NaCl's `Crypto_box` in Hardware

Michael Hutter[1](✉), Jürgen Schilling[2], Peter Schwabe[3](✉), and Wolfgang Wieser[2]

[1] Rambus Cryptography Research Division, 425 Market Street, 11th Floor, San Francisco, CA 94105, USA
`michael.hutter@cryptography.com`

[2] Graz University of Technology, IAIK, Inffeldgasse 16a, 8010 Graz, Austria
`j.schilling@student.tugraz.at, w.wies3r@gmail.com`

[3] Radboud University, Digital Security Group, PO Box 9010, 6500 Nijmegen, The Netherlands
`peter@cryptojedi.org`

Abstract. This paper presents a low-resource hardware implementation of the widely used `crypto_box` function of the Networking and Cryptography library (NaCl). It supports the X25519 Diffie-Hellman key exchange using Curve25519, the Salsa20 stream cipher, and the Poly1305 message authenticator. Our targeted application is a secure communication between devices in the Internet of Things (IoT) and Internet servers. Such devices are highly resource-constrained and require carefully optimized hardware implementations. We propose the first solution that enables 128-bit-secure public-key authenticated encryption on passively-powered IoT devices like WISP nodes. From a cryptographic point of view we thus make a first step to turn these devices into fully-fledged participants of Internet communication. Our crypto processor needs a silicon area of 14.6 kGEs and less than 40 μW of power at 1 MHz for a 130 nm low-leakage CMOS process technology.

Keywords: Internet of things · ASIC · Salsa20 · Poly1305 · Curve25519

1 Introduction

We need to empower computers with their own means of gathering information, so they can see, hear and smell the world for themselves, in all its random glory. RFID and sensor technology enable computers to observe, identify and understand the world—without the limitations of human-entered data. —Kevin Ashton, June 2009 [2]

This work was done while Michael Hutter was with Graz University of Technology. This work was supported by the Austrian Science Fund (FWF) under the grant number TRP251-N23 and by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114 Permanent ID of this document: `34306665f6a71562852503d8aa28f0f9`. Date: June 16, 2015.

In 1999, Ashton coined the term of the "Internet of Things" (IoT) for a network of sensors that communicate data over the Internet and thus give computers a way of sensing the world. Since then, various technological advances have brought us closer to turning this vision into reality and large companies are working on implementing these technologies on large scale. For example, Hewlett Packard's "Central Nervous System for the Earth (CeNSE)" project aims at implementing "a highly intelligent network of billions of nanoscale sensors designed to feel, taste, smell, see, and hear what is going on in the world" [10].

A representative platform for implementing sensors is called Wireless Identification and Sensing Platform (WISP), first proposed by Sample, Yeager, Powledge and Smith in [31]. WISP nodes are passively powered, wireless computing and sensing devices that communicate data to an ultra-high-frequency (UHF) RFID reader.

Most applications of this Internet of Things are safety-critical, security-critical, or involve processing private data. For example, HP lists as applications of CeNSE "roads, buildings, bridges, and other infrastructures; machines such as those used in airplanes and manufacturing plants; and organizations that work on health and safety issues, such as the contamination of food and water, disease control, and patient monitoring". Communicating such sensitive data over the Internet obviously raises security concerns. From a cryptographic point of view, the ideal solution is to communicate all data end-to-end authenticated and encrypted between a WISP node and the server processing the data. Key management for secret-key cryptography does not scale well, so authentication of billions of sensor nodes distributed over the whole planet calls for public-key cryptography.

For the communication of servers and large client computers such as desktop computers, laptops, and smartphones there exist various cryptographic libraries and frameworks that set up such end-to-end public-key authenticated encryption. However the question remains whether at least one of these frameworks is compatible with the highly restricted computational capabilities of WISP nodes.

This paper answers this question positively. More specifically, we present a carefully optimized hardware architecture of the basic primitives of the open Networking and Cryptography library (NaCl) [8]. In particular, we ported the `crypto_box` primitive into hardware including the X25519 elliptic-curve Diffie-Hellman key exchange [4], the Salsa20 stream cipher [5], and the Poly1305 secret-key one-time authenticator [3]. Our design is able to run a 128-bit-secure public-key authenticated encryption at the following hardware performance: Our smallest design of all primitives requires 14.6 kGEs of silicon area and is able to establish a secure Internet connection within 1.7 seconds when clocked at 4 MHz. Our fastest design needs 18 kGEs and performs public-key authenticated encryption within about 400 ms. Our crypto-processor can be used and integrated into WISPs and related low-resource sensor nodes. Optimizations of other mandatory system components such as random-number generation, analog front-end and protocol handling for RF communication, or non-volatile memory are not covered in this paper and need further investigation.

The application described above defines the optimization goals we aimed at in both choices of primitives and implementation as follows:

Low power, not low energy. WISPs harvest the power that is emitted by an RF-signal-emitting reader device. The maximal power available to the core is usually only a few tens of microwatts per megahertz. As the device is not battery-powered, energy consumption is a minor concern.

Compatibility with Internet crypto. We want our solution to be easy to integrate with existing crypto used for Internet communication. A somewhat heavy-weight choice would be SSL/TLS with a suitable selection of primitives [22]. We instead decided to go with the approach taken by NaCl, which is easier to integrate and very efficient in software. This paper shows that it can also be very efficient in hardware.

No need for signatures. There are two main differences between public-key authentication as used by NaCl's `crypto_box` and cryptographic signatures. One difference is that the authenticating parties have to be online at the same time to establish a key via static Diffie-Hellman, which prohibits a delegation of trust to offline parties. In this setting, we assume that the WISP node knows that the public key of the server is authentic which is a reasonable assumption in the IoT scenario where WISPs transmit sensed data to a specific (trusted) server in the Internet. Furthermore, the service of non-repudiation is not essentially necessary in this context, we can therefore avoid the overheads of implementing digital signatures.

Small and fast public-key authenticated encryption. The aim of the proposed design is not to optimize a single primitive, but to obtain small area and reasonable speed for a *combination of primitives* for public-key authenticated encryption. For example, a standalone hardware implementation of AES (or even a lightweight cipher such as PRESENT [9]) would be much smaller than a standalone implementation of Salsa20. Similarly, AES-GCM would be more efficient than Salsa20+Poly1305 for secret-key authenticated encryption. However, the public-key part needs arithmetic on big integers, which we compose of arithmetic on 32-bit integers. This approach gives us all the building blocks we need for Salsa20 and Poly1305 almost for free in terms of silicon area.

We believe that it is possible to achieve even smaller area if we resorted to a binary elliptic curve for Diffie-Hellman and combined this with, for example, AES-GCM. Obviously it is possible to obtain even better efficiency when reducing the security to, for example, 80 bits. However, the central contribution of this paper is to show that we can achieve 128-bit secure public-key authenticated encryption with a conservative choice of primitives on low-resource WISP nodes having a very small footprint in terms of area and power.

Nonce generation. NaCl's `crypto_box` receives as one input a public nonce. This paper does not discuss how this nonce is generated; for a discussion of how nonces are integrated into higher-level protocols, see [8, Sec. 2].

Related Work. The cryptographic primitives used in NaCl for the `crypto_box` public-key authenticated encryption have been designed for high software performance. Consequently, the primitives have so far mainly been implemented in software. Most of this software is included in the eBACS SUPERCOP benchmarking framework [7]. There also exist optimized implementations for embedded microcontrollers (e.g., AVRs, MSP430, or ARM Cortex M) which are not supported by SUPERCOP. Examples are given in [11,17,19,27]

The focus on software implementations does not mean that there exist no implementations in hardware. In particular for the Salsa20 stream cipher there exist various hardware implementations with different optimization targets. For example, there are FPGA implementations [12,24,34] and also ASIC designs [13,16,38]. The only hardware implementation of Curve25519 that we are aware of is the throughput-optimized FPGA implementation by Sasdrich and Güneysu, which achieves a throughput of more than $32,000$ scalar multiplications per second on a Xilinx Zynq 7020 FPGA [32]. Besides Curve25519 however there exist a broad range of other elliptic-curve implementations over $\mathbb{F}_p$. Most of them perform scalar multiplication on Weierstrass curves and target various FPGA platforms. Typical examples—that have a similar security level as Curve25519—are given in [1,14,15,25,26,28,30,35].

We are not aware of any hardware implementations of Poly1305.

Notation. In [4], Bernstein introduced a high-security high-performance elliptic-curve Diffie-Hellman key-exchange scheme called Curve25519. The name originally referred to the complete scheme, but was later also used to refer to the specific elliptic curve used in this scheme. To eliminate possible confusion, Bernstein recently suggested to use the term *X25519* for the "recommended Montgomery-X-coordinate DH function" and the term *Curve25519* for the underlying elliptic curve. We adopt this new terminology in this paper.

Availability of Results. We will make all results described in this paper available online. In particular we will place the HDL implementation described in this paper into the public domain to maximize reusability of our results[1]. The entire implementation avoids all patents that the authors are aware of.

Roadmap. The paper is organized as follows. In Sect. 2, we will give a short introduction into NaCl and its underlying cryptographic primitives. Section 3 presents our processor and describes the hardware architecture and all its implemented components. In Sect. 4, we describe the machine-code implementation including X25519, Salsa20, and Poly1305. Finally, results are given in Sect. 5.

2 Preliminaries – The `Crypto_box` Function

The Networking and Cryptography library (short: NaCl), developed by Bernstein, Lange, and Schwabe, advertises a "simple high-level API" [8]. The core functionality of this API is the `crypto_box` function, which offers public-key authenticated encryption. It computes an authenticated ciphertext from a message, a nonce, the

[1] The source code is available at http://mhutter.org/research/vlsi/#naclhw and at http://cryptojedi.org/crypto/#naclhw.

sender's private key, and the receiver's public key. The receiver feeds this authenticated ciphertext together with the nonce, his private key, and the sender's public key into the `crypto_box_open` function to verify the authentication tag and recover the message.

In principle, NaCl supports different independent implementations of this function with different underlying primitives. However, the default construction used in NaCl (and targeted in previous NaCl optimization papers) is a construction based on the X25519 elliptic-curve Diffie-Hellman (ECDH) key exchange, the XSalsa20 stream cipher, and the Poly1305 secret-key one-time authenticator. We briefly review these three primitives in the following subsections.

Curve25519 and the X25519 Function. In 2006, Bernstein proposed the X25519 ECDH scheme [4]. The scheme is based on arithmetic on the Montgomery curve "Curve25519" with equation $E : y^2 = x^3 + 486662x^2 + x$ defined over the field $\mathbb{F}_{2^{255}-19}$. This curve was chosen for high security and high performance. For details about the security properties see [4, Sec. 3].

X25519 secret keys are byte arrays of length 32. Inside X25519, such a byte array is interpreted as a little-endian-encoded 256-bit integer. Before this integer is used as a scalar in elliptic-curve scalar multiplication, the most significant bit is set to 0, the second-most significant bit is set to 1, and the three least significant bits are set to 0. X25519 public keys are also byte arrays of length 32, and encode the x-coordinate of a point on Curve25519.

X25519 uses the fast x-coordinate-only differential-addition chain proposed by Montgomery in [29] to compute a shared secret kP from a secret key k and a public key P. Key-pair generation uses the same computation with a fixed basepoint. The computation involves 255 "ladder steps" and one final inversion in $\mathbb{F}_{2^{255}-19}$. Each ladder step involves 5 multiplications, 4 squarings, one multiplication by the constant $(486662 + 2)/4$, and some additions and subtractions in $\mathbb{F}_{2^{255}-19}$. The final inversion can be computed as exponentiation with $\mathbb{F}_{2^{255}-21}$. An efficient addition chain for this exponentiation proposed in [4] takes 254 squarings and 11 multiplications.

The XSalsa20 Stream Cipher. The Salsa20 stream cipher is an eSTREAM finalist designed by Bernstein [5] and performes 20 rounds on an internal state. The XSalsa20 stream cipher was introduced by Bernstein in [6]. It uses the same core as Salsa20, but supports a 192-bit nonce instead of the Salsa20 64-bit nonce. This is achieved by a fast nonce-setup, called HSalsa20, followed by Salsa20 keystream generation. This combination is denoted as XSalsa20. For details see [6, Sec. 2].

The computations inside HSalsa20 and Salsa20 are very similar, in particular they both use the same 20-round transformation on blocks of 64 bytes. A block is treated as a 4×4-matrix of 32-bit words. Each of the 20 rounds consists of 16 add-rotate-xor sequences, such as

```
s4 = x0 + x12
x4 ^= (s4 >>> 25)
```

The main difference between HSalsa20 and Salsa20 is that they use a different finalization computation to produce a 64-byte output block in the case of Salsa20 and a 32-byte output block in the case of HSalsa20.

Poly1305 Secret-Key Authentication. The Poly1305 authenticator was introduced by Bernstein in [3]. The security of this authenticator requires that a key is used only once; inside `crypto_box` this is ensured by prepending a 32-byte zero-block in front of the message and then using the XSalsa20 encryption of this zero block as authentication key. The computations in Poly1305 are based on arithmetic in the finite field $\mathbb{F}_{2^{130}-5}$. The authentication tag is computed by processing the input in 16-byte blocks. For each block, Poly1305 treats the block as an element of $\mathbb{F}_{2^{130}-5}$, adds this element into a state, and multiplies the state by a secret value $r \in \mathbb{F}_{2^{130}-5}$, which is essentially the first half of the 32-byte secret key with some bits set to zero. After the whole message has been processed, the authentication tag is computed as addition of the state with the second half of the 32-byte secret key.

3 A `Crypto_box` Specific Instruction-Set Processor

Implementing public-key authenticated encryption in hardware is a challenging task and requires many different design decisions. Since we aim for a very efficient architecture in terms of low-resource requirements, we decided to implement an Application Specific Instruction Set Processor (ASIP). In contrast to microprocessors, ASIPs are usually less flexible because there might be no compiler support for the custom processor. The machine code needs to be implemented "by-hand" or by self-written compilers that support the optimized instruction set. On the other hand, ASIPs can benefit in terms of efficiency, i.e., higher speeds and lower area and power requirements. Basically, the main features of our design can be summarized as follows:

Resource efficiency. Our processor was designed with resource efficiency in mind. This means that we aimed for a low-area architecture that re-uses resources as much as possible. Our hardware components such as the implemented hardware multiplier have been chosen to require only a low number of logic gates while providing appropriate speeds.

Platform independency. Our design does not make use of any technology-specific components. It is therefore flexible in the sense that it can be synthesized on different CMOS-process technologies and FPGAs.

Security. All implemented cryptographic primitives share a security level of 128 bits. Furthermore, we avoided to use any secret branch conditions in our implementation and guarantee that all operations run in constant time. We therefore offer a baseline implementation that can be used to compare with related work and that can be extended in future to integrate hiding and masking countermeasures that offer protections against DPA and correlation-collision attacks etc.

Compatibility with the NaCl API. We are fully compatible to the existing software NaCl interface which offers easy integration of the processor in existing infrastructures and applications.

Support of efficient primitives. Our processor supports a set of high-level primitives (that originally have been designed to achieve high speeds) to offer a range of demanded cryptographic services. We support the following functions:

1. Establishing secure session keys using the X25519 Diffie-Hellman key exchange [4] by running `crypto_dh_curve25519`.
2. Data encryption and decryption using XSalsa20 [5,6].
3. Message authentication using Poly1305 [3].
4. Public-key authenticated encryption by executing the `crypto_box` function. The server can then verify the message authenticity by calling `crypto_box_open` of the NaCl API.
5. Verification and decryption of NaCl authenticated ciphertexts for secure transmission of control messages.

3.1 Hardware Implementation Overview

Our design mainly consists of a memory unit, a controller, and a `crypto_box`-specific Arithmetic Logic Unit (ALU). The ASIP can be accessed through a 32-bit interface.

Memory is one of the most critical components in efficient hardware designs. Especially in implementations of public-key cryptography, it often takes up to 80 % of the entire circuit area and also consumes a significant amount of power. We therefore decided to implement a memory with very generic elements, which can be efficiently replaced by highly optimized platform-dependent memory technologies. For volatile memory, we decided to implement a random-access memory (RAM) instead of a register file. Our design thus supports efficient RAM macros for specific CMOS process technologies. We also stored all constants regularly in a read-only memory (ROM) table. This allows a better optimization by the hardware synthesizer and also enables the use of more efficient ROM macros.

We decided to implement a 32-bit single-port RAM. Single-port memories have the advantage that they are usually smaller in size compared to multi-port memories. This fact makes them attractive for implementations running in resource-constrained environments. Essentially, there are two main reasons for their smaller footprint: (1) each memory cell is basically composed of 6 transistors while 8 transistors are usually required for dual-port memories; and (2) the additional address logic and read/write drivers of multiple ports cause an additional increase in required resources. For example, Faraday Technology Corporation offers a synchronous dual-port register-file RAM (with $32 \times 64 = 2048$ bits) that requires $0.035\,\text{mm}^2$ while a synchronous single-port register-file RAM of the same company requires only $0.023\,\text{mm}^2$ on a low-leakage 130 nm CMOS-process technology. Generally, one-port memories are about 1.4 to 1.7 times smaller than dual-port memories.

The main drawback of single-port RAMs however is *throughput*. While dual-port memories can simultaneously read and/or write two words at different addresses, single-port memories can access only one address per clock cycle. This is the reason that most of (not only high-speed but also resource-constrained) ECC implementations use dual-port memories to keep the arithmetic unit busy in each clock cycle. In particular, multi-precision multiplication (which is often the efficiency bottleneck of those implementations) requires that many partial products are calculated, each needing two operands in each clock cycle.

Optimized Single-Port Memory Arithmetic. We address the issue of low memory throughput and apply a method that allows us to keep the arithmetic unit busy despite the low throughput of single-port memories. More specifically, we use product-scanning-based multiplication but process two columns in parallel. In each clock cycle, two operands from two columns are chosen while one operand is kept in a 32-bit register and re-used in the next cycle. This allows to perform one 32×32-bit multiplication in each cycle.

Selective Memory-Bank Access. Both RAM and ROM are logically divided into a set of memory banks with a data width of 256 bits each. The RAM is composed of 9 memory banks: 1×256 bits are needed for storing the x-coordinate of the (fixed or random) base point, 1×256 bits are needed to store the private key of the X25519 Diffie-Hellman key exchange, and 7×256 bits are required for scalar multiplication. The ROM is composed of 6 memory banks, which contain constants for modular reductions in $\mathbb{F}_{2^{255}-19}$ for Curve25519 and $\mathbb{F}_{2^{130}-5}$ for Poly1305, 2 logic masks, the curve parameter $a24$, and σ for XSalsa20.

We restricted access to the memory from the I/O interface and only allow read/write to RAM banks with index 0 and 1. All other memory banks are not accessible from outside.

Secret-Key-Dependent Memory-Bank Switching. Curve25519 uses the Montgomery powering ladder [20,29] as scalar-multiplication method. In each ladder-step iteration, two curve-point coordinates need to be swapped depending on a secret scalar bit. To avoid secret-key dependent branch conditions, we implemented an additional memory logic that conditionally swaps the addresses of two memory banks depending on a single bit in constant time. This avoids a secret-key dependent branch condition in software and makes a swap-function implementation unnecessary.

3.2 The Controller

The controller is composed of two program ROMs, a program-counter, an instruction decoder, a dedicated multiplication controller, and a flattened memory-management unit including address decoding and page-table control.

We decided to implement two distinct program ROMs: one that contains the program code for Curve25519 and one that mainly contains the code for XSalsa20 and Poly1305. During implementation, it has been shown that both program ROMs have nearly the same code size (the ROM for Curve25519 is slightly smaller in size so that we padded the remaining bytes with zero to obtain equally sized

ROMs). Splitting the ROMs has the advantage that (1) the critical path is reduced due to smaller tables, (2) the area is reduced since modern hardware synthesizers can apply better optimizations, and (3) splitting allows to effectively "isolate" one ROM to reduce power consumption while the other ROM is active.

The Special-Purpose Instruction Set. We implemented 46 instructions, out of which 26 instructions are general-purpose instructions and 20 are special instructions tailored for efficient NaCl `crypto_box` computations. From these 46 instructions, there are 6 program-flow instructions that for example allow the use of subroutine calls to reduce code size. Almost all instructions directly load operands from or store to memory, which improves performance and avoids the need of (costly) CPU registers.

`Crypto_box`-Optimized Memory Paging. We further applied the following optimization in order to reduce the memory-instruction width and necessary opcode size, respectively. By analyzing our `crypto_box` implementation, we identified that in most cases only access to a limited amount of memory banks is necessary (especially in subroutines). This is why we decided to implement a lightweight memory-management unit that makes use of a *memory-paging technique* in order to reduce the length of the total opcode. Thus, we reserve only 2 bits for memory-page addressing. A page consists of 4 non-contiguous memory banks that are pre-determined and statically stored in a page-address table in ROM. During execution, only 1 page can be concurrently active and instructions can access only the 4 memory banks of this page. A memory page can be selected using the Memory Page Select (`MPS`) instruction, for subroutines we implemented the dedicated Memory Page Increment (`MPI`) and Memory Page Decrement (`MPD`) instruction that both increment/decrement the page index.

By applying these enhancements, only 9 bits of opcode are needed for all instructions. From these 9 bits, up to 5 bits are used for memory addressing purposes: two bits are needed to select one memory-bank from the active page and three bits are needed to select a single 32-bit word from the virtual 256-bit memory bank.

Single-Level Subroutines and Parameter Passing. Our memory-paging technique also enables easy address-parameter passing to subroutines. By simply selecting different pages, subroutines can operate on different memory banks without additional lines of code and without additional registers to store the parameters. To enable single-level subroutines, we integrated an 11-bit register that holds the return address. An additional multiplexer is used to update the program counter after returning from the subroutine. To efficiently address the subroutines in the program ROM, we implemented an address decoder using a ROM lookup-table.

Improving Speed by Operand Prefetching. In order to increase the speed of our ASIP design, we apply *operand prefetching*. This allows that instructions can already preset the address of an operand that is needed in the subsequent instruction. Since loading from RAM generally requires two cycles, this improves performance by simply "prefetching" an operand during execution of the previous instruction.

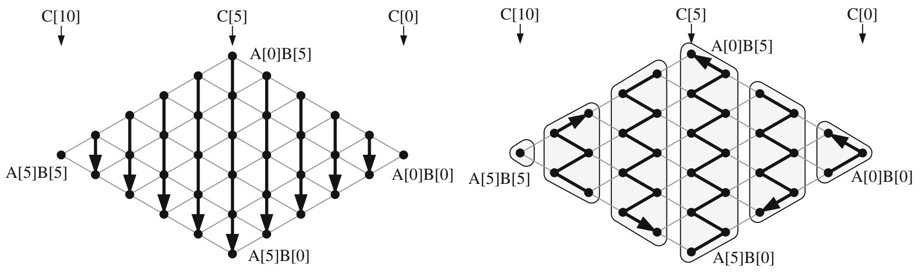

Fig. 1. Column-wise product-scanning multiplication (left) and 2-column parallel product-scanning multiplication (right).

To store prefetched operands, we integrated a dedicated 32-bit *prefetch buffer*. This buffer is located right at the input of the Arithmetic Logic Unit (ALU). The ALU can then either load the prefetched operand from the buffer or it can directly load constants from ROM within a single clock cycle.

2-Column Product-Scanning Hardware Multiplier Control. The performance of Curve25519 and Poly1305 largely depends on the performance of big-integer multiplication. We build this big-integer multiplication from 32 $\times$ 32-bit multiplications, but instead of implementing it in software, we integrated a dedicated controller that is used to provide fast multiplication based on product-scanning multiplication. Instead of processing all partial products column-wise (which is often done by executing a multiply-accumulate instruction on many platforms), we process two columns in parallel. The controller chooses operands from one and the other column alternately, therefore "hopping" between the columns as shown in Fig. 1. The figure shows an $A \times B = 6 \times 6$-word multiplication as an example, where all word multiplications $A[i] \times B[i]$ with index $i \in [0, 5]$ are represented as black dots and the processing is indicated with an arrow. The processing starts from right to left and the result C is the sum of all partial-product columns.

For three consecutive word multiplications, two operands are always the same because of the parallel processing of the two columns. This allows to *buffer* one operand in a 32-bit register while another operand can be prefetched from memory. The ALU is therefore busy in each clock cycle.

3.3 The Arithmetic Logic Unit

The ALU consists of a digit-serial multiplier, a rotation unit, a carry-handling logic, a prefetch buffer, and an accumulator register.

The core element of the ALU is a *digit-serial multiplier*. This type of multiplier processes several bits (so-called digits) in parallel in contrast to bit-serial multipliers which process one operand bit-by-bit only. They are therefore significantly faster while requiring less resources than single-cycle array multipliers (in terms of both area and power). We pay for these advantages with a longer critical path delay[2] that limits the maximum frequency of our design to a few MHz. For our targeted application range, this is not a problem.

We made our multiplier flexible such that the digit size can be configured before synthesis (parameter $w \in \{2, 4, 8, 12, 16\}$ defines parallelism). By changing this parameter we can trade higher speed for smaller area. The main components of our multiplier are a set of adders that are connected in series and that are shifted by one bit each. The output of the multiplier is then fed to an optimized *multi-operation logic* (MOL) that is able to perform both arithmetic as well as logic operations. Configured as an adder, it can be used to add the partial products of the digit-serial multiplier to the accumulator register. Besides this, it can be also configured to subtract operands (e.g., for modular reduction) or to perform an AND, OR, or XOR operation.

`Crypto_box`-Dedicated Rotating. To keep the area requirements low, we decided to implement a `crypto_box` dedicated rotation logic that is able to rotate the accumulator register by several fixed offsets that are required for implementing the `crypto_box` function. Also for the digit-serial multiplication, the accumulator register can be rotated by w bits to the right and it can be rotated back to further *multiply-and-accumulate* partial products without the need of additional registers to buffer the intermediate results (or to store or load them from memory). For Salsa20 it is necessary to rotate the accumulator by 7, 9, 13, and 18 bits to the right. To speed up the reduction in Poly1305, we also support rotation by 2 and 30 bits to the right.

For ECC scalar multiplication, one might expect an accumulator size of 67 bits: 2×32 bits for holding the words of the multiplication results, and 3 bits for storing carries. However, as we process two columns in parallel, it is necessary to extend the size of the accumulator by 32 bits to store the intermediate result. The accumulator has thus a size of 99 bits in total.

For efficient carry handling, we implemented a dedicated register that stores the carry bit. This is especially necessary for modular reduction where the prime modulus (loaded from ROM) is either added or subtracted depending on the carry/borrow of the underlying operation. To achieve constant runtime, we also load the prime modulus from ROM and perform an addition or subtraction operation in the case of already reduced results. In this case, however, we deactivate the addition/subtraction logic so that the unmodified number will be written back to memory (which equals to adding/subtracting zero).

4 Machine-Code Implementations

In this section, we summarize the implementation of Curve25519, XSalsa20, and Poly1305 using our special-purpose instruction set. In total, our implementation needs 9×256 bits of memory, i.e., 9 memory banks (further denoted by `R0`,...,`R8`). Only two memory banks are user-accessible to load and store input/output data, the remaining banks are read/write protected.

[2] Our multiplier is part of the critical path and constitutes 40–53 % of the total delay.

Table 1. The supported ASIP command set

Command	Hex	Description
DH-1	0x00	X25519 Diffie-Hellman key exchange: computes public key
DH-2	0x01	X25519 Diffie-Hellman key exchange: computes session key
INIT	0x02	HSalsa20: computes extended session key
FIRST	0x03	XSalsa20: computes first cipher block
UPDATE	0x04	XSalsa20: computes next cipher block
FINALIZE	0x05	Poly1305: computes authentication tag
DECRYPT	0x06	XSalsa20/Poly1305: decrypts and authenticates a single block

The general communication flow works as follows. Data can be sent via the 32-bit I/O interface. The input data (e.g., the plaintext) can be stored in either `R0` or `R1`. After that, a `crypto_box` operation can be started by sending one out of five supported commands listed in Table 1. In particular, we implemented a *busy-wait polling* mechanism to sample the status of the `crypto_box`-operation processing. If the busy flag is set, the device still performs operations; data (e.g., the ciphertext) can be accessed from `R0` or `R1` after the busy flag is cleared.

4.1 The X25519 Key Exchange

The first step in X25519 is to generate a public key. This can be done by our ASIP as follows. First, the private key needs to be written to `R0` and the base point of Curve25519 to `R1`. Second, by sending the `DH-1` command, the public key is calculated which can be retrieved from `R1` after busy-wait polling.

In the second step, the secret-key is established by exchanging the public-keys. For this purpose, the public-key of the opponent is written to `R1`, `R0` still holds the private key of the device and remains the same. After sending the `DH-2` command, a session-key is established and stored in the (read-protected) memory-bank `R6`.

Initialization. Our implementation of X25519 starts with an initialization phase were all memory banks are initialized. Some of the memory banks are initialized to zero or one; others are initialized to the x-coordinate of the point. In DH-1 this is the fixed base P, in DH-2 it is the public-key received from the communication partner. Additionally, it is necessary to apply masking operations to the 32-byte secret key (see Sect. 2). In total, the initialization for X25519 needs 77 instructions and 77 cycles.

Curve25519 Differential Addition-and-Doubling. We implemented the Montgomery powering ladder to perform scalar multiplication. Since Curve25519 is a curve in Montgomery form, it allows to perform efficient x-coordinate-only operations. To keep the memory-requirements as low as possible, we efficiently reordered the Montgomery formula [29] and provide an explicit formula requiring $5\mathsf{M} + 4\mathsf{S} + 8\mathsf{add} + 1\mathsf{M}_{a24}$ while needing only 6 working registers (plus the register

Listing 1. Differential addition-and-doubling on x-coord-only Montgomery curves using $5\mathsf{M} + 4\mathsf{S} + 8\mathsf{add} + 1\mathsf{M}_a$ and $6 + \{x_D\}$ registers and $a24 = (a + 2)/4$.

```
Require: $X_1$, $X_2$, $Z_1$, $Z_2$, $x_D$, $a24$
Ensure: $X_1$, $X_2$, $Z_1$, $Z_2$
1:  1. $R_1 \leftarrow X_2 + Z_2$
    2. $X_2 \leftarrow X_2 - Z_2$
    3. $Z_2 \leftarrow X_1 + Z_1$
    4. $X_1 \leftarrow X_1 - Z_1$
    5. $R_1 \leftarrow R_1 \times X_1$
    6. $X_2 \leftarrow X_2 \times Z_2$
    7. $Z_2 \leftarrow Z_2 \times Z_2$
    8. $X_1 \leftarrow X_1 \times X_1$
    9. $R_2 \leftarrow Z_2 - X_1$
    10. $Z_1 \leftarrow R_2 \times a24$
    11. $Z_1 \leftarrow Z_1 + X_1$
    12. $Z_1 \leftarrow R_2 \times Z_1$
    13. $X_1 \leftarrow Z_2 \times X_1$
    14. $Z_2 \leftarrow R_1 - X_2$
    15. $Z_2 \leftarrow Z_2 \times Z_2$
    16. $Z_2 \leftarrow Z_2 \times x_D$
    17. $X_2 \leftarrow R_1 + X_2$
    18. $X_2 \leftarrow X_2 \times X_2$
2:  return $(X_1, Z_1, X_2, Z_2)$
```

to store the base point x_D). One variable $a24 = (a + 2)/4$ is stored as a constant. The formula is shown in Listing 1.

Modular-Arithmetic Subroutines. To reduce code size, we implemented the modular arithmetic for addition, subtraction, and multiplication in subroutines. These subroutines are called by the main program. Furthermore, each subroutine can be called with different memory-page selection indices, which allows the subroutine to operate on different memory banks. Note that these subroutine implementations are responsible for the major part of the program ROM.

Modular reduction has been implemented efficiently by exploiting the underlying pseudo-Mersenne prime field form of $2^{255} - 19$. Fast reduction can be applied by basically shift and add operations. Shifting is done by multiplications with a constant. We applied an iterative modular reduction method, meaning that we first perform the arithmetic operation and reduce the result afterwards (to lower complexity). For modular reduction after a 256-bit addition/subtraction, we stored the carry/borrow bit using custom instructions named `STC`, `STI`, and `STX`. Then, we add/subtract the constant 38 from the result depending on the carry/borrow bit; otherwise zero is added/subtracted to provide constant runtime. For modular reduction after a 256-bit multiplication, the higher 256 bits of the result are multiplied with 38 and added to the lower 256 bits. This can efficiently be done using the special-purpose `MULADD` and `MULACC` instructions.

We implemented modular inversion based on Fermat's little theorem. It requires 11 multiplications and 254 squarings for a 256-bit modular inversion and runs in constant time. Squarings can be faster than multiplication, but we decided to re-use the multiplication routine to avoid additional code size for modular squaring. In our implementation, the same multiplication instruction is consecutively called up to 99 times. It is worth to note that we also evaluated if a dedicated loop instruction would further reduce the area requirements. However, it turns out that the required repeating logic would take more area than the synthesizer is able to optimize in look-up tables so we decided to keep the repetition of several multiplication instructions in ROM.

4.2 A Streaming API for `Crypto_box`

We decided to implement a streaming API that allows efficient authenticated encryption on WISPs. Basically, our implementation is able to encrypt and authenticate arbitrarily long data. However, due to the limited resources that we have available, we decided to work on 64-byte chunks only. It is therefore necessary to stream the data from external (non-volatile) memory (where WISPs usually store sensed data) and to perform encryptions in a streaming mode. The data authentication tag is calculated in parallel and can be retrieved after the encryption of the last data block. The format of the `crypto_box` output *starts* with the authentication tag, so our streaming API requires the reader to re-arrange data. This is no problem for a reader with reasonably large memory; it is impossible in the small memory of our ASIP if we want to support messages that are longer than 64 bytes. In addition to that, we provide a method to decrypt and authenticate one block of data which leads to a plaintext data payload of 32 bytes. These 32 bytes can for example be used to submit commands or status data to the WISP device (e.g., an Internet server requests sensitive data from the WISP or confirms the receipt of authenticated encrypted data).

The streaming API supports four commands: `INIT`, `FIRST`, `UPDATE`, and `FINALIZE` (listed in Table 1). We describe these commands now in a more detail:

Initialization. The `INIT` command initializes the internal state for authenticated encryption. The state for HSalsa20 has 512 bits and therefore requires two memory banks. It is initialized with a 192-bit nonce that needs to be stored into `R0` before calling `INIT`. Additionally, the state is initialized with the XSalsa20 constant σ, which is loaded from ROM, a dedicated block counter value that is incremented after each processed block, and the session key that is stored in `R6` after calling `DH-2`. The `INIT` process runs in constant time and needs to be executed only once after `DH-2` invocation.

Keystream Update. The `FIRST` and `UPDATE` commands are used to update the internal state of the keystream. The ciphertext is calculated by performing block encryption. In addition, also the state of the Poly1305 authentication tag is updated after encryption. Note that `FIRST` needs to be executed only for the first 64-byte plaintext, after that, `UDPATE` must be called. Before calling these commands, a 64-byte plaintext block needs to be stored in `R0` and `R1`. This plaintext block is then encrypted by applying the Salsa20 function on the internal state. At the end of Salsa20, the inital state has to be added to the internal state to obtain the final keystream. To save memory, we do not copy this at the beginning but calculate it on-the-fly. All needed data is already stored in RAM and ROM and therefore the execution time is not increased. The ciphertext is generated by XORing the internal state with the plaintext. Finally, the calculation of the authentication tag starts using Poly1305.

The special treatment for the first 64-byte block is necessary because this block is used as initialization state for Poly1305. As described in Sect. 2, this block consists of 32 bytes of zero and 32 bytes of plaintext stored in `R0` and `R1`. When `FIRST` is called, the plaintext is encrypted using Salsa20 and the encryption result of the first 32 zero-bytes is used to obtain the key for Poly1305. Note that during the FIRST process, the first 32 bytes of the ciphertext are zero. These zeros need to be replaced by the Poly1305 message authentication tag after the stream encryption.

After every 16 bytes of the ciphertext, Poly1305 is called to update the authentication tag. During the FIRST process, Poly1305 is executed twice, because R1 contains only 32 bytes of valid ciphertext. During the UPDATE process, Poly1305 is executed 4 times because the ciphertext is stored in both R0 and R1. Note that the preliminary authentication tag is stored in an internal memory bank and is updated after each UPDATE invocation. The main difference between UPDATE and FIRST is that during an update R0 holds a valid plaintext and therefore needs to be replaced with the appropriate ciphertext. Furthermore, the keys for Poly1305 are already generated.

Message-Authentication Tag. By sending the FINALIZE command, the final message authentication tag is calculated. The 16-byte tag is moved to the user-accessible memory-bank R0 where all other remaining bytes are set to zero.

The command flow for authenticated stream-encryption is given as follows:

1. Write a 192-bit nonce into the memory bank R0 and start the streaming operation by sending the INIT command.
2. Busy-wait polling until processor is ready.
3. Set the first 32 bytes of R0 to zero (this is the place holder for the zero-padded 16-byte message-authentication tag); send also the first 32 bytes of the plaintext to R1 and start the operation by sending the FIRST command.
4. Busy-wait polling until processor is ready; read back the 64 bytes of ciphertext from R0 and R1.
5. Write 64 bytes of the plaintext into R0 and R1 and continue authenticated encryption by sending the UPDATE command.
6. Busy-wait polling; read back the 64 bytes of ciphertext from R0 and R1.
7. Repeat Steps 5 to 6 until the whole plaintext is processed.
8. Send the FINALIZE command to generate the message-authentication tag.
9. Busy-wait polling until processor is ready; read back 32 bytes from R0 that contains the zero-padded 16-byte authentication tag.
10. Overwrite the first 32 bytes of the ciphertext (which have been previously set to zero) with the 32 bytes obtained in Step 9.

Decryption. The DECRYPTION command is used to decrypt and authenticate a 64-byte block of data. This block contains the zero-padded 16-byte authentication tag written into R0 and the 32-byte ciphertext written into R1. In a first step, the authentication tag of the ciphertext in R1 is calculated using Poly1305. This new tag is then subtracted from R0. If both tags match, the whole memory bank R0 will be zero and the ciphertext can be considered authentic. If the tags do not match, R0 will be different from zero. Finally, the ciphertext in R1 is decrypted using XSalsa20 as also done during the execution of FIRST except of the fact that only one memory bank needs to be processed to obtain the 32-byte plaintext stored in R1. Authenticated decryption is done as follows:

1. Write the 16-byte authentication tag into the memory bank `R0` and set all other bytes to zero; store the corresponding 32-byte ciphertext into `R1` and start the decryption operation by sending the `DECRYPT` command.
2. Busy-wait polling until processor is ready; read back the 32-byte plaintext from `R1` and check if `R0` is zero.

5 Implementation Results

We implemented our design in VHDL and used Cadence Encounter® RTL Compiler (v08.10-s238_1, 64-bit) for synthesis. For the following evaluation, we used the UMC 130 nm 1.2V/3.3V 1P8M LL logic CMOS process. For this process, one gate equivalent (GE) corresponds to the area of a two-input NAND gate of size 5.12 μm^2. All designs have been synthesized for a target frequency of 1 MHz. Furthermore, all area results are post-synthesis results (to make a fair comparison with related work) but note that the area requirements change during back-end design where the overhead for placement and routing (clock distribution, wire interconnections, etc.) is included. For power evaluations, we considered these overheads by simulating after place and route using First Encounter (v08.10-s273_1, 64bit) with NanoRoute (v08.10-s155).

Table 2 shows the results of our design for several multiplier configurations. We provide results for a digit size $w = 2, 4, 8, 12, 16$ to report numbers for a trade-off between speed and area. In terms of speed, most speed-up is observed for the X25519 key exchange. Depending on w, both Diffie-Hellman operations `DH-1` and `DH-2` require between 811 170 and 3 455 394 clock cycles. Note that `DH-2` needs an additional amount of 34 cycles compared to `DH-1` because it requires an inversion of the word order of the session key. Authenticated encryption using XSalsa20 and Poly1305 requires 6 641 cycles for initialization, the streaming update function needs between 7 443 and 9 291 cycles per 64-byte data block, and preparing the authentication tag needs only 62 cycles. Decryption of control messages can be performed in between 7 271 to 9 085 cycles.

In terms of area, our smallest design (using a 2-bit parallel digit-serial multiplier) requires 14.6 kGEs; the largest design needs 18 kGEs. The area of the controller is nearly the same independent of the size of the multiplier. The two program ROMs for Curve25519 and XSalsa20/Poly1305 have different lengths: 1 088 Lines of Code (LOC) are needed for Curve25519 and 1 625 LOCs are needed for XSalsa20/Poly1305. The ALU, in contrast, gets larger the more bits are processed in parallel. ROM for constants needs about 310 GEs and the 32-bit I/O interface needs 157 GEs. A closer look at the controller and datapath components shows that the major parts are due to the program ROM (31–50 %), the multiplier (9–31 %), the rotation unit (9–11 %), the multi-operation logic (4–5 %), the accumulator (4–5%), and the prefetch buffer (2–6 %).

Our smallest design needs around 40 μW of power at 1 MHz while the fastest design needs about 70 μW. Half of the power is spent for the RAM, the remaining power is consumed by the program ROM (15–26 %), the accumulator (8–24 %), the digit-serial multiplier (7–18 %), and the rotation logic (8 %).

Table 2. Performance of our `crypto_box` implementation for different multiplier digit-sizes w. We report numbers for X25519 key exchange (`DH-1` and `DH-2`) and authenticated encryption using XSalsa20 and Poly1305

w	Speed [Cycles][a]					Area [GEs][b]			
	`DH-1`	`DH-2`	`FIRST`	`UPDATE`	`DECRYPT`	Ctrl+ALU	ROM	Total incl. RAM	
								std-cells	macro
2	3 455 394	3 455 428	8 117	9 291	9 085	10 555	307	29 319	14 648
4	1 957 282	1 957 316	7 705	8 465	8 049	10 761	308	29 526	14 855
8	1 151 906	1 151 940	7 685	8 427	7 513	11 484	311	30 252	15 581
12	971 682	971 716	7 557	8 171	7 385	11 794	313	30 564	15 893
16	811 170	811 184	7 443	7 943	7 271	13 869	311	32 637	17 966

[a] `INIT` takes 6 641 cycles and `FINALIZE` needs 62 cycles for all digit sizes w.
[b] Total area is given for a 2 304-bit standard-cell based RAM design (18.3 kGEs) as well as an optimized synchronous one-port register file RAM with the FSC0L_D_SY memory technology from Faraday Technologies (needing 3 629 GEs).

We also had a closer look at the critical path of our design to evaluate the maximum supported frequency. The path starts at the instruction buffer in the controller, goes through the ROM in the memory unit and takes its way through the digit-serial multiplier in the ALU, and finally ends in RAM. Depending on the width of the multiplier, the duration of the critical-path delay is between 53.4 and 82.6 nano seconds. Thereof, the largest delay (64–75%) is caused by the adder structure of the digit-serial multiplier. The maximum frequency of our design is therefore 12–18 MHz (depending on configuration). This is fast enough for our targeted applications which are typically clocked with only a few MHz.

Further Area/Speed Trade-Offs. Further optimizations are possible, e.g., the entire 256-bit finite-field multiplication can be implemented as program code without needing a dedicated multiply control. For example, we implemented the multiplication as a classical product-scanning multiplication using 209 additional instructions. For the 130-bit multiplication in Poly1305, 83 additional instructions are necessary. As a result, we reduced the area requirements to 13.2 kGEs (including the RAM macro) for a 32-bit multiplier digit size of $w = 2$. The number of clock cycles for `DH-1` is however increased by 10.3 % to 3.852 million cycles. The cycle count for `FIRST` is increased by 12.3 % to 9 257 cycles and the `UPDATE` command takes 19.7 % longer, i.e., 11 571 cycles.

Comparison with Related Work. Table 3 lists different ASIC implementations of ECC that have similar field sizes (192–256 bits). While most related work focuses on efficient scalar multiplication on different types of curves, it shows that our design competes well even though more cryptographic services are offered. In fact, our processor supports a high-security stream cipher, a message authenticator, and a Diffie-Hellman key exchange using Curve25519. The required resources for those services, e.g., storage for the public key during key agreement is included in our numbers. Having these services available, our design is able to perform 128-bit public-key authenticated encryption using the given primitives while most related

Table 3. Comparison of ASIC implementations of ECC with similar field sizes

	Features of the (Co-)processor	**Size [bits]**	**Time [Cycles]**	**Area [GEs]**[a] std-cells	**Area [GEs]**[a] macro	**Area·Time** std-cells	**Area·Time** macro
Wolkerstorfer [37]	Weierstraß $\mathbb{F}_p/\mathbb{F}_{2^m}$	256	1 175 451	37 200[b]	n.a.	43.73	n.a.
Lai et al. [21]	Weierstraß $\mathbb{F}_p/\mathbb{F}_{2^m}$	256	252 067	197 028	n.a.	49.66	n.a.
Satoh et al. [33]	Weierstraß $\mathbb{F}_p/\mathbb{F}_{2^m}$	256	880 000	55 647	n.a.	48.97	n.a.
Liu et al. [23]	Twisted Edwards $\mathbb{F}_p = 2^{207} - 5131$	207	182 653	n.a.[c]	n.a.	n.a.	n.a.
Hutter et al. [18]	NIST P192, AES, SHA1	192	753 393	n.a.	21 502	n.a.	16.20
Wenger [36]	NIST P256	256	3 367 000	n.a.	27 244	n.a.	91.73
Ours (smallest)	Curve25519,	255	3 455 394	29 319	14 648	101.31	50.61
Ours (fastest)	Salsa20, Poly1305		811 170	32 637	17 966	26.47	14.57

[a]Area numbers include memory in either standard-cell based RAM technology or optimized RAM macro blocks. We list both for a fair comparison.
[b]Wolkerstorfer reported 31 kGEs for his core but this excludes the storage for the private key (scalar) and public key (X and Y coordinates). For a more fair comparison we included 6.2 kGE required for this memory.
[c]Authors reported 5 821 GEs for the size of their ALU. Memory is not included.

work targets authentication services only. In terms of Salsa20, we can compare our work with the smallest implementation reported so far which is due to Henzen, Carbognani, Felber, and Fichtner [16]. Their Salsa20 implementation needs 9.97 kGEs. For Poly1305 there are no previous hardware implementations to compare with.

When looking at the area-time-power (ATP) product, our design outperforms related work by more than a factor of 3. Wolkerstorfer reports a power consumption of more than 500 μW for a frequency of 1 MHz on a 192-bit curve using a 350 nm CMOS cell library. This corresponds to more than 130 μW on 130 nm CMOS and with a 256-bit curve (ATP: $>$ 5.68, standard-cell RAM). Hutter et al. report 1.6 mW on a 350 nm CMOS yielding an even larger ATP. Lai reported 578 μW for a 160-bit curve (ATP: 28.70, standard-cell RAM). Wenger's design, which is based on an 8-bit AVR clone, needs 76 μW (ATP: 6.97, RAM macros). Our fastest design needs 70 μW, which yields an ATP product of only 1.02 with RAM macros and 1.85 with standard-cell RAM.

Acknowledgements. We would like to thank Christof Paar for his valuable suggestions throughout preparing the final version of this paper.

References

1. Ananyi, K., Alrimeih, H., Rakhmatov, D.: Flexible hardware processor for elliptic curve cryptography over NIST prime fields. IEEE Trans. Very Large Scale Integr. (VLSI) **17**(8), 1099–1112 (2009). 4
2. Ashton, K.: That 'internet of things' thing. RFID J. **22**, 97–114 (2009). http://www.rfidjournal.com/articles/view?4986. 1

3. Bernstein, D.J.: The Poly1305-AES message-authentication code. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 32–49. Springer, Heidelberg (2005). http://cr.yp.to/papers.html#poly1305. 2, 6, 7
4. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). http://cr.yp.to/papers.html#curve25519. 2, 4, 5, 7
5. Bernstein, D.J.: The Salsa20 family of stream ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 84–97. Springer, Heidelberg (2008). http://cr.yp.to/papers.html#salsafamily. 2, 5, 7
6. Bernstein, D.J.: Extending the Salsa20 nonce. In Workshop record of Symmetric Key Encryption Workshop 2011 (2011). http://cr.yp.to/papers.html#xsalsa. 5, 7
7. Bernstein, D.J., Lange, T.: eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to. Accessed 28 Sep 2014. 4
8. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Hevia, A., Neven, G. (eds.) LatinCrypt 2012. LNCS, vol. 7533, pp. 159–176. Springer, Heidelberg (2012). http://cryptojedi.org/papers/#coolnacl. 2, 3, 4
9. Bogdanov, A.A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007). https://www.emsec.rub.de/research/publications/present-ultra-lightweight-block-cipher/. 3
10. Hewlett-Packard Development Company. CeNSE. http://www8.hp.com/us/en/hp-information/environment/cense.html. Accessed 25 Sep 2014. 2
11. Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A.H., Schwabe, P.: High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. Des. Codes Cryptograph. **17**, 1–22 (2015). http://dx.doi.org/10.1007/s10623-015-0087-1. 4
12. Gaj, K., Southern, G., Bachimanchi, R.: Comparison of hardware performance of selected Phase II eSTREAM candidates. In: State of the Art of Stream Ciphers Workshop - SASC 2007 (2007). http://www.ecrypt.eu.org/stream/papersdir/2007/027.pdf. 4
13. Good, T., Benaissa, M.: Hardware results for selected stream cipher candidates. In: Workshop on The State of the Art of Stream Ciphers - SASC 2007, pp. 191–204. ECRYPT (2007). http://www.ecrypt.eu.org/stream/papersdir/2007/023.pdf. 4
14. Guillermin, N.: A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 48–64. Springer, Heidelberg (2010). http://dx.doi.org/10.1007/978-3-642-15031-9_4. 4
15. Güneysu, T., Paar, C.: Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 62–78. Springer, Heidelberg (2008). http://iacr.org/archive/ches2008/51540064/51540064.pdf. 4
16. Henzen, L., Carbognani, F., Felber, N., Fichtner, W.: VLSI hardware evaluation of the stream ciphers Salsa20 and ChaCha and the compression function Rumba. In: International Conference on Signals, Circuits and Systems - SCS 2008, pp. 1–5. IEEE (2008). 4, 18
17. Hinterwälder, G., Moradi, A., Hutter, M., Schwabe, P., Paar, C.: Full-Size high-security ECC implementation on MSP430 microcontrollers. In: Aranha, D.F., Menezes, A. (eds.) LATINCRYPT 2014. LNCS, vol. 8895, pp. 31–47. Springer, Heidelberg (2015). http://www.emsec.rub.de/research/publications/Curve25519MSPLatin2014/. 4

18. Hutter, M., Feldhofer, M., Wolkerstorfer, J.: A cryptographic processor for low-resource devices: canning ECDSA and AES like sardines. In: Ardagna, C.A., Zhou, J. (eds.) WISTP 2011. LNCS, vol. 6633, pp. 144–159. Springer, Heidelberg (2011). http://mhutter.org/papers/Hutter2011ACryptographicProcessor.pdf. 18
19. Hutter, M., Schwabe, P.: NaCl on 8-bit AVR microcontrollers. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) AFRICACRYPT 2013. LNCS, vol. 7918, pp. 156–172. Springer, Heidelberg (2013). http://cryptojedi.org/papers/#avrnacl. 4
20. Joye, M., Yen, S.-M.: The Montgomery powering ladder. In: Kaliski, B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 291–302. Springer, Heidelberg (2003). http://cr.yp.to/bib/2003/joye-ladder.pdf. 8
21. Lai, J.-Y., Huang, C.-T.: A highly efficient cipher processor for dual-field elliptic curve cryptography. IEEE Trans. Circ. Syst II Express Briefs **56**(5), 394–398 (2009). 18
22. Langley, A., Chang, W.-T.: ChaCha20 and Poly1305 based cipher suites for TLS: Internet draft. https://tools.ietf.org/html/draft-agl-tls-chacha20poly1305-04. Accessed 1 Feb 2015. 3
23. Liu, Z., Wang, H., Großschädl, J., Hu, Z., Verbauwhede, I.: VLSI implementation of double-base scalar multiplication on a twisted edwards curve with an efficiently computable endomorphism. Cryptology ePrint Archive: Report 2015/421 (2015). http://eprint.iacr.org/2015/421.pdf. 18
24. Alpha Technology (INT) LTD. Implementation and analysis of Scrypt algorithm in FPGA (proof of concept). Technical report, Alpha Technology, Manchester, England (2013). https://alpha-t.net/wp-content/uploads/2013/11/Alpha-Technology-Scrypt-Analysis-on-FPGA-proof-of-concept.pdf
25. Ma, Y., Liu, Z., Pan, W., Jing, J.: A high-speed elliptic curve cryptographic processor for generic curves over GF(p). In: Lange, T., Lauter, K., Lisoněk, P. (eds.) SAC 2013. LNCS, vol. 8282, pp. 421–437. Springer, Heidelberg (2014). http://www.iacr.org/archive/ches2010/62250046/62250046.pdf. 4
26. McIvor, C.J., McLoone, M., McCanny, J.V.: Hardware elliptic curve cryptographic processor over GF(p). IEEE Trans. Circ. Syst. **53**(9), 1946–1957 (2006). 4
27. Meiser, G., Eisenbarth, T., Lemke-Rust, K., Paar, C.: Efficient implementation of eSTREAM ciphers on 8-bit AVR microcontrollers. In: International Symposium on Industrial Embedded Systems - SIES 2008, pp. 58–66 (2008). 4
28. Mentens, N.: Secure and efficient coprocessor design for cryptographic applications on FPGAs. PhD thesis, Katholieke Universiteit Leuven, Leuven-Heverlee, Belgium (2007). 4
29. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **48**(177), 243–264 (1987). http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf. 5, 8, 13
30. Sakiyama, K., De Mulder, E., Preneel, B., Verbauwhede, I.: A parallel processing hardware architecture for elliptic curve cryptosystems. In: IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP 2006, vol. 3, pp. 904–907. IEEE (2006). http://www.cosic.esat.kuleuven.be/publications/article-714.pdf. 4
31. Sample, A.P., Yeager, D.J., Powledge, P.S., Smith, J.R.: Design of a passively-powered, programmable sensing platform for UHF RFID systems. In: 2007 IEEE International Conference on RFID, pp. 149–156. IEEE (2007). https://sensor.cs.washington.edu/pubs/WISP-IEEE-RFID07-PostConf.pdf. 2

32. Sasdrich, P., Güneysu, T.: Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices. In: Goehringer, D., Santambrogio, M.D., Cardoso, J.M.P., Bertels, K. (eds.) ARC 2014. LNCS, vol. 8405, pp. 25–36. Springer, Heidelberg (2014). https://www.ei.rub.de/media/sh/veroeffentlichungen/2014/03/25/paperarc14 curve25519.pdf. 4
33. Satoh, A., Takano, K.: A scalable dual-field elliptic curve cryptographic processor. IEEE Trans. Comput. **52**(4), 449–460 (2003). 18
34. Sugier, J.: Low-cost hardware implementations of Salsa20 stream cipher in programmable devices. J. Pol. Saf. Reliab. Assoc. **4**(1), 121–128 (2013). http://jpsra.am.gdynia.pl/upload/SSARS2013PDF/VOL1/SSARS2013-Sugier.pdf. 4
35. Varchola, M., Güneysu, T., Mischke, O.: MicroECC: a lightweight reconfigurable elliptic curve crypto-processor. In: 2011 International Conference on Reconfigurable Computing and FPGAs, pp. 204–210 (2011). 4
36. Wenger, E.: A lightweight ATmega-based application-specific instruction-set processor for elliptic curve cryptography. In: Avoine, G., Kara, O. (eds.) LightSec 2013. LNCS, vol. 8162, pp. 1–15. Springer, Heidelberg (2013). https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=70640. 18
37. Wolkerstorfer, J.: Is elliptic-curve cryptography suitable for small devices? In: Oswald, E. (ed.) Workshop on RFID and Lightweight Crypto - RFIDsec 2005 (2005). 18
38. Yan, J., Heys, H.M.: Hardware implementation of the Salsa20 and Phelix stream ciphers. In: Canadian Conference on Electrical and Computer Engineering - CCECE 2007, pp. 1125–1128. IEEE (2007). http://www.engr.mun.ca/~howard/PAPERS/ccece07_yan.pdf. 4