, vol. 2104, February 1997.

[33] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," *IETF RFC*, vol. 5869, May 2010.

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

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

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

[37] Y. Zhang et al., "Recryptor: A Reconfigurable In-Memory Cryptographic Cortex-M0 Processor for IoT," *IEEE Symposium on VLSI Circuits*, pp. C264-C265, June 2017.