VLSI Implementation of Double-Base Scalar Multiplication on a Twisted Edwards Curve with an Efficiently Computable Endomorphism

Zhe Liu[1], Husen Wang[2], Johann Großschädl[1], Zhi Hu[3], and Ingrid Verbauwhede[1]

[1] LACS, University of Luxembourg, Luxembourg
`{zhe.liu,johann.groszschaedl}@uni.lu`
[2] ESAT/COSIC and iMinds, KU Leuven, Belgium
`{husen.wang,ingrid.verbauwhede}@esat.kuleuven.be`
[3] Central South University, P.R. China
`huzhi_math@csu.edu.cn`

Abstract. The verification of an ECDSA signature requires a double-base scalar multiplication, an operation of the form $k \cdot G + l \cdot Q$ where G is a generator of a large elliptic curve group of prime order n, Q is an arbitrary element of said group, and k, l are two integers in the range of $[1, n-1]$. We introduce in this paper an area-optimized VLSI design of a Prime-Field Arithmetic Unit (PFAU) that can serve as a loosely-coupled or tightly-coupled hardware accelerator in a system-on-chip to speed up the execution of double-base scalar multiplication. Our design is optimized for twisted Edwards curves with an efficiently computable endomorphism that allows one to reduce the number of point doublings by some 50% compared to a conventional implementation. An example for such a special curve is $-x^2 + y^2 = 1 + x^2y^2$ over the 207-bit prime field $\mathbb{F}_p$ with $p = 2^{207} - 5131$. The PFAU prototype we describe in this paper features a (16×16)-bit multiplier and has an overall silicon area of 5821 gates when synthesized with a 0.13μ standard-cell library. It can be clocked with a frequency of up to 50 MHz and is capable to perform a constant-time multiplication in the mentioned 207-bit prime field in only 198 clock cycles. A complete double-base scalar multiplication has an execution time of some 365k cycles and requires the pre-computation of 15 points. Our design supports many trade-offs between performance and RAM requirements, which is a highly desirable property for future Internet-of-Things (IoT) applications.

Keywords: Signature verification, Twisted Edwards curve, Endomorphism, Multiple-precision arithmetic, Pseudo-Mersenne prime

1 Introduction

Digital signatures are an indispensable component of modern security protocols like TLS [6], where they are used to authenticate the server and optionally the client too. More specifically, TLS can provide server authentication by means of a certificate that binds an identity (e.g. the server's domain name) to a public key. The certificate contains besides the ID and public key also a collection of attributes, all of which is signed by a trusted third party called Certification Authority (CA). In the initial (i.e. handshake) phase of the TLS protocol, the client normally requests the server's certificate, and if he manages to verify the signature of the CA successfully, he is assured that the public key contained in the certificate is authentic and indeed belongs to the server. The next step is then to establish a shared secret between client and server, which can be done through either RSA-based key transport or via Diffie-Hellman key exchange. In any case, without prior authentication, the key establishment process would be vulnerable to a classical Man-In-The-Middle (MITM) attack. The same holds for Datagram TLS (DTLS) [25], a variant of TLS optimized for connectionless datagram transport (i.e. UDP) that is widely considered as the future standard protocol for securing the Internet of Things (IoT) [16].

The signature algorithms supported by the most recent version (i.e. version 1.2) of TLS are RSA [26], DSA [22], as well as ECDSA [14] through a separate RFC [3]. In order to validate the server's certificate (or its chain of certificates [6]), the client needs to verify one or more signatures. Using RSA as signature algorithm has the advantage that verification is relatively fast since it involves an exponentiation by the public exponent, which is usually small (e.g. $2^{16}+1$) [26]. However, the downside of RSA signatures is their size and accompanying memory and bandwidth requirements, especially at higher security levels. This can be exemplified with [30], where RSA with a modulus length of 3248 bits is recommended to match the security of 128-bit AES, which means an RSA signature has a length of 406 bytes. Signatures of such a size can pose a problem for resource-constrained IoT devices that often have only a few hundred bytes to a few kB of RAM. On the other hand, Elliptic Curve Cryptography (ECC) is well-known for its compact key and signature sizes, which is a highly desirable feature for resource-limited devices. For example, a 255-bit ECDSA signature (matching the security of 128-bit AES) has a size of merely 64 bytes when it is compressed [2], i.e. less than one sixth of the RSA signature size. However, an inherent problem with ECDSA signatures is that, despite their small size, the verification process is relatively computation-intensive.

The verification of an ECDSA signature requires one to perform a double-base scalar multiplication, an operation of the form $k \cdot G + l \cdot Q$, where G is a point on an elliptic curve E that generates a large group of prime order n, Q is an (arbitrary) element of this group, and k and l are two integers in the range of $[1, n-1]$ [14]. Normally, $k \cdot G + l \cdot Q$ is computed in a simultaneous fashion (i.e. with joint doublings) so that at most m doublings need to be executed in total, where m is the bitlength of n [11]. Most previous attempts to reduce the execution time of this operation fall into one of two categories, namely, on the one hand, approaches that aim at minimizing the cost of a single point addition or doubling, and, on the other hand, techniques to reduce the number of these operations. An example for the former is EdDSA [2], a signature scheme based on a twisted Edwards curve [1] that allows for a more efficient implementation of the point arithmetic than a basic Weierstraß curve. Using a window method to reduce the number of point additions in a double-base scalar multiplication (as described in [11, p. 109f]) falls into the second category. Another option to cut down the number of point operations is to exploit an efficiently computable endomorphism as explained in [10] for variable-base scalar multiplication. Also a combination of both approaches, namely using the twisted Edwards addition law on so-called GLS [9] and GLV-GLS [20] curves (both of which are defined over $\mathbb{F}_{p^2}$ and possess endomorphisms) has been investigated in [19] and [7].

In this paper we introduce families of twisted Edwards curves with an efficiently computable endomorphism ϕ and demonstrate how said endomorphism can be used to speed up the ECDSA verification process, whereby we focus on hardware implementation of double-base scalar multiplication. In particular, we will study implementation properties of the twisted Edwards curve

$$E_T: \ -x^2 + y^2 = 1 + x^2y^2 \tag{1}$$

(i.e. $a = -1$ and $d = 1$) over a prime field $\mathbb{F}_p$, which is birationally equivalent over $\mathbb{F}_p$ to a so-called Gallant-Lambert-Vanstone (GLV) curve [10] of the form $E_W: \ y^2 = x^3 + ax$ (i.e. $b = 0$). Gallant et al. [10] were the first to describe how an efficiently-computable endomorphism ϕ can be used to speed up a variable-base scalar multiplication on such curves. While the exploitation of an endomorphism is independent of the curve representation, we are, to the best of our knowledge, the first to provide an explicit formula for the computation of said endomorphism on a twisted Edwards curve with $a = -1$ and $d = 1$. If $P = (x, y)$ is a point on the curve E_T given by Equation (1), then we can compute $\phi(P) = (\alpha x, 1/y)$ where α is an element of order 4 in the underlying field $\mathbb{F}_p$. The endomorphism ϕ acting on the twisted Edwards curve E_T is slightly more costly than its counterpart on the corresponding GLV curve E_W (because it requires the inversion of y), but can still be computed efficiently enough to yield a significant speed-up in practice, as we will show in this paper. In order to accelerate a scalar multiplication $k \cdot P$, the scalar k has to be split up into two "half-length" parts k_1 and k_2 (as explained in e.g. [10]), and then we can compute $k \cdot P = k_1 \cdot P + k_2 \cdot \phi(P)$ in a simultaneous fashion, which saves roughly 50% of the point doublings compared to a straightforward computation of $k \cdot P$.

While most of the previous work on exploiting endomorphisms has focussed primarily on variable-base scalar multiplication (such as needed in ECDH key exchange), we direct our attention to the double-base scalar multiplication carried out in the verification of an ECDSA signature. When taking advantage of ϕ, an m-bit double-base scalar multiplication $k \cdot G + l \cdot Q$ can be performed via four simultaneous half-length (i.e. roughly $m/2$-bit) scalar multiplications of the form $k_1 \cdot G + k_2 \cdot \phi(G) + l_1 \cdot Q + l_2 \cdot \phi(Q)$. Using a twisted Edwards curve with an efficiently computable endomorphism compares very favorably with related approaches, which are, on the one side, conventional GLV and conventional twisted Edwards curves, and, on the other side, GLS and GLV-GLS curves. When compared with conventional GLV curves given by a Weierstrass equation with e.g. $a = 0$ or $b = 0$, our curve has the advantage of a faster (and complete) addition law[4]. The major advantage of our curve over a conventional twisted Edwards curve is the existence of an efficiently computable endomorphism. On the other hand, when compared with GLS or GLV-GLS curves, our curve E_T has the advantage that it is defined over a conventional prime field $\mathbb{F}_p$ and not over a quadratic extension field $\mathbb{F}_{p^2}$. Extension fields are rarely supported by commodity cryptographic libraries, which hampers the use of GLS or GLV-GLS curves in real-world applications. Furthermore, our curve has the virtue of being more resource-friendly since it requires only arithmetic modulo p, whereas $\mathbb{F}_{p^2}$ involves both base-field (i.e. modulo p) arithmetic and extension-field arithmetic. Finally, it has to be taken into account that exploiting the 4-way endomorphism on GLV-GLS curves for double-base scalar multiplication is very expensive in terms of memory. Namely, when using the 4-way endomorphism, an m-bit double-base scalar multiplication is performed through eight simultaneous quarter-length (i.e. roughly $m/4$-bit) scalar multiplications, which requires the pre-computation and storage of (at least) 128 points. Since a single point typically occupies between 40 and 64 bytes in memory, this approach is not suitable for resource-constrained IoT devices.

The real-world benefit of our curve is the multitude of implementation options and trade-offs between execution time and silicon area (when thinking about hardware implementation) or memory footprint (in the context of software implementation) it supports. Providing many options and trade-offs is particulary important for cryptographic schemes to be used in the Internet of Things (IoT) since IoT devices come in all shapes and sizes, and have, therefore, varying resource constraints. At one end of the spectrum are devices with extreme restrictions (e.g. RFID tags, sensor nodes) where every single gate and very single byte counts. At the other end of the spectrum are devices equipped with powerful 32-bit processors and plenty of resources. Our curve offers many implementation options that allow a designer to fine-tune an implementation according to the requirements at hand. For example, when resources are constrained, one can perform a double-based scalar multiplication in the straightforward fashion by computing two simultaneous m-bit scalar multiplication, which is very economic in terms of memory. On the other hand, if more resources are available, our curve allows the designer to trade performance for memory or area (depending on whether the implementation is in software or hardware) by exploiting the efficiently-computable endomorphism. Finally, if resources are plenty, it is possible to achieve further speed-ups by combining the endomorphism with a window method for simultaneous scalar multiplication.

[4] Note that, as explained in [12], the twisted Edwards addition law can be complete even if a is a non-square or d is a square in $\mathbb{F}_p$, provided that the base point has odd order. A complete addition law is a nice feature for signature generation because it simplifies the integration of countermeasures against physical attacks. However, we do not discuss issues related to signature generation further since the focus of this paper is on signature verification. In essence, the fixed-based scalar multiplication needed for a signature generation can be performed on our curve in exactly the same way as on any other twisted Edwards curve (e.g. via a regular comb method [18]).

2 Twisted Edwards Curves with Endomorphism

2.1 Twisted Edwards Curve

Twisted Edwards curves were introduced to cryptography in 2008 [1] and are nowadays well established and become increasingly adopted in practical applications due to their efficient addition law. Let $\mathbb{F}_p$ be a prime field with $p > 3$. A twisted Edwards curve over $\mathbb{F}_p$ can be defined as

$$E_{a,d} : ax^2 + y^2 = 1 + dx^2y^2, \tag{2}$$

where a and d satisfy $ad(a-d) \neq 0$. As specified in [1], the j-invariant of $E_{a,d}$ is

$$j(E_{a,d}) = \frac{16(a^2 + 14ad + d^2)^3}{ad(a-d)^4}.$$

There is a remarkable addition law on twisted Edwards curves, which can be complete when a is a square and d a non-square in $\mathbb{F}_p$ [1]. Here, completeness means the addition produces the correct result for any two points (even if one of the points is the neutral element $\mathcal{O} = (0,1)$) on $E_{a,d}$, without exception.

2.2 GLV Method

Gallant, Lambert, and Vanstone [10] described in 2001 a new method (the so-called GLV method) for speeding up point multiplication on certain classes of elliptic curves, namely curves with an efficiently computable endomorphism. Let E be an elliptic curve over a finite field $\mathbb{F}_p$ and let $G \in E(\mathbb{F}_p)$ have prime order n. Assuming that an efficiently computable endomorphism ϕ on E exists so that $\phi(G) = [\lambda]G \in \langle G \rangle$, the GLV method replaces the computation $[k]G$ by a multi-scalar multiplication of the form $[k_1]G + [k_2]\phi(G)$, where the sub-scalars $|k_1|, |k_2| \approx r^{1/2}$. Since the number of doublings is halved, this method potentially allows for significant speedup of the point multiplication.

Gallant et al. described in [10] several families of curves featuring an efficiently computable endomorphism derived from special Complex Multiplication (CM). Let ϕ be a complex number and K be the extension field $\mathbb{Q}(\phi)$. If such an elliptic curve admits a complex multiplication by ϕ, then by [5, Thm 10.14] we obtain an endomorphism $\phi(x,y) = (\phi^{-2}\frac{f(x)}{g(x)},\ y\phi^{-3}\left(\frac{f(x)}{g(x)}\right)')$ and $\phi(\mathcal{O}) = \mathcal{O}$, where f, g are polynomial functions over $\mathbb{Q}$ with $\deg f = N_{K/\mathbb{Q}}(\phi)$ and $\deg g = N_{K/\mathbb{Q}}(\phi) - 1$ (here $N_{K/\mathbb{Q}}(\cdot)$ is the norm function from K to $\mathbb{Q}$).

2.3 Efficient Endomorphism on Twisted Edwards Curve

The twisted Edwards curve $E_{a,d} : ax^2 + y^2 = 1 + dx^2y^2$ is birationally equivalent to some short Weierstrass curve $E_s : y^2 = x^3 + a_sx + b_s$, whereby the birational equivalence map can be given as

$$\psi \ : E_{a,d} \to E_s, (x_t, y_t) \to (x_s, y_s) = (\frac{c_1(1+y_t)}{1-y_t} + c_2, \frac{c_1(1+y_t)}{x_t(1-y_t)}), \quad (3)$$

$$\psi^{-1} : E_s \to E_{a,d}, (x_s, y_s) \to (x_t, y_t) = (\frac{x_s - c_2}{y_s}, \frac{x_s - c_3}{x_s + c_4}), \quad (4)$$

with $c_1 = (a-d)/4$, $c_2 = (a+d)/6$, $c_3 = (5a-d)/12$, and $c_4 = (a-5d)/12$.

The original GLV method works on some elliptic curves in Weierstrass model with special complex multiplication, e.g. curves having CM discriminant $D = -3, -4, -7, -8$, etc. If there exists an efficient endomorphism ϕ on the curve E_s, then we can obtain an corresponding endomorphism ϕ_t on the birationally-equivalent twisted Edwards curve $E_{a,d}$ as $\psi^{-1}\phi\psi$. Thus, the GLV method is also applicable on twisted Edwards curves with some efficient endomorphism.

Usually, the computation of endomorphisms in the short Weierstrass model is much simpler than in the twisted Edwards model. In the following, we take the most common cases of "GLV friendly" curves (namely curves with j-invariant 0 and 1728) as examples.

J-invariant 0 Case. This class of elliptic curves has CM discriminant $D = -3$ and can be represented by a Weierstrass equation of the form

$$E_b : y^2 = x^3 + b \quad (5)$$

over a prime field $\mathbb{F}_p$ with $p \equiv 1 \bmod 3$, which means $\mathbb{F}_p$ contains an element β of order 3. In this case, the map $\phi : E_b \to E_b$ given by $(x, y) \mapsto (\beta x, y)$ and $\mathcal{O} \mapsto \mathcal{O}$ is an endomorphism defined over $\mathbb{F}_p$. If $G \in E_b(\mathbb{F}_p)$ is a point of prime order n, then $\phi(G) = \lambda \cdot G = (\beta x, y)$, where λ is an integer satisfying $\lambda^2 + \lambda + 1 \equiv 0 \bmod n$. There are only six possible group orders for such curves for a given field $\mathbb{F}_p$.

We can find a twisted Edwards curve birationally equivalent to the GLV curve E_b with help of the equation for the j-invariant: $j(E_{a,d}) = 0$ requires $a^2 + 14ad + d^2 = 0$, and when we fix a to -1 then $d = -7 \pm 4\sqrt{3}$. Thus, we can obtain an endomorphism on its birationally equivalent twisted Edwards curve $E_{a,d}$ as

$$\phi_t(x, y) = (\frac{x(c_5 y + c_6)}{y+1}, \frac{c_7 y + c_8}{y + c_9}), \quad (6)$$

where $c_5 = \frac{5d\beta - 2d + \beta + 2}{3(d+1)}$, $c_6 = \frac{d\beta + 2d + 5\beta - 2}{3(d+1)}$, $c_7 = \frac{5d\beta + d + \beta + 5}{(5d+1)(\beta-1)}$, $c_8 = \frac{5+d}{5d+1}$ and $c_9 = \frac{d\beta + 5d + 5\beta + 1}{(5d+1)(\beta-1)}$.

J-invariant 1728 Case. Elliptic curves with a j-invariant of 1728 have CM discriminant $D = -4$ and can be defined by a Weierstrass equation of the form

$$E_a : y^2 = x^3 + ax \quad (7)$$

over a prime field $\mathbb{F}_p$ with $p \equiv 1 \bmod 4$, i.e. it is guaranteed that $\mathbb{F}_p$ contains an element α of order 4. In this case, the map $\phi : E_a \to E_a$ given by $(x, y) \mapsto (-x, \alpha y)$ and $\mathcal{O} \mapsto \mathcal{O}$ is an endomorphism defined over $\mathbb{F}_p$. When $G \in E_a(\mathbb{F}_p)$ is a point of prime order n, then $\phi(G) = \lambda \cdot G = (-x, \alpha y)$, where λ is an integer satisfying $\lambda^2 + 1 \equiv 0 \bmod n$. There are only four possible group orders for such curves when $\mathbb{F}_p$ is fixed.

Similar as before, $j(E_{a,d}) = 1728$ requires $(a + d)(a^2 - 34ad + d^2) = 0$, and when setting $a = -1$, we get $d = 1$ or $d = 17 \pm 12\sqrt{2}$. In the latter case, we can obtain an endomorphism ϕ_t on the corresponding twisted Edwards model $E_{a,d} : -x^2 + y^2 = 1 + dx^2y^2$ in the same way as in the j-invariant 0 case. More concretely, when $d = 17 \pm 12\sqrt{2}$, the endomorphism ϕ_t can be computed as

$$\phi_t(x, y) = (-\frac{x((7d - 1)y + (7 - d))}{3\alpha(d + 1)(y + 1)}, \frac{(2d - 2)y + 5 + d}{(5d + 1)y + (2 - 2d)}). \tag{8}$$

On the other hand, if $d = 1$, ϕ_t has a particularly simple formula, namely

$$\phi_t(x, y) = (\alpha x, 1/y). \tag{9}$$

The computation of ϕ_t requires only one multiplication and one inversion in $\mathbb{F}_p$. Consequently, computing ϕ_t on $E_{-1,1} : -x^2 + y^2 = 1 + x^2y^2$ is much simpler (and faster) than computing the endomorphism(s) on the twisted Edwards GLV-GLS curves given in [20, Section 5].

3 Curve Generation

3.1 CM Method

Let $E/\mathbb{F}_p$ be our desired elliptic curve with CM discriminant D. The group order of $E/\mathbb{F}_p$ is $\#E(\mathbb{F}_p) = p + 1 - t$, where t is the Frobenius trace. We also have the CM equation as $4p = t^2 - Ds^2$, where $s \in \mathbb{Z}$. Note that the j-invariant of such curve is also determined, and there are only 2 or 4 or 6 possible group orders for desired curve. Thus, the goal of the curve generation is not to find curve parameters (since we have them already), but rather to find a prime field $\mathbb{F}_p$, and then a twisted Edwards curve defined over $\mathbb{F}_p$ (given by $a = -1$ and some fixed d) which contains a large cyclic subgroup and meets other security requirements. This contrasts with the “traditional” approach for curve generation where the field $\mathbb{F}_p$ is fixed and one has to find suitable curve parameters.

3.2 Example Curve

We choose elliptic curve with CM discriminant $D = -4$ as our example. If we fix $a = -1$ for efficiency reasons [1], then by the analysis in Section 2.3, the possible value of d is 1 or $17 \pm 12\sqrt{2}$. We finally choose $d = 1$ since in this case the endomorphism on $E_{-1,1}$ has a very simple formula as analyzed before.

Our example curve is

$$E_{-1,1}/\mathbb{F}_p : -x^2 + y^2 = 1 + x^2y^2,$$

where the prime $p = 2^{207} - 5131$. Note that $p \equiv 1 \bmod 4$, then $E_{-1,1}$ is ordinary. The group order $\#E_{-1,1}(\mathbb{F}_p) = 8 \cdot n$, where $n =$ 0xFFFFFFFFFFFFFFFFFFFF FFFFFFFE090B67A2AE9D8EC7DD7009F95 is a 204-bit prime. Thus our curve is at around 100-bit security level. The embedding degree of $E_{-1,1}/\mathbb{F}_p$ with respect to n is $n-1$, which means that it is resistant to the FR-MOV attack.

There is an efficient endomorphism ϕ_t on $E_{-1,1}$ as

$$\phi_t(x, y) = (\alpha \cdot x, 1/y),$$

where $\alpha =$ 0x5135DD9F4EBC5D1835EFB3D377F3A4A1FCB1E2DEC2911FF 2B59A satisfies $\alpha^2 + 1 \equiv 0 \bmod p$. And we can check that $\phi_t(G) = [\lambda]G$ for $G \in E_{-1,1}(\mathbb{F}_p)$ with $\lambda =$ 0xA1D776BEDB1ECFFCE5ABB8F12F8223CC0F494 D461EC0F724D06, here $\lambda^2 + 1 \equiv 0 \bmod n$.

4 Double-Base Scalar Multiplication

As mentioned before, double-base scalar multiplication is the most time-consuming operation of ECDSA signature verification and, therefore, deserves efficient implementation and optimization. Formally, *double-base scalar multiplication* is an operation of the form $k \cdot G + l \cdot Q$ and computes the sum of two scalar products, where G is fixed and Q is an arbitrary point. In the following, we review several approaches for performing the double-base scalar multiplication and describe how to speed up this operation by exploiting an endomorphism. For convenience, we assume that both scalars k and l are exactly m bits long.

Two Single Scalar Multiplications. The most straightforward method to perform the double-base scalar multiplication is to do the two single scalar multiplications separately and then add up the results. The first scalar multiplication $k \cdot G$ takes a fixed and a-priori-known point as input, which can be efficiently performed through the fixed-base comb method as described in [11, Section 3.3.2]. This single scalar multiplication requires roughly m/w point doublings and $\frac{m(2^w-1)}{w \cdot 2^w}$ point additions when using $2^w - 1$ pre-computed points, where w is the window size. The second scalar multiplication, $l \cdot Q$, is performed with an arbitrary base point Q not known in advance. When using a binary method, this arbitrary-base scalar multiplication requires to execute m point doublings and $m/2$ point additions in average. In total, the double-base scalar multiplication requires *roughly* $m + m/w$ point doublings and $m/2 + \frac{m(2^w-1)}{w \cdot 2^w}$ point additions.

Interleaving Method. A method to speed up the computation of $k \cdot G + l \cdot Q$ is to perform them in a simultaneous (or interleaved) fashion by using a technique known as *Shamir's Trick* [11, p. 109]. This method first computes the sum of G and Q, i.e. $S = G + Q$. Then, the scalars k and l are scanned from the most significant bit downward in a simultaneous way. Using this method, the number of point doubling can be reduced, so that a double-base scalar multiplication requires *roughly* m point doublings and $3m/4$ point additions on average.

Joint Sparse Form. Solinas [31] proposed a joint sparse form (JSF) representation of a pair of integers, which has the minimal joint Hamming weight. This representation can thus be used to speed up double-base scalar multiplication $k \cdot G + l \cdot Q$ by pre-computing $S = G + Q$ and $D = G - Q$. When taking advantage of JSF representation for the scalars k and l, a double-base scalar multiplication performed in an interleaved fashion requires *roughly* m point doublings and $m/2$ point additions (resp. subtractions) [11, Section 3.3.3].

(Sliding) Window Method. Another approach to reduce the number of point additions in a double-base scalar multiplication is to use a window method. Given the fixed window width w, a double-scalar multiplication first generates a look-up table with points $i \cdot G + j \cdot Q$ for all $i, j \in [0, 2^w - 1]$, and then scans w columns of the scalars k and l. This method requires storage of $2^{2w} - 1$ points and then a double scalar multiplication can be performed with roughly $(m/w - 1) \cdot w$ point doublings and $\frac{m(2^{2w}-1)}{w \cdot 2^{2w}} - 1$ point additions. The window method can be further improved by using a "sliding" window; in this case, only $2^{2w} - 2^{2(w-1)}$ points are actually needed for the look-up table, and the number of point additions is reduced to $\frac{m}{w+(1/3)}$ [11].

Double-Base Scalar Multiplication with Endomorphism

A maximum of m point doublings can be saved for a double-base scalar multiplication when using the above described approaches. In the following, we present a strategy to further reduce the number of point doublings by some 50% using an efficiently computable endomorphism.

The main idea is to compute a double-base scalar multiplication, i.e., $k \cdot G + l \cdot Q$, through four simultaneous scalar multiplications $k1 \cdot G$, $k2 \cdot \phi(G)$, $l1 \cdot Q$ and $l2 \cdot \phi(Q)$, where $k1$, $k2$, $l1$ and $l2$ are roughly $m/2$ bits long. Algorithm 1 shows the computation of double-base scalar multiplication exploiting an efficiently-computable endomorphism. We first split the scalar k into two parts $k1$ and $k2$ using [11, Algorithm 3.74], where $k1$ and $k2$ have roughly half of the bitlength of k; the second scalar l can be decomposed into $l1$ and $l2$ in the same way. Then, we calculate the points $\phi(G)$ and $\phi(Q)$ from G and Q by using the formula in Section 2, which requires one inversion and a few multiplications. After that, we generate the look-up table with 15 points (line 4). Finally, the four scalar multiplications $k1 \cdot G + k2 \cdot \phi(G) + l1 \cdot Q + l2 \cdot \phi(Q)$ is performed simultaneously, i.e. in an interleaved fashion. A double-base scalar multiplication using Algorithm 1 requires *approximately* $m/2$ point doubling and $15m/32 + 11$ point additions including the overhead for the generation of the look-up table.

5 Hardware Architecture and Implementation

As mentioned earlier, we adopt a pseudo-Mersenne prime of the form $p = 2^k - c$ for our implementation, where c is chosen to fit into one word of the target

Algorithm 1. Double-base scalar multiplication using an endomorphism

```
Input: Two $m$-bit scalars $k$ and $l$, the fixed base point $G$ and an arbitrary point $Q$ on
    the curve $E(\mathbb{F}_p)$ with endomorphism.
Output: Double-base scalar multiplication $k \cdot G + l \cdot Q$.
1: Use [11, Algorithm 3.74] to find $(k1; k2)$ of $k$ and $(l1; l2)$ of $l$.
2: Compute $\phi(G)$, $\phi(Q)$ using $G$ and $Q$.
3: $G = (k1 > 0)?G : -G$; $\phi(G) = (k2 > 0)?\phi(G) : -\phi(G)$; $Q = (l1 > 0)?Q : -Q$;
   $\phi(Q) = (l2 > 0)?\phi(Q) : -\phi(Q)$;
4: Generate look-up table $T$ with 15 points such that $T[i-1] = [(i \gg 3)\&1] \cdot \phi(Q) +$
   $[(i \gg 2)\&1] \cdot Q + [(i \gg 1)\&1] \cdot \phi(G) + (i\&1) \cdot G$ for $1 \leq i \leq 15$.
5: Let $k1 = |k1|$, $k2 = |k2|$ , $l1 = |l1|$, $l2 = |l2|$ and $h = max\{k1, k2, l1, l2\}$.
6: $R = \infty$.
7: for $i$ from $h$ by 1 down to 0 do
8:     $R \leftarrow 2R$;
9:     $s \leftarrow 8 \cdot l2_i + 4 \cdot l1_i + 2 \cdot k2_i + k1_i$;
10:    if $s > 0$ then
11:        $R \leftarrow R + T[s-1]$.
12:    end if
13: end for
14: return $R$.
```

platform (i.e. c can be at most 16 bits in our case since we use a 16-bit datapath). The basic idea of fast reduction using a pseudo-Mersenne prime is to apply the congruence relation $2^k \equiv c \bmod p$ repetitively during the reduction process. Suppose $z = z_H 2^k + z_L$ is a $2k$-bit integer, such as obtained as result of a multiplication of two k-bit integers. We can reduce z with respect to p as follows

$$z = z_H 2^n + z_L \bmod p \equiv z_H c + z_L \bmod p \tag{10}$$

Now z is already only slightly longer than k bits since c is small. To complete the reduction $z \bmod p$, we perform the multiplication by c in the same way and then at most one subtraction of p is needed to get a result that is at most k bits long. Our implementation uses the pseudo-Mersenne prime $p = 2^{207} - 5131$.

Notation. We use the following notation in this paper:

- n: the operand size (i.e. $n = 207$).
- w: the word size of the underlying processor (i.e. $w = 16$).
- m: bitlength of the scalar, i.e. the bitlength of the order of the generator G.
- A, B: two operands; $A[i : j]$ represents bits at position i to j of operand A.
- R: product $A \cdot B$, which is twice long as operand A or B.

Our implementation adopts the idea of incomplete modular reduction as discussed, for example, in [33], which means the arithmetic functions described in the following subsections do not not necessarily reduce the result to an integer in the range of $[0, p-1]$, but only ensure that the result is smaller than 2^n so that it fits into n/w words. Also, all arithmetic functions accept incompletely reduced inputs of n/w words.

Note that all arithmetic operations (except the Montgomery inverse[5]) we discuss in the following can be easily implemented in a highly regular fashion so that their execution time is completely independent of the actual value of the operands. Such constant execution time helps to thwart certain implementation attacks like timing analysis. Even though signature verification does not involve any secret values (and can, therefore, not leak any secrets), it still makes sense to implement the underlying field arithmetic in a regular way so that it can also be used for signature generation.

5.1 Modular Multiplication and Squaring

The modular multiplication is performed in three basic steps as shown in Algorithm 2. First, a conventional multi-precision multiplication is performed in a word-wise fashion based on the product-scanning technique as described in e.g. [11]. Then, we multiply the most significant 209 bits of the product by c and add the result to least significant 207 bits, which yields a result of (at most) 226 bits length. Finally, we multiply the most significant 19 bits by c and add the product to least significant 207 bits; the result is now at most 208 bits and, therefore, fits into m words. In order to achieve constant execution time, we always execute both reduction steps, even when the result is already fully reduced after the first step.

Algorithm 2. Modular multiplication for $p = 2^{207} - c$

```
Input: Two integers $A[207:0]$, $B[207:0]$, and modulus $p$
Output: $R = A \cdot B \bmod p$
1: $R = A \cdot B$
2: $R = R[415:207] \cdot c + R[206:0]$ {The 1st reduction}
3: $R = R[225:207] \cdot c + R[206:0]$ {The 2nd reduction}
```

A modular squaring can be done more efficiently thanks to the symmetry of partial products. Thus, it is possible to save the computation of (nearly) half of the partial products.

5.2 Modular Inversion

Modular inversion is the most time-consuming field arithmetic operation. Traditionally, the Extended Euclidean Algorithm (EEA) [11], Fermat's technique [11], and the Montgomery modular inversion algorithm [15, 28] are used to compute an inverse. Our inversion is mainly based on the Montgomery modular inverse, but has been optimized for the pseudo-Mersenne prime $p = 2^n - c$.

[5] The scalar multiplication performed in a signature generation process requires a constant-time inversion for the projective-to-affine conversion, which can be implemented based on Fermat's theorem as described in the appendix.

Algorithm 3. Optimized Montgomery Modular Inversion for $2^n - c$

```
Input: $a \in [1, 2^n)$ and is odd, $p > 2$ is a $n$ bits prime, precomputed $T = 2^{(-2n)} \bmod p$;
Output: $R \in [1, 2^n)$, where $R = a^{-1} \bmod p$
1: //Phase I
2: $u = -p$, $v = a$, $r = 0$, $s = 1$, $k = 0$
3: while (1) do
4:     $x = u + v$ {Both $u$ and $v$ are always odd number}
5:     $y = r + s$
6:     $tlz_x = DET(x)$ {Trailing zero detection}
7:     if $x == 0$ then
8:         break;
9:     else if $x < 0$ then
10:        $u = x >> tlz_x$ {Right-shift operation can be done in parallel with $u + v$}
11:        $r = y$
12:        $s = s << tlz_x$ {Left-shift operation can be done in parallel with $r + s$}
13:    else
14:        $v = x >> tlz_x$
15:        $s = y$
16:        $r = r << tlz_x$
17:    end if
18:    $k = k + tlz_x$
19: end while
20: //phase II
21: $s = s \cdot 2^{(2n-k)} \bmod p$
22: $s = s \cdot T \bmod p$
23: return $R = s$
```

As shown in Algorithm 3, our inversion consists of phases; phase I and phase II. In phase I, we firstly perform two additions, and then update the variables $\{u, v, r, s, k\}$ according to the sign flag of x. The trailing zero detection (DET) and right-shift operation $x >> tlz_x$ can be done in parallel with the addition of $u + v$. Furthermore, the left-shift operation of $s << tlz_x$ and $r << tlz_x$ can be done in parallel with the addition of $y = r + s$. In phase II, we perform two ordinary multiplications to get the modular inverse. The input a is set to be odd, but even if initially a is even, it can be easily changed to be odd via a modular subtraction $p - a$. The core idea behind our optimized inversion is to remove all trailing zeros of $(u + v)$ in every iteration, which keeps u and v always odd so that $(u + v)$ converges to zero quickly.

Compared to the Multibit Shifting method proposed by Savaş et al in [29], we remove all those iterations for shift operation (i.e. the iterations when u or v is even in [15, Algorithm MONTINVER]) and adopt the idea from [21] to avoid a complex comparison step by using the sign flag of x. More specifically, the number of total iterations in Phase I of [15] is in the range of $[n, 2n]$, with 50% shift operations inside. For comparison, the number of iterations of our algorithm is in the range of $[0.5n, n]$ since no such shift-operation iterations are required. Furthermore, the optimized inversion can be even faster if we keep track of the length of variables $\{u, v, r, s\}$ to save cycles for addition, since the word lengths decrease linearly with the number of iterations.

5.3 Modular Addition and Subtraction

An addition modulo $p = 2^{207} - c$ can be performed in three steps. First, a conventional multi-precision addition $R = A + B$ is performed in a word-wise fashion. Then, for reduction, we reduce the 209-bit result to 208 bits by using Equation (11). To ensure constant execution time, we perform the addition step and the reduction step for all possible inputs, even if no reduction is required.

$$R = R[209:207] \cdot 2^{207} + R[206:0] \bmod p = R[209:207] \cdot c + R[206:0] \quad (11)$$

For modular subtraction, a conventional multi-precision subtraction $R = A - B$ is performed through word-wise subtract-with-borrow operations. As the 208-bit input B can be bigger than $2p$, the result of the subtraction may be smaller than $-2p$ and, thus, up to two addition steps will be needed. As shown in Equation (12), the first addition step will guarantee that $R > -p$. If R is still negative, another addition step as shown in Equation (13) will make R a positive number in the range of $[0, 2^{208})$. To ensure constant execution time, we perform one subtraction and two additions for all possible inputs, but when R is positive after the subtraction, the words of the subtrahend with be masked out (i.e. set to 0) so that the value of R does not change.

$$R = R + 2p \quad (12)$$

$$R = R + p \quad (13)$$

5.4 Hardware Architecture

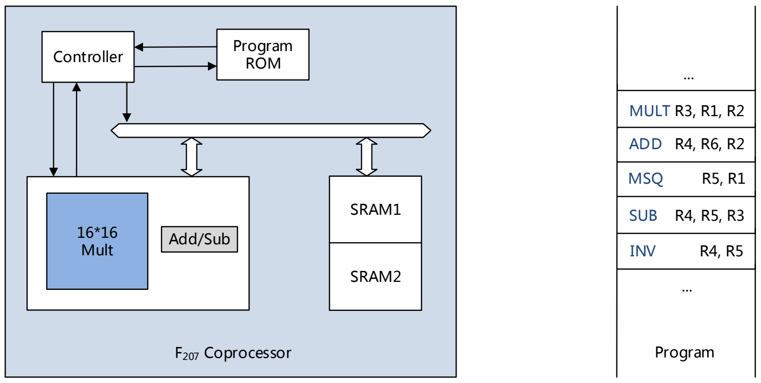

Fig. 1. Hardware architecture

The hardware architecture, as shown in Fig 1, consists of a micro-controller, a program ROM, an $\mathbb{F}_p$-coprocessor, which we call Prime-Field Arithmetic Unit (i.e. PFAU), and two dual ports SRAMs. The program ROM is used to command sequences that execute high-level functions such as pre-computations, point addition, point doubling, etc. This section focuses on the ALU.

The architecture of the ALU and other important modules is shown in Figure 2, where one (16×16)-bit multiplier, one 3-input adder, a trailing-zero detection module (`tlz`), a left-shifting module (`lshifter`), and a right-shifting module (`rshifter`) are depicted. We decided to implement a 16-bit datapath since previous research has shown that this allows one to achieve a good trade-off between performance and silicon area. The ALU supports the word-level instructions needed for modular multiplication, modular squaring, modular inversion, modular addition and modular subtraction. The critical path goes from the input registers of the multiplier to the output registers of the adder. The input from `mult` to the adder is 33 bits long due to the fact that we need to double some partial products when performing a modular squaring.

The optimized modular inversion requires the `tlz`, `lshifter`, `rshifter` modules. Using the implementation technique from [23], the `tlz` module can output the number of trailing zeros of a word (16 bits) in one clock cycle. To obtain the trailing zeros in a 208-bit operand, we can perform a zero detection word by word. If the number of trailing zeros exceeds one word, the detection process will take more than one cycle, but the probability is only 2^{-16} that we have a number with 16 trailing zeros (in fact, the probability is 2^{-15} in our case since x is always even in Algorithm 3). The `lshifter` and `rshifter` receive the output of `tlz` perform the corresponding number of shifts on the 16-bit number input. As mentioned before, the shift operation in the modular inverse can be done in parallel with the addition.

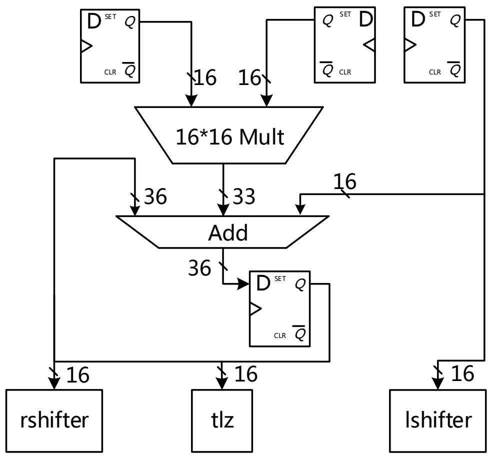

Fig. 2. ALU architecture

All operations except modular inversion are designed for constant time, which also means worst-case calculation cycles. The execution time of modular inversion are evaluated based on the average number of Phase I iterations, with two additions per iteration and two modular multiplications in Phase II.

Microcode Based Architecture An alternative implementation option, which allows one to significantly reduce the ALU area, is to adopt a microcode based architecture, as mentioned in e.g. [13, 32, 24]. The ALU described in these works only supports word-size operations, such as 16 bits addition, subtraction, multiplication, NOT, AND, OR and XOR. When following this approach, a microcode represents a sequence of commands sent to the ALU to perform certain operations such as modular multiplication, modular addition, etc. Every operation corresponds to a sequence of commands, which will be stored in ROM. Thus, complex control logic for the field arithmetic, which is needed in the case of a hardwired in ALU, is mitigated to an upper software layer, which finally ends up in more ROM area for microcode storage. Considering that the size of ROM can be well optimized, this method can result in an overall area saving, but at the expense of a certain performance loss because due to inefficiency. Taking [13] as an example, the described implementation needs at least 32 clock cycles to perform a 192-bit modular addition, without considering constant execution time. In summary, if one is willing to sacrifice some speed for a less complex ALU, then a microcode-based architecture may be an excellent option. According to our experiments, we could reduce the ALU area by roughly 30% when following a microcode approach.

6 Implementation Results

We implemented the arithmetic processor in Verilog and synthesized it with Design Compiler 2013.12 using the UMC 0.13μ 1P8M Low Leakage Standard cell Library with typical values (i.e. voltage of $1.2V$ and temperature of $25°C$). The area (in gate equivalents, GE) after placement and routing is calculated by dividing the overall area by the area of a single two-input NAND gate. The design has been synthesized for a clock frequency of $50MHz$, which is more than sufficient for common IoT devices such as FRID tags or sensor nodes.

6.1 Execution Time of Field Arithmetic

As mentioned in the previous section, we implemented the multiplication, squaring, addition and subtraction to have constant execution time. Table 1 summarizes the execution times of the five basic arithmetic operations modulo the prime $2^{207} - 5131$. The modular addition takes exactly 30 cycles, which is faster than the modular subtraction. Our constant-time modular multiplication executes in exactly 192 cycles, whereas the modular squaring has an execution time of 120 clock cycles, which means the squaring requires merely 60% of the multiplication cycles. Thanks to the optimized Montgomery modular inversion proposed in Algorithm 3, our inversion requires 4452 clock cycles in average, which corresponds to only 23 multiplications.

Table 1. Execution time of field arithmetic operation using 16×16 multiplier over $p = 2^{207} - 5131$ (in clock cycles)

Operation	`Mul`	`Sqr`	`Inv`	`Add`	`Sub`
This work	198	120	4452	30	43

6.2 Trade-offs between Performance and Memory

Table 2 reports the execution time and RAM requirements of double-base scalar multiplication for several different approaches as outlined in Section 4, as well as a combination of endomorphism and window method. Compared to the implementation using (1), a double-base scalar multiplication using a combination of (1) and (2) requires the same number of point doubling while it saves approximately 1/4 of the point additions. The number of point additions can be further reduced by using a combination of (1) and (3) with a look-up table of $2^{2w} - 1$ points. Taking the window width $w = 2$ as an example, one can save roughly 1/16 of the point additions compared to the implementation with a combination of (1) and (2). In relation to a combination of (1) and (3), the number of point doublings can be further reduced by some 50% using the technique of (4) with the same RAM occupation. A small number of point additions may potentially be saved by using a combination of (3) + (4). However, one has to consider that the look-up table will increase exponentially and a combination of (3) and (4) is only able to save point additions when n is big enough. For example, given $w = 2$, a double-base scalar multiplication using a combination of (3) and (4) requires a look-up table of 255 points and even requires more point doublings and point additions. Taking both performance and RAM requirements into account, the technique (4) (i.e. endomorphism) is the best choice to speed up the double-base scalar multiplication on resource-constraint platforms.

6.3 High-Speed Version vs Memory-Efficient Version

A double-base scalar multiplication $k \cdot G + l \cdot Q$ using endomorphism ϕ requires to compute four simultaneous scalar multiplications of the form $k1 \cdot G + k2 \cdot \phi(G) + l1 \cdot Q + l2 \cdot \phi(Q)$. In order to analyze the trade-offs between performance and RAM requirements in more detail, we study two implementations; the first one is optimized for performance, while the second is optimized for low RAM footprint. We describe the details of these implementations using the curve $-x^2 + y^2 = 1 + x^2y^2$ over $p = 2^{207} - 5131$ as follows.

Table 2. Comparison of execution time (including the generation of look-up table) and RAM requirements of double-base scalar multiplication using different approaches.

Method	`Storage`	`Pnt Dbl`	`Pnt Add`
(1)	3	m	$1 + 3m/4$
(1) + (2)	4	m	$2 + m/2$
(1) + (3)	$2^{2w} - 1$	$(2^{2(w-1)} - 2^{w-1}) + m - w$	$(3 \cdot 2^{2(w-1)} - 2^{w-1} - 1) + \frac{m \cdot 2^{2w} - 1}{w \cdot 2^{2w}}$
(4)	$2^4 - 1$	$m/2 - 1$	$11 + \frac{15m}{32}$
(3) + (4)	$2^{4w} - 1$	$(2^{4(w-1)} - 2^{w-1}) + m/2 - w$	$(15 \cdot 2^{4(w-1)} + 2^{w-1} - 5) + \frac{m \cdot (2^{4w} - 1)}{2w \cdot 2^{4w}}$

(1): Interleaved; (2): JSF; (3): Window; (4): Endomorphism.

Speed-Optimized. The speed-optimized implementation requires a look-up table containing 15 points, of which 11 points (except G, Q, $\phi(G)$ and $\phi(Q)$) will be generated by a sequence of point additions. In order to take the advantage of the efficient point addition formula on a twisted Edwards curve (i.e. the $7M$ mixed addition formulae based on [12]), we store these points in extended affine coordinate of the form (U, V, W), where $U = (x + y)/2$, $V = (y - x)/2$, $W = xy$ (in our case $d = 1$). A straightforward method to get the affine form of these points would require 11 inversions. For reducing the number of inversions, we perform the 11 inversions in a simultaneous way based on the observation that $1/x = y(1/xy)$ and $1/y = x(1/xy)$ [11, page 44]. With the help of three temporary variables, the 11 inversions can be computed by only one inversion and 83 multiplications. Given an affine point, the extended affine coordinates (U, V, W) can be obtained by performing one addition, one subtraction and one multiplication. In the main loop, a pre-computed point in extended affine coordinates will be used as operand in each iteration (i.e. line 7-13 of Algorithm 1). As a result, our speed-optimized double-base scalar multiplication requires an execution time of 365,082 clock cycles with a RAM footprint of 1612 bytes.

For comparison, a double-base scalar multiplication without exploiting the endomorphism (i.e. using just the straightforward simultaneous approach along with a JSF representation of the scalars) has a execution time of 454179 cycles, i.e. using the endomorphism yields a speed-up of roughly 20%.

Memory-Optimized. A look-up table with 15 extended affine points requires has a RAM consumption of 45 field elements in total. Instead of generating a look-up table with extended affine points, the memory-optimized implementation generates a look-up table with standard affine x, y coordinates in order to reduce RAM requirements. In the process of look-up table generation, we adopt the point addition formula with $Z1 = 1$ and $Z2 = 1$ [12, Section 3.1] and directly convert the projective representation into standard affine representation for each point. In total, the look-up table generation requires 11 point additions, 11 inversions and 22 multiplications. On the other hand, in the main loop of double-base scalar multiplication, we still use the efficient point addition formula for twisted Edwards curve (i.e. the $7M$ mixed addition formulae based on [12]). Thus, we compute the extended affine representation of an affine point on-the-fly, which requires one multiplication, one addition and one subtraction for each iteration. As a consequence, our memory-optimized double-base scalar multiplication requires an execution time of 415,392 clock cycles with a RAM consumption of only 1222 bytes, which corresponds to a saving of 33% for the look-up table (780 instead of 1170 bytes) and 24% in total (i.e. 1222 instead of 1612 bytes), by sacrificing only about 12.1% of performance.

6.4 Comparison with Other Implementations

Table 3. Comparison of execution time, area and RAM consumption with related work over prime fields. Most of the work used a 16-bit datapath.

Implementations	Order	Time (Cycles)		ALU (GEs)	SRAM (Bytes)
		Sig.	Ver.		
Chen et al [4]	256	$562K$	$1124K^1$	n.a.	n.a.
Lai et al [17]^2	176	$93,399$	$186,798^1$	n.a.	n.a.
Lai et al [17]^2	256	$252,067$	$504,134^1$	n.a.	n.a.
Satoh et al [27]	192	$1,362,906$	$2,725,812^1$	$9,456$	n.a.
Satoh et al [27]	224	$2,048,166$	$4,096,332^1$	$10,800$	n.a.
Furbass et al [8]^3	192	$502K$	$1004K^1$	$21,769$	n.a.
Hutter et al [13]^3	192	$859,188$	$1,718,376^1$	$2,371^4$	256
Wenger et al [32]	192	$1377K$	$2645K$	$4,354^4$	422
Plos et al [24]	192	$863,109$	$1,726,218^1$	$3,608^4$	256
This work^5 (HS)	204	$182,653$	$365,082$	$5,821$	$1,612$
This work^5(ME)	204	$182,653$	$415,392$	$5,821$	$1,222$

^{1}: Estimated results from the execution time of signature implementation.
^{2}: Four 32 bits multipliers used.
^{3}: $0.35\mu m$ technology library used.
^{4}: Microcode based architecture used, more ROM are required.
^{5}: Execution time of fixed-base scalar multiplication (for signature generation) and double-base scalar multiplication (for verification).

Table 3 shows our implementation results and a comparison with related work over prime fields. All related implementations (except Lai et al [17]) used a 16-bit datapath as in our work. We perform the fixed-base scalar multiplication (needed for signature generation) on the chosen twisted Edwards curve using a constant-time comb method with $w = 4$ as described in [18]). Note that signature generation requires a constant-time inversion, which we compute on basis of Fermat's theorem with an addition chain that can be evaluated by only 206 modular squarings and 14 modular multiplications (listed in the appendix). Our implementation requires an execution time of $182,653$ and $365,082$ clock cycles for (constant-time) scalar multiplication and speed-optimized double-base scalar implementation, respectively, and consumes an area of 5821 GEs. On the other hand, the memory-oriented implementation needs $415,392$ clock cycles, while consuming only 1.2 kB of RAM. Since most of the previous implementations only reported the execution time of signature generation, we estimate the cycle count of verification (i.e. double-base scalar multiplication) by simply multiplying the generation time by two. As shown in the Table 3, our implementation is at least three times faster than all the previous works using the same word size. In terms of area, the implementations from [27] and [17] support both prime and binary field arithmetic and, thus, have a large area. On the other hand, the authors of [13, 32, 24] optimized their implementation with the help of a microcode-programmable structure (for field arithmetic) and, thus, their implementations require extra instruction decoding modules and have higher ROM consumption in order to save area in the control logic. Our implementation does not include the area for SRAM since it varies for different process technologies and depends significantly on whether one has a RAM generator available or not. Besides, the needed SRAM may come from shared memory and just be temperately occupied during signature generation or verification.

7 Conclusions

In this work, we first introduced a special twisted Edwards curve with an efficiently computable endomorphism and described how said endomorphism be exploited to speed up double-base scalar multiplication. As second contribution, we presented an area-optimized VLSI design of PFAU, an arithmetic unit featuring a (16×16)-bit multiplier that has an overall silicon area of 5821 gates when synthesized with a 0.13μ standard-cell library. Our PFAU can be clocked with a frequency of up to 50 MHz and is capable to perform a constant-time multiplication in a 207-bit prime field in only 198 clock cycles. We used PFAU as a vehicle to explore different trade-offs between memory and execution time, whereby we used the twisted Edwards curve $-x^2 + y^2 = 1 + x^2y^2$ over the 207-bit prime field $\mathbb{F}_p$ with $p = 2^{207} - 5131$ as case study. In a speed-optimized setting, our implementation requires an execution time of $182,653$ and $365,082$ clock cycles for constant-time fixed-base scalar multiplication and double-base scalar multiplication, respectively. In addition, we showed that our curve supports various trade-offs between execution time and memory requirements, which gives a designer plenty options to optimize double-base scalar multiplication for different requirements.

References

1. D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards curves. In S. Vaudenay, editor, *Progress in Cryptology — AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer Verlag, 2008.
2. D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sept. 2012.

3. S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Möller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). Internet Engineering Task Force, Network Working Group, RFC 4492, May 2006.
4. G. Chen, G. Bai, and H. Chen. A high-performance elliptic curve cryptographic processor for general curves over GF(p) based on a systolic arithmetic unit. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 54(5):412–416, 2007.
5. D. A. Cox. *Primes of the Form* $x^2 + ny^2$. John Wiley & Sons, 1989.
6. T. Dierks and E. K. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. Internet Engineering Task Force, Network Working Group, RFC 5246, Aug. 2008.
7. A. Faz-Hernández, P. Longa, and A. H. Sánchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. In J. Benaloh, editor, *Topics in Cryptology — CT-RSA 2014*, volume 8366 of *Lecture Notes in Computer Science*, pages 1–27. Springer Verlag, 2014.
8. F. Furbass and J. Wolkerstorfer. ECC processor with low die size for RFID applications. In *IEEE International Symposium on Circuits and Systems (ISCAS 2007)*, pages 1835–1838. IEEE, 2007.
9. S. D. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In A. Joux, editor, *Advances in Cryptology — EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 518–535. Springer Verlag, 2009.
10. R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphism. In J. Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer Verlag, 2001.
11. D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer Verlag, 2004.
12. H. Hişil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In J. Pieprzyk, editor, *Advances in Cryptology — ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer Verlag, 2008.
13. M. Hutter, M. Feldhofer, and T. Plos. An ECDSA processor for RFID authentication. In *Radio Frequency Identification: Security and Privacy Issues*, pages 189–202. Springer, 2010.
14. D. Johnson, A. J. Menezes, and S. A. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, July 2001.
15. B. S. Kaliski. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.
16. S. L. Keoh, S. S. Kumar, and H. Tschofenig. Securing the Internet of things: A standardization perspective. *IEEE Internet of Things Journal*, 1(3):265–275, June 2014.
17. J.-Y. Lai and C.-T. Huang. A highly efficient cipher processor for dual-field elliptic curve cryptography. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 56(5):394–398, 2009.
18. Z. Liu, E. Wenger, and J. Großschädl. MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks. In I. Boureanu, P. Owezarski, and S. Vaudenay, editors, *Applied Cryptography and Network Security — ACNS 2014*, volume 8479 of *Lecture Notes in Computer Science*, pages 361–379. Springer Verlag, 2014.

19. P. Longa and C. H. Gebotys. Efficient techniques for high-speed elliptic curve cryptography. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems — CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 80–94. Springer Verlag, 2010.
20. P. Longa and F. Sica. Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. In X. Wang and K. Sako, editors, *Advances in Cryptology — ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 719–739. Springer Verlag, 2012.
21. R. Lórencz and J. Hlaváč. Subtraction-free almost Montgomery inverse algorithm. *Information Processing Letters*, 94(1):11–14, 2005.
22. National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). FIPS Publication 186-4, available for download at `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`, July 2013.
23. V. G. Oklobdzija. An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(1):124–128, 1994.
24. T. Plos, M. Hutter, M. Feldhofer, M. Stiglic, and F. Cavaliere. Security-enabled near-field communication tag with flexible architecture supporting asymmetric cryptography. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(11):1965–1974, 2013.
25. E. K. Rescorla and N. G. Modadugu. Datagram Transport Layer Security Version 1.2. Internet Engineering Task Force, Network Working Group, RFC 6347, Jan. 2012.
26. R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.
27. A. Satoh and K. Takano. A scalable dual-field elliptic curve cryptographic processor. *IEEE Transactions on Computers*, 52(4):449–460, 2003.
28. E. Savaş. The Montgomery modular inverse—Revisited. *IEEE Transactions on Computers*, 49(7):763–766, 2000.
29. E. Savaş, M. Naseer, A.-A. Gutub, and Ç. K. Koç. Efficient unified Montgomery inversion with multibit shifting. *IEE Proceedings–Computers and Digital Techniques*, 152(4):489–498, 2005.
30. N. P. Smart, editor. *ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012).* European Network of Excellence in Cryptology (ECRYPT II), Sept. 2012. Deliverable D.SPA.20, available for download at `http://www.ecrypt.eu.org/documents/D.SPA.20.pdf`.
31. J. A. Solinas. Low-weight binary representations for pairs of integers. Technical Report CORR 2001-41, Centre for Applied Cryptographic Research (CACR), University of Waterloo, Waterloo, Canada, 2001.
32. E. Wenger, M. Feldhofer, and N. Felber. Low-resource hardware design of an elliptic curve processor for contactless devices. In *Information Security Applications*, pages 92–106. Springer, 2011.
33. T. Yanık, E. Savaş, and Ç. K. Koç. Incomplete reduction in modular arithmetic. *IEE Proceedings – Computers and Digital Techniques*, 149(2):46–52, Mar. 2002.

A Constant-time Inversion over $p = 2^{207} - 5131$

The constant time inversion modulo $p = 2^{207} - 5131$ can be evaluated via only 206 modular squarings and 14 modular multiplications. One can compute $a =$

$z^{-1} \equiv z^{2^{207}-5133} \bmod p$ by the following steps (where the annotations after the # denote the value of the exponent and the cost in each step).

$$
\begin{aligned}
z_3 &\leftarrow z^2 \cdot z && \# \ 3, 1S + 1M \\
z_6 &\leftarrow z_3^{2^1} && \# \ 6, 1S \\
z_{15} &\leftarrow z_6^{2^1} \cdot z_3 && \# \ 15, 1S + 1M \\
z_{31} &\leftarrow z_{15}^{2^1} \cdot z && \# \ 2^5 - 1, 1S + 1M \\
t_0 &\leftarrow z_{31}^{2^4} && \# \ 2^9 - 2^4, 4S \\
t_1 &\leftarrow t_0^{2^4} \cdot z_{15} && \# \ 2^9 - 1, 1M \\
t_2 &\leftarrow t_1^{2^9} && \# \ 2^{18} - 2^9, 9S \\
t_3 &\leftarrow t_2 \cdot t_1 && \# \ 2^{18} - 1, 1M \\
t_4 &\leftarrow t_3^{2^5} \cdot z_{31} && \# \ 2^{23} - 1, 5S + 1M \\
t_5 &\leftarrow t_4^{2^{23}} \cdot t_4 && \# \ 2^{46} - 1, 23S + 1M \\
t_6 &\leftarrow t_5^{2^{46}} \cdot t_5 && \# \ 2^{92} - 1, 46S + 1M \\
t_7 &\leftarrow t_6^{2^{92}} \cdot t_6 && \# \ 2^{184} - 1, 92S + 1M \\
t_8 &\leftarrow (t_7^{2^{14}} \cdot z_{15} \cdot z_6)^{2^4} && \# \ (2^{198} - 2^{14} + 21) \cdot 2^4, 18S + 2M \\
t_9 &\leftarrow (t_8 \cdot t_2)^{2^5} \cdot t_0 \cdot z_3 && \# \ 2^{207} - 5133, 5S + 3M
\end{aligned}
$$