Compact and Flexible FPGA Implementation of Ed25519 and X25519

FURKAN TURAN and INGRID VERBAUWHEDE, COSIC, KU Leuven, Belgium

This paper describes an FPGA cryptographic architecture which combines the elliptic curve based Ed25519 digital signature algorithm and the X25519 key establishment scheme in a single module. Cryptographically, these are high security elliptic curve cryptography algorithms with short key sizes and impressive execution times in software. Our goal is to provide a lightweight FPGA module, that enables them on resource-constrained devices, specifically for IoT applications. In addition, we aim at extensibility with customisable countermeasures against timing and differential power analysis side-channel attacks and fault-injection attacks. For the former, we offer a choice between time-optimised versus constant-time execution, with or without Z-coordinate randomisation and base point blinding; and for the latter, we offer enabling or disabling default-case statements in the FSM descriptions. To obtain compactness and at the same time fast execution times, we make maximum use of the DSP slices on the FPGA. We designed a single arithmetic unit that is flexible to support operations with two moduli and non-modulus arithmetic. In addition, our design benefits in-place memory management and local storage of inputs into DSP slices' pipeline registers and takes advantage of distributed memory. These eliminate a memory access bottleneck. The flexibility is offered by a micro-code supported instruction-set architecture. Our design targets 7-Series Xilinx FPGAs and is prototyped on a Zynq SoC. The base design combining Ed25519 and X25519 in a single module, and its implementation requires only around 11.1 K LUTs, 2.6 K registers and 16 DSP slices. Also it achieves performance of 1.6 ms for a signature generation, and 3.6 ms for a signature verification for a 1024-bit message with a 82MHz clock. Moreover, the design can be optimised only for X25519 which gives the most compact FPGA implementation compared to previously published X25519 implementations.

CCS Concepts: • **Security and privacy → Security in hardware**; *Hardware security implementation*; Embedded systems security;

Additional Key Words and Phrases: ECC, Ed25519, EdDSA, X25519, ECDH, Curve25519, FPGA

ACM Reference Format:
Furkan Turan and Ingrid Verbauwhede. 2018. Compact and Flexible FPGA Implementation of Ed25519 and X25519. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (February 2018), 21 pages.

1 INTRODUCTION

In this paper, we present the first architecture to implement the Ed25519 digital signature algorithm [3] in hardware, to the best of our knowledge. Moreover, it has a compact and flexible design that combines the basic X25519 key establishment protocol [2] with the complex Ed25519 in a single module. These two algorithms offer elliptic curve cryptography with short key sizes, straightforward and secure implementations. Additionally, impressive speed records have been demonstrated in software [2]. Our main purpose was to investigate the suitability of these two algorithms on resource-constrained devices. Considering this, we optimised our design for low

Authors' address: Furkan Turan; Ingrid Verbauwhede, COSIC, KU Leuven, Kasteelpark Arenberg 10, bus 2452, Leuven, 3001, Belgium, firstname.lastname@esat.kuleuven.be.

1539-9087/2018/2-ART1 $15.00
https://doi.org/

resource utilisation in addition to offering optional countermeasures to strengthen it against various attacker models. These make it suitable for embedded devices and IoT applications.

Greater network connectivity and reduced cost are the two major enablers of IoT. The more IoT nodes merge into our lives, the greater amount of data they produce, and the crucial it becomes to protect this data. For instance, smart homes are appealing for many people, and the market is expected to reach $27 billion by 2021 [1]. As a result the amount of personal data going over the wires will grow significantly in coming years. Therefore, the protection of this data is of paramount importance. However, providing such protection while aiming at low cost with resource constraints is a challenge. Although symmetric key cryptography and lightweight ciphers are favourable for this purpose, the growing number of connected nodes reveal issues with key management and authentication. Public key cryptography promises solutions for these issues. Furthermore, elliptic key cryptography is often a better match with highly secure and low-cost algorithms. Two of them are Ed25519 and X25519, which offer high security with signatures and short keys, respectively for cryptographic signature and key establishment algorithms. These algorithms are defined with the same curve primitives. Taking advantage of that, we combine the algorithms in a single implementation to achieve a compact design. Hence our lightweight implementation is favourable for low resource devices, e.g. IoT nodes.

Together with our design, we share the design decisions to reduce its resource utilisation on 7-Series Xilinx FPGAs. These techniques include the design of a modular arithmetic unit that makes use of in-place memory-management, construction of a flexible datapath and the utilisation of micro-coding techniques. With these strategies we achieve low utilisation with high performance. Although, implementing only Ed25519 requires a lot more than X25519, our design combining them achieves comparable results to existing efficient X25519 hardware implementations. Moreover, our design's X25519 optimised implementation needs considerable less resources compared to all related works.

In addition to the optimisations at the core of the design, we provide users with customisations to enable or disable synthesising additional features. One of these features is to synthesise for only X25519, which trims out a great portion of the implementation, and achieves a tiny hardware. The other features involve users in the decision making for the trade-offs between security and low resource utilisation. These features offer three countermeasures against side-channel attacks, and an option to strengthen the design against fault attacks.

Our contributions include the first Ed25519 hardware design and underlying design strategies to minimise the resource utilisation of its implementation. In addition, we present an optimised base implementation together with the additive cost of the security features to it. Moreover, we compare these costs and their evaluation to aid users choosing the right trade-off.

2 RELATED WORK

There are many publications addressing Elliptic Curve Cryptography (ECC) [7] implementations on both software and hardware; however, to the best of our knowledge, this paper is the first to implement Ed25519 on hardware. However, our design's capability to implement X25519 individually allows to consider the following X25519 hardware implementations as related work. Sasdrich et al. implemented an X25519 hardware [18] offering both single- and multi-core versions of their design. It executes in constant-time to strengthen the design against attackers. Moreover, they later extended their design with various side channel resistance features [19]. Koppermann et al. proposed another implementation [12] of X25519 that aims at high performance with little latency. They later published an improved version of it [13] that achieves computing a session key in less than 100 µs. It also offers constant-time execution and randomised projective coordinates. These implementations and ours are all realised on the Xilinx Zynq SoC and benefit from the DSP slices

of the target FPGA platform. Therefore, the comparison of our design to related work, given in Section 5.3, yields fair conclusions to evaluate our design decisions. These comparisons also put the extra hardware cost of Ed25519 in perspective to the X25519 implementations.

3 BACKGROUND

Curve25519 [2] is a Montgomery curve defined by Dan Bernstein as $E : \{(x, y) \in \mathbf{F}_q \times \mathbf{F}_q \ : y^2 = x^3 + 486662x^2 + x\}$ where q is $2^{255} - 19$. In the following sub-sections, the descriptions of two Curve25519 implementations are given, as this paper presents their hardware implementation. The description does not aim at introducing all details but gives an overview on the abilities that a hardware module should have to execute them.

3.1 X25519

X25519 is an Elliptic Curve Diffie-Hellman (ECDH) [7] protocol based on Curve25519. Scalar multiplication is the only function used in the protocol to calculate the key as shown in Algorithm 1. The curve definition and allowing simplifications at implementing the multiplication offer great performance advantages to the protocol. These advantages let its software implementation to set the Diffie-Hellman speed records [2]. Moreover, with such merits it is attractive not only for high performance but also for security-critical applications on resource-constrained devices.

ALGORITHM 1: X25519 Operation

Input: a as 256-bit secret or public key, P as point on the curve
Output: x as 255-bit public key or shared secret

1 $(x, y) \leftarrow aP$

For key establishment with X25519, Alice and Bob first generate a random number as their secret key. Then, they execute the X25519 operation with the key and the curve's base point (which is public) as the inputs to calculate their own public key. Later, both parties exchange their public keys and execute the X25519 operation once again with their secret and other party's public key as the inputs to obtain a shared secret.

The corresponding scalar multiplication underlying the X25519 operation requires the sequential execution of modular arithmetic functions to calculate the result. An efficient implementation of this multiplication can follow the Montgomery ladder approach of Algorithm 2, which offers constant time execution. This approach defines a ladder step and applies it to all the bits of the scalar input repeatedly. These steps calculate add and double operations on two curve points. At the first

ALGORITHM 2: Scalar Multiplication with Montgomery Ladder

Input: S scalar, Q a point on the curve
Output: P a point on the curve

1 $P \leftarrow$ *Neutral element of the group of points*
2 **for** *each bit b of S* **do**
4 | **if** *b is* 1 **then** *swap the values of P and Q*
6 | $Q \leftarrow Add(Q, P)$
7 | $P \leftarrow Double(P)$
8 | **if** *b is* 1 **then** *swap the values of P and Q*
end

step, they are the input point and the point that is the neutral element of the group of points. In each step, first the points are swapped, then points' addition and doubling on the curve are calculated, and later the resulting points are swapped again. The details on how we avoid the key dependent swap operation are given in Section 4.1 and Algorithm 4. For X25519 operation, an implementation of this multiplication with restricted-X coordinates is recommended as stated in the Appendix B of [2]. This coordinate system simplifies the multiplication by taking advantage of the fact that the key establishment does not depend on the Y-coordinate value. With restricted-X coordinates, executing the point add and double functions requires four squarings, six multiplications and eight additions/subtractions in the prime field $\mathbb{F}_q$ (see Fig.1 in [18]).

3.2 Ed25519

Ed25519 is an Edwards-curve Digital Signature Algorithm (EdDSA) [10] interesting for many applications as it offers fast execution of high-security elliptic curve cryptography. It is originally designed for high performance, and our focus is to allow it on the resource constrained devices.

The algorithm is defined on the twisted Edwards Curve, birationally equivalent to the Curve25519. This curve's equation is $E : \{(x, y) \in \mathbf{F}_q \times \mathbf{F}_q \ : -x^2 + y^2 = 1 + dx^2y^2\}$ where d is 121665/121666. The listing of operations to sign a message and verify the signature are given in Algorithm 3.

ALGORITHM 3: Ed25519 Sign and Verify Operations

Define: $H(x)$ is SHA512 hash of x
B is the base point on the curve
l is a 253-bit prime
$\underline{S}$ is the 256-bit little-endian encoding of integer S
$\underline{R}$ is encoding of point R as: $R_Y + (R_X \ \& \ 1)$
$\underline{A}$ is encoding of point A as: $A_Y + (A_X \ \& \ 1)$

Signing

Input: $\underline{A}$ as 256-bit public key,
SK as 256-bit secret key,
M arbitrary length message,

Output: $(\underline{R}, \underline{S})$ 512-bit signature

1 $(h_0, ..., h_{511}) \leftarrow H(SK)$
2 $a \leftarrow 2^{254} + \sum_{i=3}^{253} 2^i h_i$
3 $r \leftarrow H(h_{256}, ..., h_{511}, M)$
4 $R \leftarrow rB$
5 $S \leftarrow (r + H(\underline{R}, \underline{A}, M)a) \ mod \ l$

Verifying

Input: $\underline{A}$ as 256-bit public key,
M arbitrary length message,
$(\underline{R}, \underline{S})$ 512-bit signature

Output: A boolean decision

6 $x \leftarrow SB$
7 $y \leftarrow R + H(\underline{R}, \underline{A}, M)A$
8 $x \stackrel{?}{=} y$

These operations include hashing, $\mathbb{F}_q$ and $\mathbb{F}_l$ arithmetic. The former field is the one that the curve is defined over. For the latter, l is the order of the base point defined as a 253-bit prime number, 2^{252} + 27742317777372353535851937790883648493. Though the hash function can be replaced, SHA512 is preferred as the default.

As shown in Algorithm 3, the signing scheme starts with taking the SHA512 hash of secret key SK and message at the first three lines of the algorithm. Then the first half of the signature is calculated by the scalar multiplication of curve's base point and the hash at line 4. Finally, the second half of the signature is calculated with the equation on line 5. The verification scheme verifies the signature by the group equation on line 6 and 7. Executing it requires two scalar multiplications, hashing and modulus l addition.

For an efficient implementation of the scalar multiplication on this curve, [3] suggests using the Montgomery ladder approach with extended coordinates described in Section 3.1 of [8]. The efficiency is a results of avoiding the expensive modular division. The recommended ladder implementation requires 18 multiplications and 18 additions/subtractions at each step.

3.3 Comparison of Implementing Ed25519 and X25519

This section gives a comparison of the implementation requirements of Ed25519 and X25519. In general, implementing X25519 requires significantly less effort because, first, it only requires calculating a scalar multiplication function on the curve. Secondly, the scalar multiplication can be implemented cheaply with restricted-X coordinates (see Equation 1, where Z is a user-picked auxiliary coordinate). In comparison, Ed25519 requires several calculations which include scalar multiplication, modulus l arithmetic and hash function. The curve of Ed25519 leads to an expensive scalar multiplication operation. Since it uses the value of the Y-coordinate (the recovery of which from X is not trivial as will be introduced later), the restricted-X coordinate implementation of X25519 is not directly applicable to Ed25519. However, using an extended-coordinate system (see Equation 2) for Ed25519 eliminates the need for expensive modular division, at the price of extra storage for the auxiliary coordinate components Z and T for each point.

$$(x, y)_{\texttt{Affine}} \mapsto (X, Z)_{\texttt{Restricted-X}} = (xZ, Z) \tag{1}$$

$$(x, y)_{\texttt{Affine}} \mapsto (X, Y, Z, T)_{\texttt{Extended}} = (xZ, yZ, Z, xyZ) \tag{2}$$

The difference of the curves used in both algorithms limits implementation of one to inherently support the other. Nevertheless, the points can be moved between the curves since they are birationally equivalent. As a result, the scalar multiplication function implemented for one curve can be used for the other with a penalty of performing a coordinate conversion. However, this conversion is not trivial because Ed25519 and X25519 benefit from different coordinate systems. The conversion requires both mapping the points from one curve to the other, and changing the coordinate system.

The conversion of a point on the twisted Edwards curve using extended coordinates, to a point on the Montgomery curve using restricted-X coordinates is relatively simple with the transformation given in Equation 3. However, the conversion in the reverse direction is not straightforward. As shown in Equation 4, this conversion requires knowing the source point's Y coordinate which is not available in the restricted-X coordinates. Moreover, recovering the Y coordinate requires solving a complex equation which leads to two solutions with the same magnitude but different sign. Therefore, completing the conversion needs verifying the solutions with the curve equation.

In conclusion, the benefits of using X25519's cheap scalar multiplication with restricted-X coordinates for Ed25519 are lost due to the complexity of coordinate conversion. Moreover, when using Ed25519's scalar multiplication for implementing X25519 loses up its cheaper multiplication.

$$(X, Y, Z, T)_{\mathsf{Ed}} \mapsto (X, Z)_{\mathsf{M}} = (Y_{\mathsf{Ed}} + Z_{\mathsf{Ed}}, -Y_{\mathsf{Ed}} + Z_{\mathsf{Ed}}) \tag{3}$$

$$\begin{aligned}(X, Z)_{\mathsf{M}} \mapsto (X, Y, Z, T)_{\mathsf{Ed}} = \left(\frac{X_{\mathsf{M}}}{Y_{\mathsf{M}}}, \frac{X_{\mathsf{M}} - 1}{X_{\mathsf{M}} + 1}, 1, \frac{X_{\mathsf{M}}}{Y_{\mathsf{M}}}\frac{X_{\mathsf{M}} - 1}{X_{\mathsf{M}} + 1}, \right) \\ where\ Y_{\mathsf{M}} = \pm\sqrt{X_{\mathsf{M}}^3 + AX_{\mathsf{M}}^2 + X_{\mathsf{M}}}\end{aligned} \tag{4}$$

We would like to remark that qDSA [15] is another digital signature scheme, proposed after we started this work. It again uses Curve25519, but in comparison to Ed25519 it enjoys the Montgomery curve even allowing the restricted-X coordinates implementation. It is also free from group operations, but it still requires the overhead of executing the same second modulus operations. Furthermore, qDSA is a bit more constrained on the choice of hash function, while Ed25519 emphasise its robustness against hash collisions. Since hash functions often demand expensive hardware, freedom to choose a lightweight hash is an advantage of Ed25519. In summary, qDSA can also be a good fit for a hardware implementation especially when combined with X25519. It is clear that there are several difficulties to implement Ed25519. Therefore, one research direction is to investigate if the recently proposed qDSA scheme could alleviate some of these difficulties and as such be a better match to the X25519 implementation. We keep this as future work.

4 IMPLEMENTATION

In this section we describe the design strategies that we followed to combine both Ed25519 and X25519 in a single hardware. Our design minimises resource utilisation, and provides it with countermeasures against attacks. Ed25519 is implemented using extended-coordinates representation for its twisted Edwards curve to execute the scalar multiplication. Moreover, X25519 is supported either executing on Ed25519's implementation, or with the restricted-X coordinates on the Montgomery curve it is defined for. The former is the case when our design is configured to support both Ed25519 and X25519 on the same hardware, while the latter is used for X25519-only configuration. Our design also offers customisability which provides users with *Verilog Compiler Directives* to optimise the design at synthesis time for resource, performance or security. Moreover, we introduced domain-specific programmability in our architecture, which enables modifying its underlying operations with instructions and micro-codes.

The target of our implementation is Xilinx's 7-series FPGAs that employ DSP48E1 slices [22] since we designed our arithmetic unit hardware module to benefit from these slices at most. We specifically used a ZedBoard which comes with a Zynq-7020 [23] System-on-Chip (SoC) that offers both hardware and software programmability. We also verified the correctness of our hardware by mapping it as an accelerator to software and executing the operations with the test-vectors provided together with the TweetNaCl [4] and Ed25519-donna reference implementations.

In this section, we first describe the design of our datapath in detail, later give the details of the arithmetic unit. Afterwards, we introduce our optional security features and the compiler directives to customise the design.

4.1 Datapath

Our architecture operates like a processor employing the Harvard model as shown in Figure 1. For each command received, a set of instructions are executed sequentially. The description of the

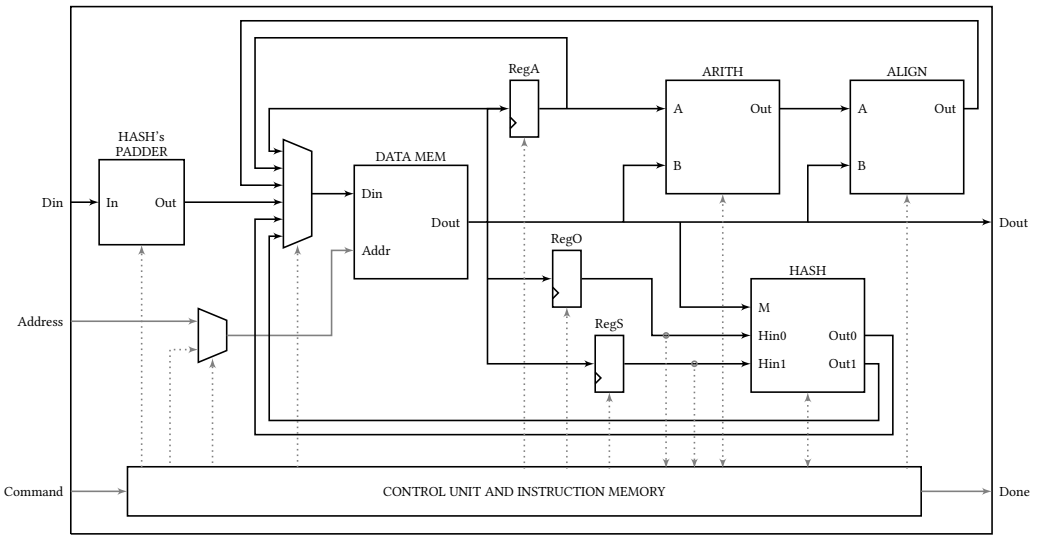

Fig. 1. The figure shows our datapath design with the interconnection of the basic building blocks. With command input, the control unit starts executing series of instructions from the instruction memory. Load/Store instructions move data between the memory and other blocks. Arithmetic and hash instructions first read inputs from the memory, execute the operation and write the result back to the memory. *(All the black lines correspond to 256-bit data signals, and the grey lines correspond to control signals, the width of which vary for different blocks.)*

datapath is given in the following paragraphs with explaining the functionality of each block in detail.

The instruction memory is located within the control unit and stores instructions for both arithmetic and hash units and for various memory operations. It is implemented as a ROM loaded with a predefined array of instructions. As the structure is given in Listing 1, the instructions are typed using Verilog parameters to maintain human readable format. Each instruction consists of a 2-bit opcode and 5-bit operand. We defined two type of instructions. The first type include memory operations such as loads and stores to move data between RAM, registers, input and execution units. For this type, the operand corresponds to an address for the data memory. The second type target arithmetic and hash unit operations that will be described in Section 4.2. For this type of instructions, the operand field is used only to pass the target operation's name to the control unit, as the operations' input and output addresses are fixed.

Micro-coding is implemented as an aid to the scalar multiplication instruction. It defines the consecutive execution of arithmetic unit operations to calculate the multiplication. As discussed in Section 3, the Montgomery Ladder algorithm [8, 14] is preferred to implement the multiplication. The algorithm defines a ladder step that is executed for each bit of the scalar, and the step consists of add and double functions which are calculated with the sequential execution of modular arithmetic operations, e.g. Algorithm 4. We prefer to follow the micro-coding approach to sequence the execution of these operations. Alternatively, we could design a fixed state machine to handle the sequence with state visits. However, it would require a sophisticated next-state logic and poor code readability. Conversely, micro-code simplifies the design and results into lower resource utilisation with a simpler state machine. In our platform, we can save the code in the data memory without increasing the memory size.

Listing 1. Instruction Memory Context Structure (Example)

```
module instructionROM (                         // Define the module IO
    input  wire [7:0] address,
    output wire [6:0] instruction );

    reg [6:0] rom [0:SIZE];                     // Define ROM as an array of registers

    parameter LoadA       = 2'b00;              // Define 2-bit opcodes
    parameter StoreA      = 2'b01;
    parameter Op          = 2'b10;

    parameter scalar_mult = 5'b00011;           // Define 5-bit operands

    initial begin                               // Define the ROM content
        rom[1] = { LoadA  , 5'h08       };
        rom[2] = { StoreA , 5'h0A       };
        rom[3] = { Op     , scalar_mult };
```

Our micro-code design works as follows. A bit-string encodes the list of operations (in lines 1-18 of Algorithm 4), and is loaded to the RAM at design time. Each operation is represented with 14 bits, and the bit-string concatenates them. For scalar multiplication, the control unit first reads the bit-string into RegO. That provides the unit access to only the least significant 14 bits of the RegO, which corresponds to a single operation. However, RegO is implemented as a shift register, and shifted by 14 bits for switching to the next instruction.

Swapless scalar-multiplication is preferred to protect against timing attacks. For this purpose, we eliminated the conditional swap operations which were introduced before in Section 3. These operations swap the two curve points at the beginning and end of the ladder step if the corresponding bit of the scalar at the step is one (Algorithm 2 and Alg.1 in [12]). We eliminated the swap by defining two different sets of operations for the two values of the bit. These sets are shown in the Algorithm 4 for Montgomery curve with restricted-X coordinates, and similar sets are used for twisted Edwards curve with extended coordinates. At design time, we create two bit-strings each for one column. One bit-string encodes the set of operations for the bit value equals one, and the other for the value equals to zero. Then we store them at two different addresses of the data memory. At run time, we skip the swap operations by executing the ladder step by selectively loading the bit-strings. As a result, we eradicate spending an extra or conditional cycle for them. This approach does not increase the number of required operations to execute the scalar-multiplication. Moreover, the sets require the same number and sequence of arithmetic operations. Hence, they eliminate information leaks as a countermeasure against timing side-channel attacks (which will be introduced in the further sections) at the expense of extra memory to store the two micro-code bit-strings instead of one.

One-hot state machine with forward edge transitions is employed to simplify the state-machine implementation of the control unit and to minimise its resource utilisation. The control unit consists of a hierarchy of state machines. The top state machine has a dedicated state for each instruction. Each of these states employs an individual state machine for managing the underlying operations sequentially to execute the corresponding instruction. While the outer state machine has only 23 states, the inner state machines consist of greatly varying number of states, which sometimes require complex next-state decision logic. To implement them best, we opt for a custom one-hot state machine and restricted its state transitions to only forward-edges. These two approaches are described in the following paragraphs.

For easing the synthesiser's work to map the state definitions into logic, we preferred a custom definition of a one-hot state machine. Our outer state machine manually enumerates the states, each employs an inner state machine that map their states to an individual bit of the RegS. Alternatively, we could just implement the state machines with basic Verilog primitives, and configure the

ALGORITHM 4: Scalar Multiplication without Swap Operation

Below listing corresponds to how the add and double operations of Algorithm 2's for loop is implemented (for Montgomery curve with restricted-X coordinates). The left column shows the original implementation [4] which has conditional swap on the loop's variable bit b. The right columns show our implementation which uses either one or the other list of operations selected with the value of b, and avoids the swap and any change in the operation sequence.

Define: $a \leftarrow Q_Z,\ b \leftarrow Q_X,\ c \leftarrow P_X,\ d \leftarrow P_Z,\ X \leftarrow 121665,\ Y \leftarrow Q_X$ (of the first loop), and $t1, t2, t3, t3 \leftarrow 0$

	Original Implementation	**Our Modified Implementation**	
	For any b	For b is 0	For b is 1
0	SWAP Q and P, if b is 1		
1	t1 ← a + c	t1 ← a + c	t1 ← b + d
2	a ← a − c	t3 ← a − c	t3 ← b − d
3	c ← b + d	t2 ← b + d	t2 ← a + c
4	b ← b − d	b ← b − d	b ← a − c
5	d ← t1 × t1	t4 ← t1 × t1	t4 ← t1 × t1
6	t2 ← a × a	a ← t2 × t3	a ← t2 × t3
7	a ← c × a	t2 ← t3 × t3	t2 ← t3 × t3
8	c ← b × t1	c ← b × t1	c ← b × t1
9	t1 ← a + c	t1 ← a + c	t1 ← a + c
10	a ← a − c	a ← a − c	a ← a − c
11	b ← a × a	t3 ← a × a	t3 ← a × a
12	c ← d − t2	b ← t4 − t	b ← t4 − t
13	a ← c × X	a ← b × X	a ← b × X
14	a ← a + d	a ← a + t	a ← a + t
15	c ← c × a	c ← b × a	d ← b × a
16	a ← d × t2	d ← t3 × Y	c ← t3 × Y
17	d ← b × Y	a ← t4 × t2	b ← t4 × t2
18	b ← t1 × t1	b ← t1 × t1	a ← t1 × t1
19	SWAP Q and P, if b is 1		

synthesiser to prefer one-hot representation and resource-sharing. Though it would yield some advantages of one-hot representation, the design would be dependent on the synthesiser area-optimisation strategies. Hence, it would hurt the portability of high-level description, and leave little room to benefit from performance optimisation strategies.

Besides the resource sharing design, we spent effort on the next-state logic to further optimise our state machines. Our design limits the state transitions of the inner state machines to forward paths only, which sometimes require executing the same operations in different states. However, it simplifies the next-state logic significantly. Eventually, the next-state logic is reduced to the loading of the RegS with one at the initial state, and shifting it by one bit for the state transitions.

Distributed memory is a better match to implement the RAM in our datapath compared to block RAM for minimising area use. As an alternative, we experimented with the design which instantiates a small Block RAM for each DSP slice, as was done by Sasdrich et al [18]. Although that is an advantageous strategy for implementing an arithmetic unit dedicated for a single modulus operation, we prefer a single datapath for both moduli to reduce resource utilisation. Therefore, our DSP slices parse their inputs for different word sizes, which results into using signals from various RAMs. Distributed memory helps the implementation as memory elements can be divided into words, and located closer to the logic elements they are processed at. Hence, we achieved better placement results with distributed memory, and reached a lower resource utilisation.

4.2 Modular Arithmetic Unit

We designed the modular arithmetic unit with the aim of using the DSP slices of the FPGA for fast execution with minimum resource utilisation. These slices provide 18×25-bit high-performance single cycle multipliers, and configurable datapath for various operations. Our unit divides the

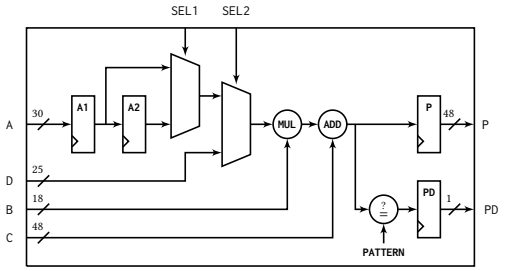

Fig. 2. Our DSP slice module instantiates a multi-functional Xilinx 7 Series FPGA DSP48E1 Slice and configures it to construct the drawn datapath. (The figure shows only the data signals.)

operands into 17-bit words, to make the word-wise operations fit to DSP slices' input size. Moreover, considering the 255-bit modulus q operations, 17 is a fine word size as it is divides 255 with no remainder. The unit instantiates 16 DSP slices (with an extra slice for special operations that will be described later), and configures them to construct the datapath given in Figure 2. The input and output signals of the unit are parsed into words and connected to these slices. Also a state machine is implemented inside the unit to control the DSP slices for the arithmetic operations' execution.

Our arithmetic unit implements all the operations needed to calculate the Ed25519 sign and verify functions given in Algorithm 3 except SHA512. We implemented three modular arithmetic operations with modulus q which are multiplication, addition and subtraction for the scalar multiplications shown in line 4, 6 and 7 of the algorithm. Moreover, modulus l addition and multiplication operations are needed for the curve equations. In the algorithm, the modulus l operations may receive an input with a bit-length that is greater than the modulus, e.g. SHA512 creates a 512-bit result while l is 253-bits. Therefore, we chose to implement a reduction operation to simplify the arithmetic. The addition of two $\mathbb{F}_l$ numbers never produces an output greater than q. Therefore, it is possible to perform modulus l addition by re-using the already implemented modulus q adder if they are reduced already to l. However, the same is not true for the modular multiplication which can be solved by either a dedicated modulus l or a regular multiplier that is followed by modulus l reduction. We preferred the latter for two reasons: the reduction avoids a dedicated modulus l addition, allowing the re-use of modulus q adder. A dedicated modular multiplier is an expensive piece of hardware, while regular multiplication can re-use the hardware primitives already implemented for the modulus q multiplier.

The align module of the datapath (Figure 1) is designed as a post-processor for the arithmetic unit. The datapath is designed for 256-bit word size; however, the arithmetic unit does not always satisfy this size. For example, it can produce the output as two partial results which are not 256-bit word aligned (How the operations calculate their output will be introduced later). In such cases, the align module constructs the final result by moving the second output's lowest bits to the first output's highest part. An extra functionality is to perform the point encoding operations shown in Algorithm 3.

In the following paragraphs of this section, we introduce how our arithmetic unit handles the operations with detailed explanation for each operation in individual subsections.

4.2.1 Mod q Multiplication. The algorithm we used for the modular multiplication for the prime field $\mathbb{F}_q$ with $q = 2^{255} - 19$ is given in Algorithm 5. The modular multiplication is calculated in several steps which are multiplication, reduction and carry propagation. The first step executes a school-book multiplication by dividing the operands into small blocks to fit the parallel execution

of multiple DSP slices. Further steps combine the partial results at the output of these slices and calculate the final result. The details of all these steps are explained in the following paragraphs.

ALGORITHM 5: Mod q Multiplication

Input: A and B

Output: $P \leftarrow (A \times B) \mod (2^{255} - 19)$

Define: $\underline{X} = X \times 19$,
$A = A_{00} + A_{01} \times 2^{17} + A_{02} \times 2^{34} + \dots$,
$B = B_{00} + B_{01} \times 2^{17} + B_{02} \times 2^{34} + \dots$

Step 1 - Schoolbook Multiplication

1 $P_{00} \leftarrow A_{00}\,B_{00} + \underline{A_{14}}\,B_{01} + \underline{A_{13}}\,B_{02} + \underline{A_{12}}\,B_{03} + \cdots + \underline{A_{02}}\,B_{13} + \underline{A_{01}}\,B_{14}$

2 $P_{01} \leftarrow A_{01}\,B_{00} + A_{00}\,B_{01} + \underline{A_{14}}\,B_{02} + \underline{A_{13}}\,B_{03} + \cdots + \underline{A_{03}}\,B_{13} + \underline{A_{02}}\,B_{14}$

3 $P_{02} \leftarrow A_{02}\,B_{00} + A_{01}\,B_{01} + A_{00}\,B_{02} + \underline{A_{14}}\,B_{03} + \cdots + \underline{A_{04}}\,B_{13} + \underline{A_{03}}\,B_{14}$

4 $P_{03} \leftarrow A_{03}\,B_{00} + A_{02}\,B_{01} + A_{01}\,B_{02} + A_{00}\,B_{03} + \cdots + \underline{A_{05}}\,B_{13} + \underline{A_{04}}\,B_{14}$

$\cdots$

14 $P_{13} \leftarrow A_{13}\,B_{00} + A_{12}\,B_{01} + A_{11}\,B_{02} + A_{10}\,B_{03} + \cdots + A_{00}\,B_{13} + \underline{A_{14}}\,B_{14}$

15 $P_{14} \leftarrow A_{14}\,B_{00} + A_{13}\,B_{01} + A_{12}\,B_{02} + A_{11}\,B_{03} + \cdots + A_{01}\,B_{13} + A_{00}\,B_{14}$

Step 2 - Propagation and Reduction

16 $[P_{00}\ ,\ P_{01}] \leftarrow [P_{00} \mod 2^{17}\ ,\ P_{01} + P_{00} \gg 17]$

17 $[P_{01}\ ,\ P_{02}] \leftarrow [P_{01} \mod 2^{17}\ ,\ P_{02} + P_{01} \gg 17]$

18 $[P_{02}\ ,\ P_{03}] \leftarrow [P_{02} \mod 2^{17}\ ,\ P_{03} + P_{02} \gg 17]$

$\cdots$

28 $[P_{13}\ ,\ P_{14}] \leftarrow [P_{13} \mod 2^{17}\ ,\ P_{13} + P_{14} \gg 17]$

29 $[P_{14}\ ,\ P_{00}] \leftarrow [P_{14} \mod 2^{17}\ ,\ P_{00} + \underline{P_{14} \gg 17}]$

The multiplication step employs 15 DSP slices to execute multiply-and-accumulate operations in 16 cycles. Our implementation makes use of *in-place memory management* by subtle assignment of operands to DSP slices to execute the multiplication operations. This trick aims at minimising the cycles spent for operand accesses. We handcrafted the order of multiplications to fix one operand constant over the consecutive multiplications. Then, we load these constant operands to the slices' internal registers to keep them available. Hence, we limited the operand accesses to one without spending extra resources, and could issue new multiplications in consecutive cycles. The details of how the multiplications are implemented are given in the next paragraphs.

The multiplication inputs are divided into 17-bit word operands, which are processed with the DSP multipliers as shown in the line 1-15 of Algorithm 5. We assigned one DSP slice to each line for scanning the operands cross-wise and circularly while traversing these lines column-wise from left to right. This assignment is illustrated with the highlighted multiplications in the Algorithm 5, which are executed with the same DSP slice in consecutive cycles. With this assignment, the index of a multiplication operand at each DSP slice is fixed to either A_x or $\underline{A_x} = 19 \times A_x$, and we store these two values in the DSP slices' pipeline registers for input `A` (shown in Figure 2). This trick does not only eliminate spending extra FPGA resources for storing the A_x values, but also enables executing the multiplications with a single operand.

The first cycle of the multiplication step is spent for loading the A_x values to the `A1` pipeline registers, and the calculation of $19 \times A_x$ is started with the operands as `A1` registers and wiring 19

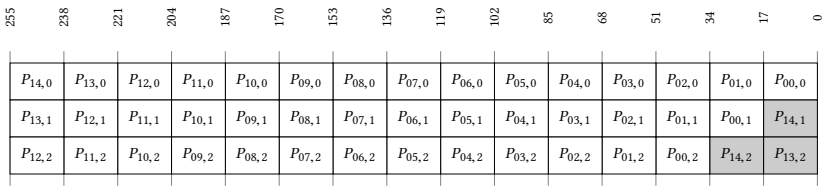

Fig. 3. To handle propagation of multiplication step's partial results stored in the DSP Slices' P registers $P_0, \cdots, P_{14}$, these results are divided into 17-bit words as $P_x = 2^{34}P_{x,2} + 2^{17}P_{x,1} + P_{x,0}$, aligned as shown, and tree words in each column are summed.

directly to slices' B input. In the next cycle, the first column of multiplications shown with lines 1-15 of Algorithm 5 are initiated. As this first column do not include a multiplication with $\underline{A}_x$, all multiplications are executed using the A_x stored in the A1 registers, while the B inputs are wired to B_{00}. Moreover, the result of executing $19 \times A_x$ operation becomes available at the P register in this cycle, and it is forwarded to the A1 register, and the A1's current value is forwarded to the A2 register. Therefore, at the end of this cycle, the A1 and A2 pipeline registers store the A_x and $\underline{A}_x$ values respectively. In the following cycles, the DSP slices are configured to execute the operation $P = A \times B + C$, by connecting the slices' C input to P output, to use the P register as an accumulator. Moreover, in accordance with the executed column of multiplications in the corresponding cycle, the A_x values are used from the A1 and A2 pipeline registers, and the B inputs are used to feed the multiplications with the B_x operands.

In summary, we perform the multiplication step in 16 cycles; the first cycle is used for loading the A registers and the next cycles are for multiply-and-accumulate. In addition to using DSP slices' pipeline registers for storing the A_x values, our use of the P registers for accumulating and storing the output eliminates the need for instantiating registers to store the intermediate or output values. In total, we save 770 registers by customising the use of DSP slices.

Reduction step follows the multiplications: the step first divides each partial multiplication result at the P registers into two, i.e. the lower 17-bits and all the higher bits. Later, it propagates the higher bits of a partial result to the lower bits of the next partial result, as shown with the lines 16-29 of Algorithm 5. In this scheme, the bits crossing the 255-bit boundary are propagated to the least significant side after multiplication with 19 for the modular reduction of the end result. Even though the reduction step is shown with 15 lines in the algorithm, we implemented the propagation to execute these lines in parallel by re-using the DSP slices. The implementation details of the reduction step is given in the next paragraph.

At the end of the multiplication step, the DSP slices' P registers hold the accumulated partial results which may reach up to 43-bits. Therefore, considering that 255-bit values are processed in 15 DSP slices each for a 17-bit word, these P register values are divided into three pieces of maximum 17-bit words, i.e. $P_x = 2^{34}P_{x,2} + 2^{17}P_{x,1} + P_{x,0}$. The alignment of these words are shown in Figure 3, where the vertical lines denote every 17th bit. In the figure, the grey boxes correspond to the words which overflow the 255-bit boundary and are moved to the least significant side after multiplication with 19 for modular reduction. The propagation is handled in two cycles to sum the words at each column in parallel using the DSP slices. In these cycles, the DSP slices are configured for $P = C + D \times B$ operation where the P registers are used as accumulator by connecting the C input to the P output, and the B registers are set to either 1 or 19. In the first cycle, the $P_{x,1}$ words of x'th partial result at P_x are summed with the $P_{x+1,0}$ word of P_{x+1}. In addition, the $P_{x,2}$ words are stored in 9-bit registers as they will be lost when the additions will update the P registers' value. In

the next cycle, the $P_{x,2}$ words of P_x which were stored in the 9-bit registers are propagated to the $P_{x+2,0}$ word of P_{x+2}.

Carry Propagation is the next step of the modular multiplication that handles the carry propagation between the words at the P registers if they occur after the two additions of the previous step. The propagation is handled in the same fashion as the reduction. Any bit that is greater than 17th bit of a P register is added to the least significant 17-bit word of the next P register. Once again, 15 DSP slices are used to execute all additions in parallel and complete the propagation of all words in a single cycle. However, any of these 15 additions may set a carry bit again, in which case the carry propagation cycle needs to be repeated. This step completes in 2.6 repetitions on average according to our tests with a thousand multiplications with random inputs. This part of our implementation results in non-constant-time execution; however, we offer a customisation to make it constant-time by repeating the propagation step even when there is no carry. This option will be described in Section 4.4.

Special cases may occur when the multiplication result is lower than 2^{255} while it is still greater than $q = 2^{255} - 19$, as the reduction cannot be addressed completely with the steps described in the paragraphs above. These cases require special care, for which we used the *pattern detection* functionality of the DSP slices, which basically compares the output value with a pre-defined bit pattern (shown in Figure 2). At such large numbers, the value of all the bits is one, except the least significant five bits which may vary between 0x0E and 0x1D. We made use of pattern detection with a pattern to check if the bits greater than the 5th are all ones. In addition, we used a 5-bit comparator for the least significant bits to check if the result is one of the special cases. To handle such cases, we spent an extra cycle to perform the $P = C + D \times B$ operation with C input is wired to zero, D to one and B is the difference between the least-significant five bits of the results and 0xD.

The special case reduction is, in fact, not always necessary, e.g. when the result will be processed by another mod q operation. In such cases, the latter operation should yield the reduction. This saves a lot in the case of scalar multiplication (described in Section 3) since it repeatedly uses the output of a mod q operation as an input to another. Therefore, we limited the handling of special cases, and spending an extra cycle for it, to the operations that determine the final result of the scalar multiplication. Moreover, we added an optional feature that executes the special case handling even when there is no need, to avoid potential variance in the execution time. We bound this feature to the same compiler directive used for the constant-time execution of carry propagation.

4.2.2 Mod q Addition and Subtraction. Both operations are executed similarly to the multiplication described in the previous subsection. The Addition or subtraction are handled with 15 DSP slices in parallel in a single cycle, followed by propagation of the partial results, and carry bits. In the first cycle, the DSP slices are configured for $P = C \pm D \times B$, where the input C and D are used for operands, and the input B is wired to 1. The reduction and carry propagation of the calculated partial results at the P registers are handled following the same scheme designed for mod q multiplication and described in Section 4.2.1.

4.2.3 Mod l Reduction. In addition to modulus q arithmetic operations, the reduction for modulus l is required for Ed25519 as mentioned at the beginning of Section 4.2. This subsection explains the details of how our arithmetic unit addresses the reduction operation following a similar fashion as mod q arithmetic operations.

The *mod l* reduction in the Ed25519 (Algorithm 3) receives an input x which is 512-bits at maximum that is SHA512's output length. The l is defined in [3] as a 253-bit prime $2^{252} + 27742317777372353535851937790883648493$. We implemented the reduction by repeating the equation $x = (x \ mod \ 252) - (x \gg 252) \times 27742317777372353535851937790883648493$ until no borrow

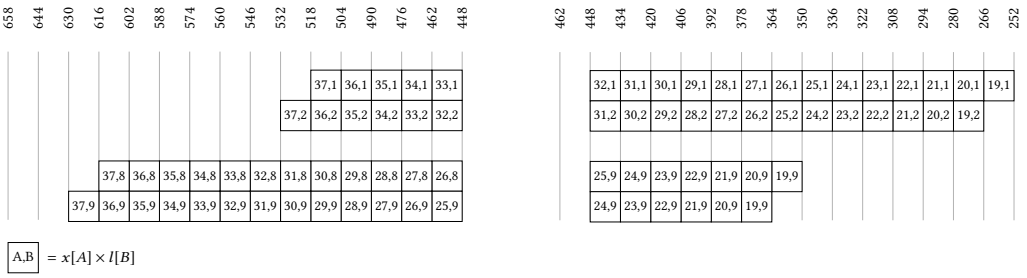

Fig. 4. The schoolbook multiplication for *mod* l reduction is handled with the shown alignment of the 14-bit word multiplications which are grouped into two for calculating the most and least significant parts of the $(x \gg 252) \times 27742317777372353535851937790883648493$ respectively with the multiplications shown at the left and right.

occurs at the end of the subtraction. Moreover, we preferred to process the input in 14-bit chunks as it simplifies the implementation since 14 divides 252-bits without remainder.

Towards our low-resource utilisation goal, we preferred to implement the reduction operation on the same arithmetic unit for modulus q to reuse the DSP slices. Therefore, considering the 15 DSP slices, we divided the operands into two parts, executing the reduction equation in two steps. Each step handles $14 \times 14 = 196$-bits with 14-bit words processed by 14 DSP slices. The 15th slice is allocated for calculating the carry/borrow value that will be propagated from the result of the first step to the second. Both steps of the reduction first execute school-book multiplication for $(x \gg 252) \times 27742317777372353535851937790883648493$ (shown in Figure 4), then subtraction of result from (x *mod* 252). In the first step, we calculate the part of these two consecutive operations that contributes to the first 196-bits of the output, while the most significant bits in the next step.

We processed the operands in DSP slices using similar techniques that we employed for the reduction to modulus q described in Section 4.2.1. An initialisation cycle is spent for storing the 14-bit words of the input that will be reduced in the A registers of the DSP slices. Next 9 cycles are spent for multiplication operations executed in the DSP slices with $P = C + A \times B$ configuration, where the P register is used as accumulator by connecting the input C to the output P, and the input B is assigned with 14-bit words of l starting with $l[1]$ in first cycle, reaching to $l[9]$ at the last. These 14-bit word multiplications handled in each cycle are shown in Figure 4 together with their bit-wise alignment. In the right side of Figure 4, the multiplications of the first step of the reduction are shown: the first row corresponds to the first cycle in which the parallel multiplications are handled for the first word of l with 14 words of x which are the words between 19 and 32. In the next cycle, the multiplications are handled with the second word of l with 13 words of x. And when the last cycle is reached only five multiplications are executed as the ninth word is the most significant word of l. At the end of the ninth cycle, the DSP slices' P registers accumulate the multiplication results for each column show in the right side of Figure 4, where the 15th accumulator is left empty as it has not employed any multiplications yet. The next cycle executes the subtraction of the accumulated results from the 14 least significant 14-bit words of the x, which is followed by propagation cycle for adding the bits overflowing the 14-bit boundary of each P register to the next. Again the propagation additions may result into a carry and in that case the cycle is repeated to propagate the carry bits as well. In this propagation scheme, the 14th DSP slice's propagation is done to the empty accumulator of the 15th DSP slice, which is the contribution of the first step of the reduction to the next as either carry or borrow bits. When the described first step of the

reduction equation is executed, the least significant 196-bit of the result is calculated as 14-bit words stored in the P registers, and become present at the output port of the arithmetic unit as a partial result of the reduction.

In the next step of the reduction, the same scheme of the first step described in the above paragraph is followed. The 14-bit word multiplications are handled for the words shown in the left side of Figure 4 with the corresponding alignment. Following the multiplications, the values at the 15 accumulators are subtracted from the minuend which is the five 14-bit words of x that corresponds to the bits between 196th and 252nd, prepended with zeros. In the next cycle, the propagations are handled. When this second step is completed, 14-bit words at the output of 15 DSP slices yield the 210-bit second partial result of the reduction.

Processing the reduction equation with the above described two steps provide partial outputs which yield a 406-bit result when concatenated using the align block described in Section 4.1. However, it is not the final result that reduces an input to modulus l as long as there is a set bit that is more significant than 252nd. Therefore, the given reduction equation is executed three times for completing the reduction to satisfy this demand for 512-bit input size.

4.2.4 Non-modular Multiplication. The multiplication is implemented for operands of 255-bit length, calculating a 510-bit output. The implementation follows a similar structure as the modulus q multiplication that processes the operands in 17-bit words as described in Section 4.2.1, excluding the reduction. Moreover, the multiplication is divided into two steps to calculate the result partially, similar to the two-step implementation described in Section 4.2.3. At the first step, 14 DSP are used to calculate the least significant $14 \times 17 = 238$-bits of the result, while the 15th calculates the propagation from the first step of the multiplication to the next. In the second step 16-DSP slices are used to calculate the most significant $16 \times 17 = 272$-bits of the result. The final result is constructed with 30 words which can be processed in two steps with 15 DSP slices. However, we preferred spending an extra slice, for the accumulation of carry from the first step to the next. It helps keeping the existing 17-bit propagation pattern and avoid instantiating extra multiplexers at the connections between the DSP slices. Therefore, we utilised an extra DSP slice in the second step, which is dedicated to non-modular multiplication operation only, and is kept idle in the execution of the other arithmetic operations described in previous subsections.

4.3 Hash Unit

We re-used an existing SHA512 implementation which complies with FIPS 180-4 Secure Hash Standard. Its implementation is prudent on utilising logic elements and calculates the hash in 80 cycles for 1024-bit input message blocks. Padder is used as a helper module that appends the pad field of SHA512 to the received hash input data. Following the fetch of four or less than four (256-bit) input message blocks, consecutive operations are executed to read the $Hash_{IN}$ data from memory to RegO and RegS, execute the hash module and write the $Hash_{OUT}$ data to the memory. For the first execution, the initial hash value pre-loaded to the memory at design time is used as $Hash_{IN}$. For the next executions the previously calculated $Hash_{OUT}$ is used.

4.4 Side-Channel Resistance Features

Cryptography is used to protect secrets and its security relies on the secrecy of the keys. Moreover, the security requires not only strong algorithms but also implementations that resist side-channel attacks. Indeed, secrets can leak unintentionally through variations in execution time, differences in power consumption or electromagnetic radiation, to name a few. Thus, we provide our design with two countermeasures offering resistance against side-channel attacks. These countermeasures aim to make such attacks difficult to carry out and they are explained in the following paragraphs.

Constant time execution is required to defend against timing attacks. It is a prominent attack on algorithms which use conditional operations that use secrets (e.g., key) to decide on taking the branch. The attack is performed by measuring the execution time differences of the executed operation, to deduce the secret. Though it is a long-standing attack vector, it is often powerful exploit. For example, [5] has mounted an attack against Montgomery ladder for scalar multiplication, and achieved full key recovery from a TLS server.

In our design, the used algorithms Algorithm 1 and Algorithm 3 have neither branch nor lookup operations depending on the secret values. Moreover, we do not introduce any condition-based operation for their implementation except our carry propagation mechanism described in the Section 4.2. Varying number of cycles to complete the propagation might be an undesired practice. Hence, we defined a Verilog compiler directive (that will be introduced in Section 4.5) to enable a resistance feature if needed. This feature enforces executing the propagation step for the worst case. It thwarts the timing attacks by repeating carry propagation even when there is no carry. As a result, the propagation step requires 16 cycles instead of 2.6.

Z-coordinate Randomization [6] is a protection mechanism for hardening the implementation against differential power analysis attacks. As the key is an input to the scalar multiplication function, these attacks aim at capturing power traces of various multiplications and deduce information about the key by analysing them. The first example of this attack that recovers a smart card's secret key was shown by [11] in 1998. The next year, [6] showed the attack on elliptic curves and proposed countermeasures, including Z-coordinate randomisation. This countermeasure applies randomisation to the projective coordinates, which masks the trace data with the randomness. Simply, a random value is picked for the Z-coordinate, and it is multiplied with all other coordinate components before the multiplication. The transformation is inverted naturally when the used projective coordinates are transferred back to affine coordinates after following a scalar multiplication. Hence, this is a cheap countermeasure that requires only few extra multiplications for the scalar multiplication for each coordinate system.

Base point blinding [6] is countermeasure that relies on the randomisation of the base point. The countermeasure applies a transformation to the curve's base point for the masking, and inverts it to obtain the intended multiplication result. In the implementation of this countermeasure, a random point R is used to randomise the base point B into $B+R$. The scalar multiplication to compute kB is then performed in two steps. First two multiplications $k(B + R)$ and kR are performed, then their difference is taken. The biggest overhead of this countermeasure is that it requires executing two scalar multiplications instead of one.

The blinding of the base-point B with R requires randomness, the generation of which is a standalone problem. To address the problem, our design requires a random number as input and generates R from it. We reserved an address in the data memory for this number and defined a command that receives the randomness input, creates a point from it and stores the point in the memory. In addition to that, we attached to our design a True Random Number Generator (TRNG) [17] that generates unpredictable bit sequences based on physical noise on a ring oscillator. While this TRNG costs only a little hardware resource utilisation (30 LUTs and 82 Registers), it generates high quality randomness.

4.5 Design Customisation

We also offer five Verilog *Compiler Directives* for enabling or disabling additional features. These customisations offer trade-offs to optimise our design for functionality, security or performance. Three of these directives are to enable the side-channel countermeasures described in Section 4.4, and the other two are explained in the paragraphs below. The evaluation of these trade-offs are given in the next section.

We offer a directive to select implementing either Ed25519 and X25519 together on the same hardware, or X25519 individually. When they are both enabled, X25519 is supported natively with the operations implemented for Ed25519. This selection does not have an adverse effect over Ed25519 performance; however, it affects X25519 badly. Because, it is moved to the twisted-Edwards Curve, leaving the benefits of its Montgomery curve implementation, and paying the penalty of coordinate conversions. In contrast, if the directive is set to implement the standalone X25519 implementation, greater key generation performance is offered, as well as significantly low utilisation. Moreover, these results are highly comparable to the related works of X25519.

Another customisation offers the benefits of Vivado's *full_case* synthesis attribute [21] to reduce the resource utilisation. This attribute indicates that all possible case values are specified in a case-statement. Therefore, the case-statement is synthesised only for the specified cases, but no extra logic is created for the unspecified i.e. default cases. In our experimentation, we found that it offers significant resource utilisation improvement in parallel to the increasing design complexity, which will be highlighted in the next section with utilisation results. However, it should be noted that this customisation makes the design vulnerable against attackers who have an ability of performing fault attacks [9]. Such attacks aim at flipping at least one-bit of a signal connected to the control unit's state machine. As a result, an unspecified state or state transition is forced, and the circuit shows an undefined behaviour which may leak critical information. Moreover, [16] already shows that such an attack is practical against EdDSA. Therefore, the resource utilisation benefits of this customisation is limited to the attacker models which do not involve the ability to perform such physical attacks.

5 RESULTS AND EVALUATION

This section presents the resource utilisation and performance results of our implementation, highlighting the effects of each optional feature. Moreover, we provide comparison of our results with the related works to evaluate our design decisions. However, this comparison focuses mostly on the X25519-only operation since yet there is no Ed25519 hardware implementation published.

5.1 Resource Utilisation

Our Ed25519 implementation's resource utilisation is given in the upper half of the Table 1 starting with an area-optimised design, showing the cost of enabling the features in the respective order. In the second half of the table, our design is compared to the related works with its X25519-only configuration. All results in the table for both our implementation and related works are reported for Xilinx Zynq SoC.

The resource utilisation results given in Table 1 show that Ed25519 has a large cost compared to X25519. The overhead is mostly because of the support for a secondary modulus operation on the arithmetic unit and the SHA512 hash. The instruction storage contributes to LUT utilisation as distributed ROM, which is minimised for X25519-only option. Moreover, our implementation's register utilisation is always lower than the related works, even when it supports the whole Ed25519 algorithm, which is an outcome of our design strategies described in Section 4.

The hash hardware occupies the 25% of base design's implementation, dominating the REG utilisation that is 60% of the whole. Therefore, that module is specifically indicated in the table with an extra row. We would like to emphasise that an existing implementation was used for the hash and it was excluded from our optimisation work. Moreover, Ed25519 can be implemented with a lightweight hash implementation instead of SHA512, which might be preferable for resource constrained IoT nodes.

The table also shows the additive cost of security features to our design. An interesting result is the cost added by the implementation of *default cases* to avoid visiting undefined states at a bug

Table 1. Utilisation and performance comparison of our design's various configurations and related works. All results are given for the same FPGA architecture. (The overhead of SHA512 hardware that is included in all Ed25519 configurations is shown in a separate row with its individual resource utilisation.)

Implementation	Utilisation					Kilo Cycle Cost / Ops per sec			Freq. (MHz)
	Slices	LUT	REG	DSP	BRAM	Sign*	Verify*	Key Gen.	
Ed25519: *Our optimised implementation that also supports X25519 and the overhead of each security feature*									
Optimised	3 204	11 148	2 656	16	0	132 / 621.2	301 / 272.4	173 / 474.0	82
Default Cases	4 128	14 590	2 656	16	0	132 / 621.2	301 / 272.4	173 / 474.0	82
Constant Time	3 264	11 310	2 656	16	0	271 / 302.6	668 / 122.8	355 / 230.9	82
Z-coord. Random.	3 234	11 167	2 656	16	0	132 / 621.2	301 / 272.4	173 / 474.0	82
Basepoint Blinding	4 261	15 249	2 656	16	0	350 / 234.3	534 / 153.6	458 / 178.8	82
All Four Above	4 303	15 276	2 656	16	0	690 / 118.8	1050 / 78.1	904 / 90.7	82
SHA512	*798*	*3 157*	*1 615*						
X25519: *Our implementation optimised only for X25519 (disabling Ed25519) and the related works*									
Ours	775	2 707	962	15	0	~	~	75.0 / 1400.0	105
[18] - Single Core	1 029	3 592	2 783	20	2	~	~	79.4 / 2518.9	200
[18] - Multi Core	11 277	43 875	34 009	220	22	~	~	34.0 / 2941.2	100
[12]	8 639	21 107	26 483	260	0	~	~	13.6 / 8431.7	115
[13]	6 161	17 939	21 077	175	0	~	~	10.5 / 10989.0	115

* Reported for 1024-bit (128-byte) message

or fault attack. The former is avoidable by wise implementation practices, but the latter depends on the users' attacker model. In our implementation, preventing the design from an attacker's capability to flip some bits [9] requires spending 30% more LUTs.

The countermeasures against side-channel attacks have only a slight effect on the LUT utilisation in comparison to the default-cases. This is an expected result in fact. The constant-time execution feature only changes the condition checks for finalising the propagation step (as described in Section 4.2). Z-coordinate randomisation adds extra multiplications within the state-machine. Base-point blinding feature adds only extra instructions in the ROM.

5.2 Performance

The performance results given in Table 1 allow various deductions, such as the cycle cost of the cryptographic functions and the performance overhead of security features. Moreover, the cycle costs of all operations are presented in Table 2 where the first three entries correspond to the functions that depend on the modular arithmetic operations given below them.

The dominance of scalar multiplication over the execution time of all cryptographic operations becomes obvious when the two tables are read together. In summary, signature verification takes twice the amount of cycles to complete compared to signing, because it demands executing two scalar multiplications (see Algorithm 3). Similarly, the execution time increases significantly when the blinding feature is enabled as it introduces an extra multiplication (as described in Section 4.4).

The constant-time execution feature (optionally) strengthens the implementation against the attackers' ability to exploit side-channels in the form of performing timing attacks. As a trade-off, enabling this feature has an adverse effect on performance as it forces the arithmetic operations to repeat the carry propagation cycle even when there is no carry. Enabling this feature affects all the arithmetic operations underlying the scalar multiplication. Hence, the cost of this protection is slightly greater than doubling the execution time.

Table 2. Cycle Cost of Operations

Implementation	Cycle Overhead	
	Optimised	Const. Time
Scalar Mult. for Ed25519	120 260	250 821
Scalar Mult. for X25519-only	63 890	135 446
Modulus q Inversion	10 786	18 489
Modulus q Multiplication	18.6	33
Modulus q Add/Sub	3.6	17
Modulus l Reduction	38	99
Non-modulus Multiplication	39	66
SHA512*	80	80

* Execution of a single round for a 1024-bit chunk of input

The complexity increase of Ed25519 with a compact hardware implementation limits the maximum clock frequency to 82 MHz, while the X25519-only implementation achieves 105 MHz. Moreover, the X25519 operation requires more than twice the amount of cycles when implemented on the Ed25519's twisted Edwards curve. This is because of, first, leaving the optimised scalar multiplication on the Montgomery curve with restricted-X coordinates, and secondly paying the penalty of coordinate conversion between two curve coordinates.

For comparing the performance of our design to executing the same operations on software, we turned it into an elementary hardware accelerator on the Zynq SoC. For this purpose, first we wrote an AXI IP core wrapper to interface with our module from the ARM processors of the device. We also set a basic address mapping for the IP core's inputs and outputs. Later, we implemented functions in software to send messages to hardware to make them signed, and later to make the signatures verified. Finally, we compared execution time of these operations with the performance of TweetNaCl [4] and Ed25519-donna software implementations. TweetNaCl aims at a very small code length for the whole NaCl to fit into a hundred tweets, and it has a poor performance. Thus our design achieves more than 100 times better execution time results than it at minimum. However, when we compare with the performant ed25519-donna implementation, the minimum improvement reduces to 1.75 times for processing one-byte message, and converges to 2.5 with the increasing message size. The result is, of course, open to improvements as it is comparison against Zynq's hardcoded ARM cores that are powerful processors with a fast clock, and running the software baremetal. Moreover, our simple test setup did not benefit an optimised interface between hardware and software. Such interface is known to introduce a great communication overhead, mostly for handshaking and cache-coherency requirements [20]. Hence, designing an optimised interface, e.g. making use of fast data moving DMAs, is an individual design problem left out of this work's scope.

5.3 Comparison to Related Work

To the best of our knowledge, there exists no Ed25519 hardware implementation published. Therefore, we compare our design decisions with related works [12, 13, 18], by configuring our design for the X25519-only implementation.

Unfortunately, comparing FPGA implementations with overall area utilisation is not possible, because the implementations are divided into different type of resources. Moreover, the fair benchmarking only with Table 1 is not very easy, considering that our work aims at achieving low resource utilisation while the related works opt to performance optimisations. However, each work in this table targets the same FPGA architecture; therefore, their judicious comparison is possible.

Our resource utilisation is lower for every type of resource when compared to the related works. Even at the minimum, our register utilisation is almost only one third the related works. The economy of our implementation in register utilisation has been possible with our handcrafted DSP slice use. The arithmetic unit benefits in-place memory management trick by storing the operands in the slices' pipeline registers. Besides registers, our design is carefully considered on the use of other precious resources, i.e. DSP and Block RAM.

Our X25519-only implementation uses the same Montgomery ladder as the given related works. However, the execution times vary with the difference in how the arithmetic operations are executed in hardware. Our work requires a comparable number of clock cycles for generating a key. However, the constant-time execution has adverse effects, as it is shown in Table 2. Also, the maximum clock frequency of our design is lower than the others, which is almost always the cost of aiming at low resource utilisation.

In the Table 1, the single core implementation of Sasdrich et al. [18] achieves an efficient design with great performance compared to its resource requirements. That is mostly made possible by high clock frequency support with using BRAM slices. In comparison, we did not preferred distributed memory for compactness combining two moduli on the same hardware, as discussed in Section 4.1. The multi-core implementation of their work gives poor performance improvement for the added resource consumption due to its adverse effects on the clock frequency. The work of Koppermann et al. [12] achieves high performance without a high clock frequency, but at the expense of greater resource utilisation. Moreover, its improved version [13] achieves a better performance with lower utilisation number, which is the results of their optimisation strategies.

The optional security features reduce the performance over utilisation ratio, since they either reduce the performance or increase the hardware cost for the sake of reaching a secure design. As mentioned in Section 2, the related works also support certain countermeasures. For instance, random projective coordinates are supported by each work. It is inherently supported by [13], and [19] adds the support to its predecessor [18]. However, this countermeasure has only a slight effect over the performance and resource utilisation, so it is almost invisible in the comparison of the results. Moreover, [19] benefits two masking techniques, namely random scalar blinding and memory address scrambling. They are supported with minor overheads to the overall design. In contrast, we opt for base point blinding, which has a relatively big overhead over the execution time. The comparisons of these countermeasures are also not directly possible due to different design decisions. Our design combines X25519 and Ed25519 in a single implementation with area optimisation, while [19] minds an efficient implementation of only X25519.

6 CONCLUSION

In this work, the first hardware implementation of Ed25519 digital signature algorithm is presented. The design also offers X25519 key establishment functionality, either on the Ed25519 implementation or with a dedicated optimisation for it. We showed that our design strategies surpass similar works in resource utilisation on the same FPGA architecture. Moreover, we provide optional security features on top of the optimised base implementation to strengthen it against various attacker capabilities. In addition to offering such customisability, we also show the additional cost of enabling such security features to give an evaluation of the trade-offs.

ACKNOWLEDGMENTS

This work was supported in part by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89), the KU Leuven Research Council through C16/15/058, and ERC Advanced Grant 695305.

REFERENCES

[1] 2017. Estimated value of the North American smart home market from 2012 to 2021 (in Billion U.S. Dollars). (Jul 2017). Retrieved from https://www.statista.com/statistics/296113/north-america-smart-home-market-revenue/.

[2] Daniel J Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*. Springer, 207–228.

[3] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of Cryptographic Engineering* 2, 2 (2012), 77–89.

[4] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. 2014. TweetNaCl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 64–83.

[5] Billy Bob Brumley and Nicola Tuveri. 2011. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security*. Springer, 355–371.

[6] Jean-Sébastien Coron. 1999. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems*. Springer, 725–725.

[7] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. 2005. Guide to elliptic curve cryptography. *Computing Reviews* 46, 1 (2005), 13.

[8] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Edward Dawson. 2008. Twisted Edwards Curves Revisited. In *Asiacrypt*, Vol. 5350. Springer, 326–343.

[9] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (1997), 75–82.

[10] Simon Josefsson and Ilari Liusvaara. 2017. Edwards-curve digital signature algorithm (eddsa). (2017). No. RFC 8032. 2017.

[11] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1998. Introduction to Differential Power Analysis and Related Attacks. (1998).

[12] Philipp Koppermann, Fabrizio De Santis, Johann Heyszl, and Georg Sigl. 2016. X25519 Hardware Implementation for Low-Latency Applications. In *Digital System Design (DSD), 2016 Euromicro Conference on*. IEEE, 99–106.

[13] Philipp Koppermann, Fabrizio De Santis, Johann Heyszl, and Georg Sigl. 2017. Low-latency X25519 hardware implementation: breaking the 100 microseconds barrier. *Microprocessors and Microsystems* 52 (2017), 491–497.

[14] Peter L Montgomery. 1987. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation* 48, 177 (1987), 243–264.

[15] Joost Renes and Benjamin Smith. 2017. qDSA: Small and Secure Digital Signatures with Curve-based Diffie–Hellman Key Pairs. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 273–302.

[16] Yolan Romailler and Sylvain Pelissier. 2017. Practical fault attack against the Ed25519 and EdDSA signature schemes. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2017 Workshop on*. IEEE, 17–24.

[17] Vladimir Rozic, Bohan Yang, Wim Dehaene, and Ingrid Verbauwhede. 2015. Highly efficient entropy extraction for true random number generators on FPGAs. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 1–6.

[18] Pascal Sasdrich and Tim Güneysu. 2014. Efficient Elliptic-Curve Cryptography Using Curve25519 on Reconfigurable Devices. *ARC* 8405 (2014), 25–36.

[19] Pascal Sasdrich and Tim Güneysu. 2015. Implementing Curve25519 for Side-Channel–Protected Elliptic Curve Cryptography. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 9, 1 (2015), 3.

[20] Furkan Turan, Ruan De Clercq, Pieter Maene, Oscar Reparaz, and Ingrid Verbauwhede. 2016. Hardware Acceleration of a Software-based VPN. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 1–9.

[21] Xilinx 2012. *Vivado Design Suite User Guide: Synthesis*. Xilinx. v2012.2.

[22] Xilinx 2014. *7 Series DSP48E1 Slice*. Xilinx. v1.8.

[23] Xilinx 2015. *Zynq-7000 All Programmable SoC Technical Reference Manual*. Xilinx. v1.10.

Received February 2018; revised August 2018